Contents

Referência da linguagem C

13/05/2021 • 2 minutes to read

A *referência da linguagem C* descreve a linguagem de programação C como implementada em Microsoft C. A organização do manual baseia-se no padrão ANSI C (às vezes conhecido como C89) com material adicional sobre as extensões da Microsoft para o padrão ANSI C.

- Organização da referência da linguagem C

Para material de referência adicional sobre C++ e o pré-processador, consulte:

- Referência da linguagem C++
- Referência do pré-processador

As opções do compilador e do vinculador estão documentadas na Referência de compilação C/C++.

Veja também

Referência da linguagem C++

Organização da referência da linguagem C

13/05/2021 • 2 minutes to read

Veja também

Referência da linguagem C

Escopo deste manual

13/05/2021 • 2 minutes to read

C é uma linguagem flexível que deixa várias decisões de programação a seu critério. De acordo com essa filosofia, o C têm poucas restrições assuntos como a conversão de tipos. Embora essa característica da linguagem possa tornar seu trabalho de programação mais fácil, você deve saber bem a linguagem para entender como os programas se comportarão. Este manual fornece informações sobre os componentes e recursos da linguagem C da implementação da Microsoft. A sintaxe para a linguagem C é ANSI X3.159-1989, *American National Standard for Information Systems – Programming Language – C* (chamado aqui de padrão ANSI C), embora não faça parte do padrão ANSI C. Resumo da sintaxe da linguagem C fornece a sintaxe e uma descrição de como ler e usar as definições de sintaxe.

Este manual não aborda a programação com a linguagem C++. Para obter mais informações sobre a linguagem C++, consulte Referência da linguagem C++.

Veja também

Organização da referência da linguagem C

Conformidade com ANSI

13/05/2021 • 2 minutes to read

O Microsoft C está em conformidade com o padrão da linguagem C, como mostrado na edição de 9899:1990 do padrão ANSI C.

As extensões da Microsoft para o padrão ANSI C são observadas no texto e na sintaxe deste manual, além de referência online. Como as extensões não são uma parte do padrão ANSI C, seu uso pode restringir a portabilidade dos programas entre sistemas. Por padrão, as extensões da Microsoft estão habilitadas. Para desabilitar as extensões, especifique a opção do compilador /Za. Com /Za, todo código que não for ANSI gerará erros ou avisos.

Veja também

Organização da referência da linguagem C

Elementos de C

13/05/2021 • 2 minutes to read

Esta seção descreve os elementos da linguagem de programação C, incluindo os nomes, os números e os caracteres usados para construir um programa em C. A sintaxe ANSI C rotula os tokens desses componentes.

Esta seção explica como definir tokens e como o compilador os avalia.

Os seguintes tópicos são abordados:

- Tokens
- Comentários
- Palavras-chave
- Identificadores
- Constantes
- Literais de cadeia de caracteres
- Pontuação e caracteres especiais

A seção também inclui tabelas de referência para trigrafos, limites em Floating-Point constantes, limites de inteiro C e C++e sequências de escape.

Operadores são símbolos (caracteres individuais e combinações de caracteres) que especificam como os valores devem ser manipulados. Cada símbolo é interpretado como uma única unidade, chamada de token. Para obter mais informações, consulte Operadores.

Veja também

Referência da linguagem C

Tokens C

13/05/2021 • 2 minutes to read

No programa de origem C, o elemento básico reconhecido pelo compilador é o "token". Um token é o texto do programa de origem que o compilador não divide em elementos componentes.

Sintaxe

token: *keyword*

identifier

constant

literal de cadeia de caracteres

operator

punctuator

> **NOTE**
> Consulte o resumo da sintaxe de linguagem C Introduction to para obter uma explicação das convenções de sintaxe ANSI.

As palavras-chave, os identificadores, as constantes, os literais de cadeia de caracteres e os operadores descritos nesta seção são exemplos de tokens. Caracteres de pontuação como colchetes ([ ]), chaves ({ }), parênteses (**(** **)**) e vírgulas (,) também são tokens.

Veja também

Elementos de C

Caracteres de espaço em branco

13/05/2021 • 2 minutes to read

Os caracteres de espaço, tabulação, alimentação de linha (nova linha), retorno de carro, avanço de página e tabulação vertical são chamados de "caracteres de espaço em branco" porque têm a mesma finalidade dos espaços entre palavras e linhas em uma página impressa – facilitam a leitura. Os tokens são delimitados (limitados) por caracteres de espaço em branco e por outros tokens, como operadores e pontuação. Ao analisar o código, o compilador C ignora caracteres de espaço em branco a menos que você os use como separadores ou como componentes de constantes de caracteres ou de literais de cadeia de caracteres. Use caracteres de espaço em branco para tornar um programa mais legível. Observe que o compilador também trata comentários como espaço em branco.

Veja também

Tokens C

Comentários C

13/05/2021 • 2 minutes to read

Um "comentário" é uma sequência de caracteres que começa com uma combinação de barra/asterisco (**/***) que é tratada como um único caractere de espaço em branco pelo compilador e, caso contrário, é ignorada. Um comentário pode incluir qualquer combinação de caracteres do conjunto de caracteres representáveis, incluindo caracteres de nova linha, mas excluindo o delimitador de "final comment" (***/**). Os comentários podem ocupar mais de uma linha mas não podem ser aninhados.

Os comentários podem aparecer em qualquer lugar em que um caractere de espaço em branco é permitido. Como o compilador trata um comentário como um único caractere de espaço em branco, você não pode incluir comentários nos tokens. O compilador ignora os caracteres no comentário.

Use comentários para documentar seu código. Este exemplo é um comentário aceito pelo compilador:

```
/* Comments can contain keywords such as
   for and while without generating errors. */
```

Os comentários podem aparecer na mesma linha de uma instrução de código:

```
printf( "Hello\n" );  /* Comments can go here */
```

Você pode optar por preceder funções ou módulos do programa com um bloco de comentário descritivo:

```
/* MATHERR.C illustrates writing an error routine
* for math functions.
*/
```

Como os comentários não podem conter comentários aninhados, este exemplo causa um erro:

```
/* Comment out this routine for testing

   /* Open file */
    fh = _open( "myfile.c", _O_RDONLY );
    .
    .
    .
*/
```

O erro ocorre porque o compilador reconhece o primeiro `*/`, após as palavras `Open file`, como o fim do comentário. Ele tenta processar o texto restante e gera um erro ao encontrar `*/` fora de um comentário.

Embora você possa usar comentários para executar determinadas linhas de código inativo para fins de teste, as políticas `#if` e `#endif` do pré-processador e a compilação condicional são uma alternativa útil para essa tarefa. Para obter mais informações, consulte Diretivas do pré-processador na *Referência do pré-processador*.

Específico da Microsoft

O compilador da Microsoft também dá suporte a comentários de linha única precedidos por duas barras (**//**). Se você compilar com /Za (padrão ANSI), esses comentários gerarão erros. Esses comentários não podem se estender para uma segunda linha.

```
// This is a valid comment
```

Os comentários que começam com duas barras (**//**) são encerrados pelo próximo caractere de nova linha que não é precedido por um caractere de escape. No próximo exemplo, o caractere de nova linha é precedido por uma barra invertida (****), criando uma "sequência de escape". Essa sequência de escape faz com que o compilador trate a próxima linha como parte da linha anterior. (Para obter mais informações, consulte Sequências de escape.)

```
// my comment \
    i++;
```

Portanto, a instrução `i++;` é comentada.

O padrão do Microsoft C é que as extensões da Microsoft sejam habilitadas. Use /Za para desabilitar essas extensões.

FINAL específico da Microsoft

Veja também

Tokens C

Avaliação de tokens

13/05/2021 • 2 minutes to read

Quando o compilador interpreta tokens, ele inclui o máximo de caracteres possível em um único token antes seguir para o próximo token. Devido a esse comportamento, o compilador pode não interpretar tokens como você gostaria se eles não forem separados corretamente por um espaço em branco. Considere a seguinte expressão:

```
i+++j
```

Neste exemplo, o compilador primeiro cria o operador mais longo possível (`++`) dos três sinais positivos, depois processa o sinal de positivo restante como um operador de adição (`+`). Assim, a expressão é interpretada como `(i++) + (j)` , não `(i) + (++j)` . Nesse e em casos semelhantes, use espaço em branco e parênteses para evitar ambiguidade e assegurar a avaliação apropriada da expressão.

Específico da Microsoft

O compilador C trata um caractere CTRL+Z como um indicador de fim do arquivo. Ignora qualquer texto após CTRL+Z.

FINAL específico da Microsoft

Veja também

Tokens C

Palavras-chave C

13/05/2021 • 2 minutes to read

Palavras-chave são palavras que têm um significado especial para o compilador do C. Nas fases de conversão 7 e 8, um identificador não pode ter a mesma grafia e caso como uma palavra-chave C. Para obter mais informações, consulte fases de tradução na referência do *pré-processador*. Para obter mais informações sobre identificadores, consulte identificadores.

Palavras-chave C padrão

A linguagem C usa as seguintes palavras-chave:

`auto`
`break`
`case`
`char`
`const`
`continue`
`default`
`do`
`double`
`else`
`enum`

`extern`
`float`
`for`
`goto`
`if`
`inline`[1, um]
`int`
`long`
`register`
`restrict`[1, um]
`return`

`short`
`signed`
`sizeof`
`static`
`struct`
`switch`
`typedef`
`union`
`unsigned`
`void`
`volatile`

`while`

`_Alignas` [2, um]
`_Alignof` [2, um]
`_Atomic` [2, b]
`_Bool` [1, um]
`_Complex` [1, b]
`_Generic` [2, um]
`_Imaginary` [1, b]
`_Noreturn` [2, um]
`_Static_assert` [2, um]
`_Thread_local` [2, b]

[1] palavras-chave introduzidas no C99 ISO.

[2] palavras-chave introduzidas no C11 ISO.

[a] partir do Visual Studio 2019 versão 16,8, essas palavras-chave têm suporte no código compilado como C quando as `/std:c11` `/std:c17` Opções de compilador ou são especificadas.

[b] a partir do Visual Studio 2019 versão 16,8, essas palavras-chave são reconhecidas, mas não têm suporte do compilador no código compilado como C quando as `/std:c11` `/std:c17` Opções do compilador ou são especificadas.

Você não pode redefinir palavras-chave. No entanto, você pode especificar o texto para substituir palavras-chave antes da compilação usando as diretivasde pré-processador C.

Palavras-chave C específicas da Microsoft

Os padrões ANSI e ISO C permitem que identificadores com dois sublinhados à esquerda sejam reservados para implementações de compilador. A Convenção da Microsoft é preceder nomes de palavras-chave específicos da Microsoft com sublinhados duplos. Essas palavras não podem ser usadas como nomes de identificador. Para obter uma descrição das regras para identificadores de nomenclatura, incluindo o uso de sublinhados duplos, consulte identificadores.

As palavras-chave e os identificadores especiais a seguir são reconhecidos pelo compilador Microsoft C:

`__asm` [05]
`__based` [3, 5]
`__cdecl` [05]
`__declspec` [05]
`__except` [05]
`__fastcall`
`__finally` [05]

`__inline` [05]
`__int16` [05]
`__int32` [05]
`__int64` [05]
`__int8` [05]
`__leave` [05]
`__restrict`

`__stdcall` [05]
`__try` [05]
`dllexport` [quatro]
`dllimport` [quatro]

`naked` [quatro]

`static_assert` [152]

`thread` [quatro]

[3] a `__based` palavra-chave tem usos limitados para compilações de destino de 32 bits e 64 bits.

[4] esses são os identificadores especiais quando usados com `__declspec` o; seu uso em outros contextos é irrestrito.

[5] 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.

[6] se você não incluir <Assert. h>, o compilador do Microsoft Visual C `static_assert` será mapeado para a `_Static_assert` palavra-chave C11.

As extensões da Microsoft são ativadas por padrão. Para auxiliar na criação de código portátil, você pode desabilitar extensões da Microsoft especificando a opção /za (Disable Language Extensions) durante a compilação. Quando você usa essa opção, algumas palavras-chave específicas da Microsoft são desabilitadas.

Quando as extensões do Microsoft são habilitadas, você pode usar as palavras-chave listadas acima em seus programas. Para conformidade com os padrões, a maioria dessas palavras-chave é precedida por um duplo sublinhado. As quatro exceções„ `dllexport` `dllimport` `naked` e `thread` , são usadas apenas com `__declspec` e não exigem um sublinhado duplo à esquerda. Para compatibilidade com versões anteriores, há suporte para as versões de sublinhado único do restante das palavras-chave.

Consulte também

Elementos de C

Identificadores C

13/05/2021 • 4 minutes to read

Os "identificadores" ou "símbolos" são os nomes que você fornece para variáveis, tipos, funções e rótulos em seu programa. Os nomes de identificadores devem ser diferentes na ortografia e nas maiúsculas e minúsculas de todas as palavras-chave. Você não pode usar palavras-chave (no C ou Microsoft) como identificadores; elas são reservadas para uso especial. Você cria um identificador especificando-o na declaração de variável, tipo ou função. Neste exemplo, `result` é um identificador para uma variável de inteiro, e `main` e `printf` são nomes de identificadores para funções.

```
#include <stdio.h>

int main()
{
    int result;

    if ( result != 0 )
        printf_s( "Bad file handle\n" );
}
```

Após a declaração, você pode usar o identificador em instruções de programa posteriores para fazer referência ao valor associado.

Um tipo especial de identificador, chamado de rótulo de instrução, pode ser usado em `goto` instruções. (Declarações são descritas em Declarações e tipos Rótulos de instrução são descritos em Instruções goto e rotuladas.)

Sintaxe

identificador:
Não dígito
identifier nondigit
identifier digit

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

digit: uma destas opções
0 1 2 3 4 5 6 7 8 9

O primeiro caractere do nome de identificador deve ser `nondigit` (ou seja, o primeiro caractere deve ser um sublinhado ou letra maiúscula ou minúscula). Permite seis caracteres ANSI significativos em um nome de identificador externo e 31 para nomes (dentro de uma função) de identificadores internos. Os identificadores externos (declarados no escopo global ou declarados com a classe de armazenamento `extern`) podem estar sujeitos a restrições de nomenclatura adicionais, pois esses identificadores precisam ser processados por outros softwares, como vinculadores.

Específico da Microsoft

Ainda que o ANSI permita 6 caracteres significativos em nomes de identificadores externos e 31 para nomes (dentro de uma função) de identificadores internos, o compilador do C da Microsoft permite 247 caracteres em um nome de identificador interno ou externo. Se você não tiver preocupado com compatibilidade de ANSI, pode

alterar esse padrão para um número menor ou maior usando a opção /H (restringe o comprimento dos nomes externos).

FINAL específico da Microsoft

O compilador do C considerará as letras maiúsculas e minúsculas como caracteres distintos. Esse recurso, chamado de "diferenciação de maiúsculas e minúsculas" permite que você crie identificadores distintos que têm a mesma ortografia mas maiúsculas e minúsculas diferentes para uma ou mais letras. Por exemplo, cada um dos seguintes identificadores é exclusivo:

```
add
ADD
Add
aDD
```

Específico da Microsoft

Não selecione nomes dos identificadores que comecem com dois sublinhados ou com um sublinhado seguido por uma letra maiúscula. O padrão ANSI C permite que os nomes dos identificadores que começam com essas combinações de caracteres sejam reservados para uso pelo compilador. Os identificadores com escopo no nível de arquivo não devem ser nomeados com um sublinhado e uma letra minúscula como as duas primeiras letras. Os nomes de identificadores que começam com esses caracteres também são reservados. Por convenção, a Microsoft usa um sublinhado e uma letra maiúscula para iniciar nomes de macro e sublinhados duplos para nomes de palavras-chave específicos da Microsoft. Para evitar conflitos de nomenclatura, sempre selecione nomes do identificador que não comecem com um ou os dois sublinhados, ou os nomes que comecem com um sublinhado seguido por uma letra maiúscula.

FINAL específico da Microsoft

Os exemplos a seguir são identificadores válidos em conformidade com restrições de nome de ANSI ou Microsoft:

```
j
count
temp1
top_of_page
skip12
LastNum
```

Específico da Microsoft

Ainda que os identificadores nos arquivos de origem tenham diferenciação de maiúsculas e minúsculas por padrão, os símbolos nos arquivos de objeto não são. O Microsoft C trata os identificadores em uma unidade de compilação com diferenciação de maiúsculas e minúsculas.

O vinculador da Microsoft faz diferenciação de maiúsculas e minúsculas. Você deve especificar todos os identificadores consistentemente de acordo com maiúsculas e minúsculas.

O "conjunto de caracteres de origem" é o conjunto de caracteres válidos que podem aparecer nos arquivos de origem. Para o Microsoft C, o conjunto de origem é o conjunto de caracteres ASCII padrão. O conjunto de caracteres de origem e o conjunto de caracteres de execução incluem os caracteres ASCII usados como sequências de escape. Consulte Constantes de caractere para obter informações sobre o conjunto de caracteres de execução.

FINAL específico da Microsoft

Um identificador tem “escopo”, que é a região de programa em que é conhecido, e “vinculação”, que determina

se o mesmo nome em outro escopo faz referência ao mesmo identificador. Esses tópicos são explicados em Tempo de vida, escopo, visibilidade e vinculação.

Veja também

Elementos de C

Caracteres multibyte e largos

13/05/2021 • 2 minutes to read

Um caractere multibyte é um caractere composto por sequências de um ou mais bytes. Cada sequência de bytes representa um único caractere no conjunto de caracteres estendido. Os caracteres multibyte são usados nos conjuntos de caracteres como Kanji.

Os caracteres largos são códigos de caracteres multilíngues que sempre têm 16 bits de largura. O tipo de constantes de caractere é `char` ; para caracteres largos, o tipo é `wchar_t` . Como os caracteres largos são sempre de tamanho fixo, usar caracteres largos simplifica a programação com conjuntos de caracteres internacionais.

O literal de cadeia de caracteres largo `L"hello"` se torna uma matriz de seis inteiros do tipo `wchar_t` .

```
{L'h', L'e', L'l', L'l', L'o', 0}
```

A especificação Unicode é a especificação de caracteres largos. As rotinas da biblioteca em tempo de execução para converter entre caracteres multibyte e largos incluem `mbstowcs` , `mbtowc` , `wcstombs` e `wctomb` .

Veja também

Identificadores de C

Trígrafos

13/05/2021 • 2 minutes to read

O conjunto de caracteres de origem dos programas de origem em C está contido dentro do conjunto de caracteres ASCII de 7 bits, mas é um superconjunto do conjunto de código invariável ISO 646-1983. As sequências de trígrafos permitem que os programas em C sejam escritos usando apenas o conjunto de código invariável da ISO (Organização Internacional de Padronização). Trígrafos são sequências de três caracteres (introduzidos por dois pontos de interrogação consecutivos) que o compilador substitui pelos caracteres de pontuação correspondentes. Você pode usar trígrafos em arquivos de origem em C com um conjunto de caracteres que não contenha representações gráficas convenientes para alguns caracteres de pontuação.

O C++17 remove trígrafos da linguagem. As implementações podem continuar a dar suporte a trígrafos como parte do mapeamento definido pela implementação do arquivo de origem físico para o *conjunto de caracteres de origem básico*, embora o padrão incentive as implementações a não fazer isso. Por meio de C++14, os trígrafos tem suporte como em C.

O Visual C++ continua a dar suporte à substituição de trígrafo, mas é desabilitado por padrão. Para obter informações sobre como habilitar a substituição de trigraphl, consulte `/Zc:trigraphs` (substituição de trigrafos).

A tabela a seguir mostra as nove sequências de trígrafos. Em um arquivo de origem, todas as ocorrências dos caracteres de pontuação na primeira coluna são substituídas pelo caractere correspondente na segunda coluna.

Sequências de trígrafos

TRÍGRAFO	CARACTERE DE PONTUAÇÃO
`??=`	`#`
`??(`	`[`
`??/`	`\`
`??)`	`]`
`??'`	`^`
`??<`	`{`
`??!`	`|`
`??>`	`}`
`??-`	`~`

Um trígrafo é sempre tratado como um único caractere de origem. A conversão de trígrafos ocorre na primeira fase de conversão, antes do reconhecimento dos caracteres de escape em literais de cadeias e constantes de caracteres. Somente os nove trígrafos mostrados na tabela acima são reconhecidos. Todas as outras sequências de caracteres são mantidas sem conversão.

A sequência de escape de caractere, `\?` , impede a interpretação inapropriada das sequências de caracteres do tipo trigrafos. (Para obter informações sobre sequências de escape, consulte sequências de escape.) Por

exemplo, se você tentar imprimir a cadeia de caracteres `What??!` com esta `printf` instrução

```
printf( "What??!\n" );
```

a cadeia impressa será `What|` porque `??!` é uma sequência de trígrafo que é substituída pelo caractere `|`. Escreva a instrução da seguinte maneira para imprimir a cadeia de caracteres corretamente:

```
printf( "What?\?!\n" );
```

Nessa instrução `printf`, um caractere de escape de barra invertida na frente do segundo ponto de interrogação impede a interpretação errônea de `??!` como um trígrafo.

Veja também

`/Zc:trigraphs` (Substituição de trigrafos)
Identificadores de C

Constantes C

13/05/2021 • 2 minutes to read

Uma *constante* é um número, um caractere ou uma cadeia de caracteres que pode ser usada como um valor em um programa. Use constantes para representar os valores de ponto flutuante, inteiro, enumeração ou de caractere que não podem ser alterados.

Sintaxe

`constant`:

- `floating-point-constant`
- `integer-constant`
- `enumeration-constant`
- `character-constant`

As constantes são caracterizadas como tendo um valor e um tipo. Ponto flutuante, inteiro e constantes de caracteres são discutidos nas próximas três seções. As constantes de enumeração são descritas em Declarações de enumeração.

Veja também

Elementos de C

Constantes de ponto flutuante C

13/05/2021 • 2 minutes to read

Uma "constante de ponto flutuante" é um número decimal que representa um número real assinado. A representação de um número real assinado inclui uma parte inteira, uma parte fracionária e um expoente. Use constantes de ponto flutuante para representar valores de ponto flutuante que não podem ser alterados.

Sintaxe

`floating-point-constant`:
- `fractional-constant` optar por `exponent-part` aceitar `floating-suffix`
- `digit-sequence` ** `exponent-part` `floating-suffix` aceitar

`fractional-constant`:
- `digit-sequence` aceitar . `digit-sequence`
- `digit-sequence` .

`exponent-part`:
- e `sign` aceitar `digit-sequence`
- E `sign` aceitar `digit-sequence`

`sign`: um dos
- `+` `-`

`digit-sequence`:
- `digit`
- `digit-sequence` `digit`

`floating-suffix`: um dos
- `f` `l` `F` `L`

Você pode omitir os dígitos anteriores ao ponto decimal (a parte inteira do valor) ou posteriores (a parte fracionária), mas não ambos. Você pode deixar o ponto decimal somente se incluir um expoente. Nenhum caractere de espaço em branco pode separar os dígitos ou caracteres da constante.

Os exemplos a seguir ilustram alguns formatos de constantes de ponto flutuante e expressões:

```
15.75
1.575E1   /* = 15.75   */
1575e-2   /* = 15.75   */
-2.5e-3   /* = -0.0025 */
25E-4     /* =  0.0025 */
```

As constantes de ponto flutuante são positivas, a menos que sejam precedidas por um sinal de subtração (`-`). Nesse caso, o sinal de subtração é tratado como um operador aritmético unário de negação. As constantes de ponto flutuante têm tipo `float` , `double` ou `long double` .

Uma constante de ponto flutuante sem um `f` `F` sufixo,, `l` ou `L` tem tipo `double` . Se a letra `f` ou `F` for o sufixo, a constante terá o tipo `float` . Se for sufixado pela letra `l` ou `L` , ele terá o tipo `long double` . Por exemplo:

```
10.0L  /* Has type long double  */
10.0   /* Has type double       */
10.0F  /* Has type float        */
```

O compilador do Microsoft C representa internamente `long double` o mesmo que o tipo `double` . No entanto, os tipos são distintos. Consulte armazenamento de tipos básicos para obter informações sobre o tipo `double` , `float` e `long double` .

Você pode omitir a parte inteira da constante de ponto flutuante, como mostrado nos seguintes exemplos. O número 0,75 pode ser expresso de várias maneiras, incluindo os seguintes exemplos:

```
.0075e2
0.075e1
.075e1
75e-2
```

Confira também

Constantes C

Limites em constantes de ponto flutuante

13/05/2021 • 2 minutes to read

Específico da Microsoft

Os limites dos valores de constantes de ponto flutuante são fornecidos na seguinte tabela. O arquivo de cabeçalho FLOAT.H contém essas informações.

Limites em constantes de ponto flutuante

CONSTANTE	SIGNIFICADO	VALOR
FLT_DIG **DBL_DIG** **LDBL_DIG**	Número de dígitos, *p*, de modo que um número de ponto flutuante com *q* decimais dígitos pode ser arredondado para uma representação de ponto flutuante e voltar sem perda de precisão.	6 15 15
FLT_EPSILON **DBL_EPSILON** **LDBL_EPSILON**	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
FLT_GUARD		0
FLT_MANT_DIG **DBL_MANT_DIG** **LDBL_MANT_DIG**	Número de dígitos na raiz especificada por **FLT_RADIX** em significando de ponto flutuante. A raiz é 2; portanto, esses valores especificam bits.	24 53 53
FLT_MAX **DBL_MAX** **LDBL_MAX**	Número de ponto flutuante representável máximo.	3.402823466e+38F 1,7976931348623158e+308 1,7976931348623158e+308
FLT_MAX_10_EXP **DBL_MAX_10_EXP** **LDBL_MAX_10_EXP**	Inteiro máximo de modo que 10 gerados para esse número sejam um número de ponto flutuante representável.	38 308 308
FLT_MAX_EXP **DBL_MAX_EXP** **LDBL_MAX_EXP**	Inteiro máximo de modo que **FLT_RADIX** gerado para esse número seja um número de ponto flutuante representável.	128 1024 1024
FLT_MIN **DBL_MIN** **LDBL_MIN**	Valor positivo mínimo.	1.175494351e-38F 2,2250738585072014e-308 2,2250738585072014e-308
FLT_MIN_10_EXP **DBL_MIN_10_EXP** **LDBL_MIN_10_EXP**	Inteiro negativo mínimo de modo que 10 gerados para esse número sejam um número de ponto flutuante representável.	-37 -307 -307

CONSTANTE	SIGNIFICADO	VALOR
FLT_MIN_EXP **DBL_MIN_EXP** **LDBL_MIN_EXP**	Inteiro negativo mínimo de modo que **FLT_RADIX** gerado para esse número seja um número de ponto flutuante representável.	-125 -1021 -1021
FLT_NORMALIZE		0
FLT_RADIX **_DBL_RADIX** **_LDBL_RADIX**	Raiz de representação do expoente.	2 2 2
FLT_ROUNDS **_DBL_ROUNDS** **_LDBL_ROUNDS**	Arredondamento do modo para a adição de ponto flutuante.	1 (próximo) 1 (próximo) 1 (próximo)

Observe que as informações da tabela acima podem ser diferentes em implementações futuras.

FINAL específico da Microsoft

Veja também

C Floating-Point constantes

Constantes de inteiro C

13/05/2021 • 2 minutes to read

Uma *constante de inteiro* é um número decimal (base 10), octal (base 8) ou hexadecimal (base 16) que representa um valor integral. Use as constantes de inteiro para representar os valores de inteiro que não podem ser alterados.

Sintaxe

integer-constant:
  decimal-constant integer-suffix$_{opt}$
  octal-constant integer-suffix$_{opt}$
  hexadecimal-constant integer-suffix$_{opt}$

decimal-constant:
  dígito diferente de zero
  decimal-constant digit

octal-constant:
  0
  octal-constant octal-digit

hexadecimal-constant:
  hexadecimal-prefix hexadecimal-digit
  hexadecimal-constant hexadecimal-digit

hexadecimal-prefix: um de
  0x 0x

nonzero-digit: one of
  1 2 3 4 5 6 7 8 9

octal-digit: one of
  0 1 2 3 4 5 6 7

hexadecimal-digit: one of
  0 1 2 3 4 5 6 7 8 9
  a b c d e f
  A B C D E F

integer-suffix:
  unsigned-suffix long-suffix$_{opt}$
  unsigned-suffix long-long-suffix
  unsigned-suffix 64-bit-integer-suffix
  long-suffix unsigned-suffix$_{opt}$
  long-long-suffix unsigned-suffix$_{opt}$
  64-bit-integer-suffix

unsigned-suffix: one of
  u U

long-suffix: one of
  l L

long-long-suffix: um de
ll LL

64-bit-integer-suffix: um de
i64 I64

Os sufixos **i64** e **I64** são específicos da Microsoft.

As constantes de inteiro são positivas, a menos que sejam precedidas por um sinal de subtração (-). O sinal de subtração é interpretado como o operador aritmético unário de negação. (Consulte Operadores aritméticos unários para obter informações sobre este operador.)

Se uma constante inteira começar com **0x** ou **0X**, ela é hexadecimal. Se ela começar com o dígito **0**, é octal. Caso contrário, pressupõe-se que é decimal.

As seguintes constantes de inteiro são equivalentes:

```
28
0x1C   /* = Hexadecimal representation for decimal 28 */
034    /* = Octal representation for decimal 28 */
```

Nenhum caractere de espaço em branco pode separar os dígitos de uma constante de inteira. Esses exemplos mostram algumas constantes decimais, octais e hexadecimais válidas.

```
/* Decimal Constants */
int                dec_int    = 28;
unsigned           dec_uint   = 4000000024u;
long               dec_long   = 2000000022l;
unsigned long      dec_ulong  = 4000000000ul;
long long          dec_llong  = 9000000000LL;
unsigned long long dec_ullong = 900000000001ull;
__int64            dec_i64    = 9000000000002I64;
unsigned __int64   dec_ui64   = 90000000000004ui64;

/* Octal Constants */
int                oct_int    = 024;
unsigned           oct_uint   = 04000000024u;
long               oct_long   = 02000000022l;
unsigned long      oct_ulong  = 04000000000UL;
long long          oct_llong  = 04400000000000000ll;
unsigned long long oct_ullong = 044400000000000000001Ull;
__int64            oct_i64    = 0444400000000000000002i64;
unsigned __int64   oct_ui64   = 0444400000000000000004uI64;

/* Hexadecimal Constants */
int                hex_int    = 0x2a;
unsigned           hex_uint   = 0XA0000024u;
long               hex_long   = 0x20000022l;
unsigned long      hex_ulong  = 0XA0000021uL;
long long          hex_llong  = 0x8a00000000000000ll;
unsigned long long hex_ullong = 0x8A40000000000010uLL;
__int64            hex_i64    = 0x4a44000000000020I64;
unsigned __int64   hex_ui64   = 0x8a44000000000040Ui64;
```

Veja também

Constantes C

Tipos de inteiro

13/05/2021 • 2 minutes to read

Cada constante de inteiro recebe um tipo com base em seu valor e a forma como é expressa. Você pode forçar qualquer constante de inteiro para tipo `long` acrescentando a letra `l` ou `L` ao final da constante; você pode forçá-la a ser tipo `unsigned` acrescentando `u` ou `U` ao valor. A letra minúscula `l` pode ser confundida com o dígito 1 e deve ser evitada. Seguem algumas formas de `long` constantes de inteiro:

```
/* Long decimal constants */
10L
79L

/* Long octal constants */
012L
0115L

/* Long hexadecimal constants */
0xaL or 0xAL
0X4fL or 0x4FL

/* Unsigned long decimal constant */
776745UL
778866LU
```

O tipo que você atribui a uma constante depende do valor que a constante representa. O valor de uma constante deve estar no intervalo de valores representáveis para o seu tipo. O tipo de uma constante determina quais conversões são executadas quando a constante é usada em uma expressão ou quando o sinal de subtração (`-`) é aplicado. Esta lista resume as regras de conversão para constantes de número inteiro.

- O tipo de uma constante decimal sem um sufixo é `int` , `long int` ou `unsigned long int` . O primeiro destes três tipos em que o valor da constante pode ser representado é o tipo atribuído à constante.
- O tipo atribuído às constantes octais e hexadecimal sem sufixos é `int` , `unsigned int` , `long int` ou `unsigned long int` dependendo do tamanho da constante.
- O tipo atribuído a constantes com um `u` `U` sufixo or é `unsigned int` ou `unsigned long int` dependendo do seu tamanho.
- O tipo atribuído a constantes com um `l` `L` sufixo or é `long int` ou `unsigned long int` dependendo do seu tamanho.
- O tipo atribuído a constantes com um `u` or `U` e um `l` `L` sufixo or é `unsigned long int` .

Veja também

Constantes de inteiro C

Limites de inteiro C e C++

13/05/2021 • 2 minutes to read

Específico da Microsoft

Os limites para tipos de inteiros em C e C++ são listados na tabela a seguir. Esses limites são definidos no arquivo de cabeçalho padrão C `<limits.h>`. O cabeçalho da biblioteca padrão C++ `<limits>` inclui o `<climits>`, que inclui o `<limits.h>`.

O Microsoft C também permite a declaração de variáveis inteiras de tamanho, que são tipos integrais de tamanho de 8, 16-, 32-ou 64 bits. Para obter mais informações sobre os inteiros de tamanho em C, consulte tipos de inteiro de tamanho.

Limites em constantes de inteiro

CONSTANTE	SIGNIFICADO	VALOR
CHAR_BIT	Número de bits na menor variável que não é um campo de bit.	8
SCHAR_MIN	Valor mínimo para uma variável do tipo `signed char`.	-128
SCHAR_MAX	Valor máximo para uma variável do tipo `signed char`.	127
UCHAR_MAX	Valor máximo para uma variável do tipo `unsigned char`.	255 (0xff)
CHAR_MIN	Valor mínimo para uma variável do tipo `char`.	-128; 0 se a opção /J for usada
CHAR_MAX	Valor máximo para uma variável do tipo `char`.	127; 255 se a opção /J for usada
MB_LEN_MAX	Número máximo de bytes em uma constante de vários caracteres.	5
SHRT_MIN	Valor mínimo para uma variável do tipo `short`.	-32768
SHRT_MAX	Valor máximo para uma variável do tipo `short`.	32767
USHRT_MAX	Valor máximo para uma variável do tipo `unsigned short`.	65535 (0xffff)
INT_MIN	Valor mínimo para uma variável do tipo `int`.	-2147483647 - 1

CONSTANTE	SIGNIFICADO	VALOR
INT_MAX	Valor máximo para uma variável do tipo `int` .	2147483647
UINT_MAX	Valor máximo para uma variável do tipo `unsigned int` .	4294967295 (0xffffffff)
LONG_MIN	Valor mínimo para uma variável do tipo `long` .	-2147483647 - 1
LONG_MAX	Valor máximo para uma variável do tipo `long` .	2147483647
ULONG_MAX	Valor máximo para uma variável do tipo `unsigned long` .	4294967295 (0xffffffff)
LLONG_MIN	Valor mínimo para uma variável do tipo `long long` .	-9.223.372.036.854.775.807-1
LLONG_MAX	Valor máximo para uma variável do tipo `long long` .	9,223,372,036,854,775,807
ULLONG_MAX	Valor máximo para uma variável do tipo `unsigned long long` .	18446744073709551615 (0xFFFFFFFFFFFFFFFF)

Se um valor exceder a representação do maior inteiro, o compilador da Microsoft gera um erro.

FINAL específico da Microsoft

Veja também

Constantes de inteiro C

Constantes de caractere C

13/05/2021 • 2 minutes to read

Uma "constante de caractere" é formada colocando um único caractere do conjunto de caracteres representável entre aspas simples (' '). As constantes de caractere são usadas para representar caracteres no conjunto de caracteres de execução.

Sintaxe

caractere-constante: **'** *c-Char-Sequence* **'**

L ' *c-Char-Sequence* **'**

c-Char-Sequence: *c-Char*

c-char-sequence c-char

c-char: Qualquer membro do conjunto de caracteres de origem, exceto aspas únicas ('), barra invertida (****) ou caractere de nova linha

sequência de escape

escape-sequence: *simple-escape-sequence*

octal-escape-sequence

sequência de escape hexadecimal

simple-escape-sequence: um dentre **\a \b \f \n \r \t \v**

\' \" \\ \?

octal-escape-sequência: **** *octal-dígito*

**** *dígito octal octal-dígito*

**** *dígito octal-dígito octal-dígito*

hexadecimal-escape-sequence: **\x** *hexadecimal-digit*

hexadecimal-escape-sequence hexadecimal-digit

Veja também

Constantes C

Tipos de caractere

13/05/2021 • 2 minutes to read

Uma constante de caractere inteiro não precedida pela letra **L** tem o tipo `int` . O valor de uma constante de caractere inteiro que contém um único caractere é o valor numérico do caractere interpretado como um inteiro. Por exemplo, o valor numérico do caractere `a` é 97 em decimal e 61 em hexadecimal.

Sintaticamente, uma "constante de caractere largo" é uma constante de caractere prefixada pela letra **L**. Uma constante de caractere largo tem tipo `wchar_t` , um tipo inteiro definido no stddef. Arquivo de cabeçalho H. Por exemplo:

```
char    schar =  'x';   /* A character constant          */
wchar_t wchar = L'x';   /* A wide-character constant for
                           the same character           */
```

As constantes de caractere largo têm 16 bits de largura e especificam membros do conjunto estendido de caracteres de execução. Eles permitem que você expresse caracteres em alfabetos que são muito grandes para serem representados por tipo `char` . Consulte Multibyte e caracteres largos para obter mais informações sobre os caracteres largos.

Veja também

Constantes de caractere C

Conjunto de caracteres de execução

13/05/2021 • 2 minutes to read

Este conteúdo refere-se ao "conjunto de caracteres de execução". O conjunto de caracteres de execução não é necessariamente o mesmo conjunto de caracteres de origem usado para escrever programas C. O conjunto de caracteres de execução inclui todos os caracteres no conjunto de caracteres de origem bem como o caractere nulo, o caractere de nova linha, backspace, tabulação horizontal, tabulação vertical, retorno de carro e sequências de escape. Os conjuntos de caracteres de origem e de execução podem diferir em outras implementações.

Veja também

Constantes de caractere C

Sequências de escape

13/05/2021 • 2 minutes to read

As combinações de caracteres que consistem em uma barra invertida (****) seguida por uma letra ou por uma combinação de dígitos são chamadas de "sequências de escape". Para representar um caractere de nova linha, aspas simples ou alguns outros caracteres em uma constante de caractere, você deve usar sequências de escape. Uma sequência de escape é considerada um único caractere e, portanto, é válida como uma constante de caractere.

Geralmente, as sequências de escape são usadas para especificar ações como retornos de carro e movimentos de tabulação em terminais e impressoras. Também são usadas para fornecer representações literais de caracteres não imprimíveis e de caracteres que normalmente têm significados especiais, como as aspas duplas ("). A tabela a seguir lista as sequências de escape ANSI e o que elas representam.

Observe que o ponto de interrogação precedido por uma barra invertida (**\ ?**) especifica um ponto de interrogação literal em casos em que a sequência de caracteres seria interpretada erroneamente como um trigraph. Consulte Trígrafos para obter mais informações.

Sequências de escape

SEQUÊNCIA DE ESCAPE	REPRESENTA
\a	Campainha (alerta)
\b	Backspace
\f	Avanço de formulário
\n	Nova linha
\r	Retorno de carro
****	Guia horizontal
\v	Guia vertical
\'	Aspas simples
\"	Aspas duplas
****	Barra invertida
\?	Ponto de interrogação literal
****OOO**	Caractere ASCII em notação octal
\x *hh*	Caractere ASCII em notação hexadecimal

SEQUÊNCIA DE ESCAPE	REPRESENTA
\X *HHHH*	Caractere Unicode em notação hexadecimal, se esta sequência de escape é usada em uma constante de caractere largo ou uma literal de cadeia de caracteres Unicode. Por exemplo, `WCHAR f = L'\x4e00'` ou `WCHAR b[] = L"The Chinese character for one is \x4e00"` .

Específico da Microsoft

Se uma barra invertida precede um caractere que não consta da tabela, o compilador trata o caractere indefinido como o próprio caractere. Por exemplo, `\c` é tratado como um `c`.

FINAL específico da Microsoft

As sequências de escape permitem que você envie caracteres de controle não gráficos a um dispositivo de exibição. Por exemplo, o caractere de ESC (**\033**) costuma ser usado como o primeiro caractere de um comando de controle para um terminal ou uma impressora. Algumas sequências de escape são específicas ao dispositivo. Por exemplo, as sequências de escape de tabulação vertical e de avanço de página (**\v** e **\f**) não afetam a saída da tela, mas executam operações de impressora apropriadas.

Você também pode usar a barra invertida (****) como um caractere de continuação. Quando um caractere de nova linha (equivalente a pressionar a tecla RETURN) vem logo após a barra invertida, o compilador ignora a barra invertida e o caractere de nova linha e trata a próxima linha como parte da linha anterior. Isso é útil principalmente para definições de pré-processador que ocupam mais do que uma única linha. Por exemplo:

```
#define assert(exp) \
( (exp) ? (void) 0:_assert( #exp, __FILE__, __LINE__ ) )
```

Confira também

Constantes de caractere C

Especificações de caractere octa e hexadecimal

13/05/2021 • 2 minutes to read

A sequência **** *OOO* significa que você pode especificar qualquer caractere no conjunto de caracteres ASCII como um código de caractere octal de três dígitos. O valor numérico inteiro octal especifica o valor de caractere desejado ou do caractere largo.

De maneira semelhante, a sequência **\x***hhh* permite especificar qualquer caractere ASCII como um código de caracteres hexadecimal. Por exemplo, o caractere de backspace ASCII pode ser atribuído como a sequência de escape C normal (**\b**) ou codificado como **\010** (octal) ou **\x008** (hexadecimal).

Você pode usar somente os dígitos de 0 a 7 em uma sequência de escape octal. As sequências de escape octais nunca poderão ser maiores do que três dígitos e são encerradas pelo primeiro caractere que não seja um dígito octal. Embora não seja necessário usar todos os três dígitos, você deve usar pelo menos um. Por exemplo, a representação octal é **\10** para o caractere de backspace ASCII e **\101** para a letra A, conforme indicado em um gráfico ASCII.

De maneira semelhante, você deve usar pelo menos um dígito para uma sequência de escape hexadecimal, mas você pode omitir o segundo e o terceiro dígitos. Portanto, é possível especificar a sequência de escape hexadecimal para o caractere de backspace como **\x8**, **\x08** ou **\x008**.

O valor da sequência de escape octal ou hexadecimal deve estar no intervalo de valores representáveis para `unsigned char` o tipo para uma constante de caractere e `wchar_t` um tipo para uma constante de caractere largo. Consulte Caracteres multibytes e largos para obter informações sobre constantes de caractere largo.

Ao contrário das constantes de escape octal, o número de dígitos hexadecimais em uma sequência de escape é ilimitado. Uma sequência de escape hexadecimal é encerrada no primeiro caractere que não seja um dígito hexadecimal. Como os dígitos hexadecimais incluem as letras **a** a **f**, é preciso tomar cuidado para garantir que a sequência de escape seja encerrada no dígito pretendido. Para evitar confusão, você pode colocar definições octais ou hexadecimais de caracteres em uma definição de macro:

```
#define Bell '\x07'
```

Para valores hexadecimais, você pode dividir a cadeia de caracteres para mostrar claramente o valor correto:

```
"\xabc"    /* one character  */
"\xab" "c" /* two characters */
```

Veja também

Constantes de caractere C

literais String C

13/05/2021 • 2 minutes to read

Um "literal de cadeia de caracteres" é uma sequência de caracteres do conjunto de caracteres de origem entre aspas duplas (**"** **"**). Os literais de cadeia de caracteres são usados para representar uma sequência de caracteres que, juntos, formam uma cadeia de caracteres terminada em nulo. Você sempre deve prefixar literais de cadeia de caracteres largos com a letra **L**.

Sintaxe

cadeia de caracteres literal:
" *s-Char-opt-Sequence* **"**
L" $s\text{-}char\text{-}sequence_{opt}$ **"**

s-char-sequence:
s-Char

s-char-sequence s-char

s-Char:
qualquer membro do conjunto de caracteres de origem, exceto a marca de aspas duplas ("), a barra invertida (\) ou o caractere de nova linha

sequência de escape

Comentários

O exemplo abaixo é um literal de cadeia de caracteres simples:

```
char *amessage = "This is a string literal.";
```

Todos os códigos de escape listados na tabela Sequências de escape são válidos em literais de cadeia de caracteres. Para representar uma aspa dupla em um literal de cadeia de caracteres, use a sequência de escape **\"**. As aspas simples (') podem ser representadas sem uma sequência de escape. A barra invertida (****) deve ser seguida com uma segunda barra invertida (****) quando ela aparece dentro de uma cadeia de caracteres. Quando uma barra invertida aparece no fim de uma linha, é sempre interpretada como um caractere de continuação de linha.

Veja também

Elementos de C

Tipo para literais da cadeia de caracteres

13/05/2021 • 2 minutes to read

Literais de cadeia de caracteres têm matriz de tipo de `char` (ou seja, **Char []**). (Cadeias de caracteres largos têm matriz de tipos de `wchar_t` (ou seja, **wchar_t []**).) Isso significa que uma cadeia de caracteres é uma matriz com elementos do tipo `char` . O número de elementos na matriz é igual ao número de caracteres na cadeia de caracteres mais um, para o caractere nulo de encerramento.

Veja também

Literais de cadeia de caracteres C

Armazenamento de literais da cadeia de caracteres

13/05/2021 • 2 minutes to read

Os caracteres de uma cadeia de caracteres literal são armazenados em locais de memória contíguos. Uma sequência de escape (como **** ou **\ "**) dentro de um literal de cadeia de caracteres conta como um único caractere. Um caractere nulo (representado pela sequência de escape **\0**) é acrescentado automaticamente às literais de cadeia de caracteres e marca o fim de cada uma delas. (Isso ocorre durante a fase 7 da tradução .) Observe que o compilador pode não armazenar duas cadeias de caracteres idênticas em dois endereços diferentes. /GF força o compilador a colocar uma única cópia de cadeias de caracteres idênticas no arquivo executável.

Comentários

Específico da Microsoft

As cadeias de caracteres têm duração de armazenamento estático. Consulte Classes de armazenamento para obter informações sobre a duração de armazenamento.

FINAL específico da Microsoft

Veja também

Literais de cadeia de caracteres C

Concatenação literal da cadeia de caracteres

13/05/2021 • 2 minutes to read

Para formar literais de cadeia de caracteres que ocupam mais de uma linha, você pode concatenar duas cadeias de caracteres. Para fazer isso, digite uma barra invertida e pressione a tecla RETURN. A barra invertida faz com que o compilador ignore o próximo caractere de nova linha. Por exemplo, o literal de cadeia de caracteres

```
"Long strings can be bro\
ken into two or more pieces."
```

é idêntico à cadeia de caracteres

```
"Long strings can be broken into two or more pieces."
```

A concatenação de cadeias de caracteres pode ser usada em qualquer lugar em que você usou anteriormente uma barra invertida seguida por um caractere de nova linha para inserir cadeias de caracteres mais longas que uma linha.

Para forçar uma nova linha dentro de uma literal de cadeia de caracteres, digite a sequência de escape de nova linha (**\n**) no ponto na cadeia de caracteres em que deseja que a linha seja interrompida, como segue:

```
"Enter a number between 1 and 100\nOr press Return"
```

Como as cadeias de caracteres podem começar em qualquer coluna do código-fonte e as cadeias de caracteres longas podem ser retomadas em qualquer coluna de uma linha sucessora, você pode posicionar as cadeias de caracteres para aumentar a legibilidade do código-fonte. Em ambos os casos, sua representação na tela resultante não é afetada. Por exemplo:

```
printf_s ( "This is the first half of the string, "
           "this is the second half ") ;
```

Contanto que cada parte da cadeia de caracteres esteja entre aspas duplas, as partes são concatenadas e reproduzidas como uma única cadeia de caracteres. Essa concatenação ocorre de acordo com a sequência de eventos durante a compilação especificada pelas fases de conversão.

```
"This is the first half of the string, this is the second half"
```

Um ponteiro de cadeia de caracteres, inicializado como duas cadeias literais distintas separada apenas por espaço em branco, é armazenado como uma única cadeia de caracteres (os ponteiros são descritos em Declarações de ponteiro). Quando referenciado corretamente, como no exemplo a seguir, o resultado é idêntico ao do exemplo anterior:

```
char *string = "This is the first half of the string, "
               "this is the second half";

printf_s( "%s" , string ) ;
```

Na fase 6 de conversão, as sequências de caracteres multibyte especificadas por qualquer sequência de literais

de cadeia de caracteres adjacentes ou literais de cadeia de caracteres largos adjacentes são concatenadas em uma única sequência de caracteres multibytes. Consequentemente, não crie programas para permitir a modificação de cadeias de caracteres literais durante a execução. O padrão ANSI C especifica que o resultado da modificação de uma cadeia de caracteres é indefinido.

Veja também

Literais de cadeia de caracteres C

Comprimento máximo da cadeia de caracteres

13/05/2021 • 2 minutes to read

Específico da Microsoft

A compatibilidade ANSI requer que um compilador aceite até 509 caracteres em uma cadeia de caracteres literal depois da concatenação. O comprimento máximo de uma literal de cadeia de caracteres permitida no Microsoft C é de aproximadamente 2.048 bytes. No entanto, se o literal da cadeia de caracteres consiste em partes incluídas entre aspas duplas, o pré-processador concatena as partes em uma única cadeia de caracteres e, para cada linha concatenada, ele acrescenta um byte adicional ao número total de bytes.

Por exemplo, imagine que uma cadeia de caracteres consiste em 40 linhas com 50 caracteres por linha (2.000 caracteres) e uma linha com 7 caracteres e que cada linha é cercada por aspas duplas. Isso adiciona até 2.007 bytes mais um byte para o caractere nulo de encerramento para um total de 2.008 bytes. Na concatenação, um caractere adicional é acrescentado para cada uma das primeiras 40 linhas. Isso totaliza 2.048 bytes. No entanto, se continuações das linhas (\) forem usadas em vez de aspas duplas, o pré-processador não acrescentará um caractere adicional para cada linha.

Quando uma cadeia de caracteres individual entre aspas não pode ter mais que 2048 bytes, um literal de cadeia de caracteres de 65535 bytes pode ser criado por meio da concatenação de cadeias de caracteres.

FINAL específico da Microsoft

Veja também

Literais de cadeia de caracteres C

Pontuação e caracteres especiais

13/05/2021 • 2 minutes to read

A pontuação e os caracteres especiais no conjunto de caracteres C têm vários usos, da organização do texto do programa para definir as tarefas que o compilador ou o programa compilado executa. Eles não especificam uma operação a ser executada. Alguns símbolos de pontuação também são operadores (consulte Operadores). O compilador determina o uso do contexto.

Sintaxe

`punctuator`: um de **()** **[]** **{}** * ,: =; ... #

Esses caracteres têm significados especiais em C. Seus usos são descritos neste livro. O sinal de libra (**#**) só pode ocorrer em diretivas de pré-processamento.

Veja também

Elementos de C

Estrutura do programa

13/05/2021 • 2 minutes to read

Esta seção fornece uma visão geral sobre programas em C e execução de programas. Os termos e os recursos importantes para a compreensão de programas e componentes em C também são apresentados. Os tópicos abordados incluem:

- Arquivos e programas de origem
- A função principal e a execução de programas
- Analisando argumentos de linha de comando
- Tempo de vida, escopo, visibilidade e vinculação
- Namespaces

Como esta seção é uma visão geral, os tópicos abordados contém apenas material introdutório. Consulte as informações indicadas nas referências para obter explicações detalhadas.

Veja também

Referência da linguagem C

Arquivos e programas de origem

13/05/2021 • 2 minutes to read

Um programa de origem pode ser dividido em um ou mais "arquivos de origem " ou "unidades de conversão". A entrada no compilador é chamada de "unidade de conversão".

Sintaxe

translation-unit:
  declaração externa
  translation-unit external-declaration

external-declaration:
  função-definição
  mesma

Visão geral das declarações fornece a sintaxe da `declaration` não terminal e *Referência do pré-processador* explica como a unidade de translação é processada.

> **NOTE**
> Consulte a introdução do Resumo da sintaxe da linguagem C para obter uma explicação das convenções de sintaxe ANSI.

Os componentes de uma unidade de conversão são declarações externas que incluem definições de função e declarações de identificador. Essas declarações e definições podem estar nos arquivos de origem, arquivos de cabeçalho, bibliotecas e outros arquivos de que o programa precisa. Você deve compilar cada unidade de conversão e vincular os arquivos de objeto resultantes para criar um programa.

Um "programa de origem" C é uma coleção de políticas, pragmas, declarações, definições, blocos de instruções e funções. Para que sejam componentes válidos de um programa do Microsoft C, cada um deve ter a sintaxe descrita neste manual, embora possam aparecer em qualquer ordem no programa (sujeito às regras definidas ao longo deste manual). No entanto, o local desses componentes em um programa afeta como variáveis e funções podem ser usadas em um programa. (Consulte Tempo de vida, escopo, visibilidade e vinculação para obter mais informações.)

Os arquivos de origem não precisam conter instruções executáveis. Por exemplo, você pode considerar útil colocar definições de variáveis em um arquivo de origem e depois declarar referências a essas variáveis em outros arquivos de origem que as usam. Essa técnica torna as definições fáceis de localizar e atualizar quando necessário. Pela mesma razão, constantes e macros são geralmente organizadas em arquivos separados chamados "arquivos de inclusão" ou "arquivos de cabeçalho" que podem ser referenciados nos arquivos de origem conforme necessário. Consulte a *Referência do pré-processador* para obter informações sobre macros e arquivos de inclusão.

Veja também

Estrutura do programa

Diretivas para o pré-processador

13/05/2021 • 2 minutes to read

Uma "política" instrui o pré-processador C a realizar uma ação específica no texto do programa antes da compilação. As Diretivas para o pré-processador são totalmente descritas na *Referência de pré-processador*. Este exemplo usa a política de pré-processador `#define`:

```
#define MAX 100
```

Esta instrução determina que o compilador substitua cada ocorrência de `MAX` por `100` antes da compilação. As políticas de pré-processador do compilador C são:

#DEFINE	#ENDIF	#IFDEF	#LINE
`#elif`	`#error`	**#ifndef**	**#pragma**
`#else`	`#if`	`#include`	`#undef`

Veja também

Arquivos de origem e programas de origem

C Pragmas

13/05/2021 • 2 minutes to read

Específico da Microsoft

Um *pragma* instrui o compilador a executar uma ação específica no momento da compilação. Os pragmas variam de compilador para compilador. Por exemplo, você pode usar o `optimize` pragma para definir as otimizações a serem executadas em seu programa. Os pragmas do Microsoft C são:

`alloc_text`
`auto_inline`
`bss_seg`
`check_stack`
`code_seg`
`comment`
`component`
`const_seg`
`data_seg`

`deprecated`
`detect_mismatch`
`fenv_access`
`float_control`
`fp_contract`
`function`
`hdrstop`
`include_alias`
`inline_depth`

`inline_recursion`
`intrinsic`
`make_public`
`managed`
`message`
`omp`
`once`
`optimize`
`pack`

`pop_macro`
`push_macro`
`region`, `endregion`
`runtime_checks`
`section`
`setlocale`
`strict_gs_check`
`unmanaged`
`warning`

Consulte diretivas de pragma e a `__Pragma` palavra-chave para obter uma descrição dos pragmas do compilador C da Microsoft.

FINAL específico da Microsoft

Veja também

Arquivos de origem e programas de origem

Declarações e definições C

13/05/2021 • 2 minutes to read

Uma "declaração" estabelece uma associação entre uma variável, uma função ou um tipo específico e os respectivos atributos. A visão geral das declarações apresenta a sintaxe ANSI para o não terminal `declaration`. Uma declaração também especifica onde e quando um identificador pode ser acessado (a "vinculação" de um identificador). Consulte Tempo de vida, escopo, visibilidade e vinculação para obter informações sobre vinculação.

Uma "definição" de uma variável estabelece as mesmas associações que uma declaração, mas também faz com que ocorra a alocação de armazenamento para a variável.

Por exemplo, as funções `main`, `find` e `count` e as variáveis `var` e `val` são definidas em um arquivo de origem, nesta ordem:

```
int main() {}

int var = 0;
double val[MAXVAL];
char find( fileptr ) {}
int count( double f ) {}
```

As variáveis `var` e `val` podem ser usadas nas funções `find` e `count`; nenhuma declaração adicional é necessária. Porém, esses nomes não são visíveis (não podem ser acessados) em `main`.

Veja também

Arquivos de origem e programas de origem

Declarações de função e definições

13/05/2021 • 2 minutes to read

Os protótipos de função estabelecem o nome da função, seu tipo de retorno, o tipo e o número de seus parâmetros formais. Uma definição de função inclui o corpo da função.

Comentários

As declarações de função e de variável podem aparecer dentro ou fora de uma definição de função. Qualquer declaração dentro de uma definição de função é dita aparecer no nível "interno" ou "local". Uma declaração fora de todas as definições de função é dita aparecer como "externa", "global", ou no nível do "escopo de arquivo". As definições de variáveis, como declarações, podem aparecer no nível interno (dentro de uma definição de função) ou no nível externo (fora de todas as definições de função). As definições de função sempre ocorrem no nível externo. As definições de função são abordadas em mais detalhes em Definições de função. Os protótipos de função são abordados em Protótipos de função.

Veja também

Arquivos de origem e programas de origem

Blocos

13/05/2021 • 2 minutes to read

Uma sequência de declarações, definições e instruções entre chaves ({ }) é denominada "bloco". Existem dois tipos de blocos em C. A "instrução composta", uma instrução composta por uma ou mais instruções (consulte A instrução composta), é um tipo de bloco. O outro, a "definição de função", consiste em uma instrução composta (o corpo da função) com o "cabeçalho" associado à função (o nome da função, o tipo de retorno e os parâmetros, formais). Um bloco dentro de outros blocos é chamado de "aninhado".

Observe que, ainda que todas as instruções compostas estejam entre chaves, nem tudo que está entre chaves constitui uma instrução composta. Por exemplo, embora as especificações de matriz, estrutura ou dos elementos de enumeração possam aparecer entre chaves, elas não são instruções compostas.

Veja também

Arquivos de origem e programas de origem

Programa de exemplo

13/05/2021 • 2 minutes to read

O seguinte programa de origem de C consiste de dois arquivos de origem. Ele fornece uma visão geral das várias declarações e definições possíveis em um programa C. As seções posteriores neste livro descrevem como escrever essas declarações, definições e inicializações e como usar palavras-chave C, como `static` e `extern` . A função `printf` é declarada no arquivo de cabeçalho STDIO.H do C.

As funções `main` e `max` são assumidas como em arquivos separados e a execução do programa começa com a função `main` . Nenhuma função explícita do usuário é executada antes de `main` .

```
/*************************************************************
                    FILE1.C - main function
*************************************************************/

#define ONE     1
#define TWO     2
#define THREE   3
#include <stdio.h>

int a = 1;                      // Defining declarations
int b = 2;                      //  of external variables

extern int max( int a, int b );  // Function prototype

int main()                      // Function definition
{                               //  for main function
    int c;                      // Definitions for
    int d;                      //  two uninitialized
                                //  local variables

    extern int u;               // Referencing declaration
                                //  of external variable
                                //  defined elsewhere
    static int v;               // Definition of variable
                                //  with continuous lifetime

    int w = ONE, x = TWO, y = THREE;
    int z = 0;

    z = max( x, y );            // Executable statements
    w = max( z, w );
    printf_s( "%d %d\n", z, w );
    return 0;
}

/*************************************************************
            FILE2.C - definition of max function
*************************************************************/

int max( int a, int b )         // Note formal parameters are
                                //  included in function header
{
    if( a > b )
        return( a );
    else
        return( b );
}
```

FILE1.C contém o protótipo da função `max`. Esse tipo de declaração, às vezes, é chamado de "declaração de encaminhamento" porque a função é declarada antes de ser usada. A definição para a função `main` inclui chamadas a `max`.

As linhas iniciadas com `#define` são políticas do pré-processador. Essas políticas mandam o pré-processador substituir os identificadores `ONE`, `TWO` e `THREE` pelos números `1`, `2` e `3`, respectivamente, por FILE1.C. No entanto, as políticas não se aplicam a FILE2.C, que é compilado separadamente e vinculado com FILE1.C. As linhas iniciadas com `#include` informam o compilador para incluir o arquivo STDIO.H, que contém o protótipo da função `printf`. As Políticas de pré-processador são explicadas na *Referência de pré-processador*.

FILE1.C usa a declarações de definição para inicializar as variáveis globais `a` e `b`. As variáveis locais `c` e `d` são declaradas mas não inicializadas. O armazenamento é atribuído a todas esses variáveis. As variáveis estáticas e externas, `u` e `v`, são inicializados automaticamente como 0. Portanto, apenas `a`, `b`, `u`, e `v` contêm valores significativos quando declarados pois são inicializados, explicitamente ou implicitamente. FILE2.C contém a definição de função para `max`. Esta definição satisfaz as chamadas a `max` em. FILE1.C.

O tempo de vida e a visibilidade dos identificadores são discutidos em Tempo de vida, escopo, visibilidade e vinculação. Para obter mais informações sobre funções, consulte Funções.

Veja também

Arquivos de origem e programas de origem

Função main e execução do programa

13/05/2021 • 2 minutes to read

Cada programa C tem uma função principal (main) que deve ser nomeada **main**. Se seu código obedece ao modelo de programação Unicode, você pode usar a versão de **main** para caracteres largos, **wmain**. A função **main** serve como o ponto de partida para a execução do programa. Em geral, ela controla a execução direcionando as chamadas para outras funções no programa. Normalmente, um programa para de ser executado no final de **main**, embora possa terminar em outros pontos no programa por diversos motivos. Às vezes, talvez quando um determinado erro é detectado, pode ser conveniente forçar o término de um programa. Para fazer isso, use a função **exit**. Consulte a *Referência da biblioteca em tempo de execução* para obter informações sobre como usar a função exit, juntamente com um exemplo.

Sintaxe

```
main( int argc, char *argv[ ], char *envp[ ] )
```

Comentários

As funções do programa de origem executam uma ou mais tarefas específicas. A função **main** pode chamar essas funções para executar as respectivas tarefas. Quando **main** chama outra função, passa para ela o controle da execução; assim, a execução começa na primeira instrução da função. Uma função retorna o controle para **Main** quando uma `return` instrução é executada ou quando o final da função é atingido.

Você pode declarar qualquer função, inclusive **main**, para ter parâmetros. O termo "parâmetro" ou "parâmetro formal" refere-se ao identificador que recebe um valor passado para uma função. Consulte Parâmetros para obter informações sobre como passar argumentos para parâmetros. Quando uma função chama outra, a função chamada recebe valores para os respectivos parâmetros da função que chamou. Esses valores são chamados de "argumentos". É possível declarar parâmetros formais para **main** de modo que a função possa receber argumentos da linha de comando usando este formato:

Quando você quer passar informações para a função **main**, tradicionalmente os parâmetros são nomeados `argc` e `argv`, embora o compilador de C não exija esses nomes. Os tipos para `argc` e `argv` são definidos pela linguagem C. Tradicionalmente, se um terceiro parâmetro é passado para **main**, ele é nomeado `envp`. Exemplos posteriormente nesta seção mostram como usar esses três parâmetros para acessar argumentos de linha de comando. As seções a seguir explicam esses parâmetros.

Consulte Usando wmain para obter uma descrição da versão de **main** para caracteres largos.

Veja também

argumentos da função principal e da linha de comando (C++)
Analisando argumentos do C Command-Line

Usando wmain

13/05/2021 • 2 minutes to read

Específico da Microsoft

No modelo de programação Unicode, é possível definir uma versão de caractere largo da função **main**. Use **wmain** em vez de **main** se você quiser escrever um código portátil que obedeça ao modelo de programação Unicode.

Sintaxe

```
wmain( int argc, wchar_t *argv[ ], wchar_t *envp[ ] )
```

Comentários

Parâmetros formais são declarados para **wmain** usando um formato semelhante ao de **main**. Em seguida, pode passar argumentos de caractere largo e, opcionalmente, um ponteiro de ambiente de caractere largo para o programa. Os parâmetros `argv` e `envp` de **wmain** são do tipo `wchar_t*`. Por exemplo:

Se seu programa usa uma função **main**, o ambiente de caractere multibyte é criado pela biblioteca em tempo de execução na inicialização do programa. Uma cópia de caractere amplo do ambiente é criada apenas quando necessário (por exemplo, por uma chamada para as funções `_wgetenv` ou `_wputenv`). Na primeira chamada para `_wputenv`, ou na primeira chamada para `_wgetenv` se um ambiente MBCS já existir, um ambiente correspondente de cadeia de caracteres largos será criado. Em seguida, a variável global `_wenviron`, uma versão de caractere largo da variável global `_environ`, apontará para esse ambiente. Nesse momento, duas cópias do ambiente (MBCS e Unicode) existirão simultaneamente e serão mantidas pelo sistema operacional durante toda a vida do programa.

Da mesma forma, se seu programa usar uma função **wmain**, um ambiente de caractere amplo será criado na inicialização do programa e apontado pela variável global `_wenviron`. Um ambiente de MBCS (ASCII) é criado na primeira chamada para `_putenv` ou `getenv` e apontado pela variável global `_environ`.

Para obter mais informações sobre o ambiente de MBCS, consulte Internacionalização na *Referência da biblioteca em tempo de execução.*

FINAL específico da Microsoft

Veja também

Execução principal da função e do programa

Descrição do argumento

13/05/2021 • 2 minutes to read

O parâmetro `argc` nas funções **main** e **wmain** é um inteiro que especifica quantos argumentos são transmitidos ao programa na linha de comando. Como o nome do programa é considerado um argumento, o valor de `argc` é pelo menos um.

Comentários

O parâmetro `argv` é uma matriz de ponteiros para cadeias de caracteres de terminação nula que representam os argumentos do programa. Cada elemento da matriz aponta para uma representação de cadeia de caracteres de um argumento transmitido a **main** (ou a **wmain**). (Para obter informações sobre matrizes, consulte declarações de matriz.) O `argv` parâmetro pode ser declarado como uma matriz de ponteiros para o tipo `char` (`char *argv[]`) ou como um ponteiro para ponteiros para o tipo `char` (`char **argv`). Para **wmain**, o `argv` parâmetro pode ser declarado como uma matriz de ponteiros para Type `wchar_t` (`wchar_t *argv[]`) ou como um ponteiro para ponteiros para Type `wchar_t` (`wchar_t **argv`).

Por convenção, `argv` **[0]** é o comando com que o programa é invocado. No entanto, é possível gerar um processo usando CreateProcess e, se você usar o primeiro e o segundo argumentos (`lpApplicationName` e `lpCommandLine`), `argv` **[0]** talvez não seja o nome do executável; use GetModuleFileName para recuperar o nome do executável.

O último ponteiro (`argv[argc]`) é **NULL**. (Consulte getenv na *Referência da biblioteca em tempo de execução* para conhecer um método alternativo para obter informações sobre a variável de ambiente.)

Específico da Microsoft

O parâmetro `envp` é um ponteiro para uma matriz de cadeias de caracteres de terminação nula que representam os valores definidos nas variáveis de ambiente do usuário. O `envp` parâmetro pode ser declarado como uma matriz de ponteiros para `char` (`char *envp[]`) ou como um ponteiro para ponteiros para `char` (`char **envp`). Em uma função **wmain** , o `envp` parâmetro pode ser declarado como uma matriz de ponteiros para `wchar_t` (`wchar_t *envp[]`) ou como um ponteiro para ponteiros para `wchar_t` (`wchar_t **envp`). O encerramento da matriz é indicado por um ponteiro **NULL** *. Observe que o bloco de ambiente transmitido para **main** ou **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` / `_wgetenv` e as `_environ` `_wenviron` variáveis ou) será alterado, mas o bloco apontado por `envp` não será alterado. O parâmetro `envp` é compatível com ANSI em C, mas não em C++.

FINAL específico da Microsoft

Veja também

Execução principal da função e do programa

Expandindo argumentos de curinga

13/05/2021 • 2 minutes to read

A expansão de argumento curinga é específica da Microsoft.

Ao executar um programa C, você pode usar qualquer um dos dois curingas, o ponto de interrogação (`?`) e o asterisco (`*`), para especificar os argumentos filename e Path na linha de comando.

Por padrão, os curingas não são expandidos em argumentos de linha de comando. Você pode substituir a rotina de carregamento de vetor de argumento normal `argv` por uma versão que expanda curingas vinculando-se ao `setargv.obj` `wsetargv.obj` arquivo ou. Se o seu programa usar uma `main` função, vincule com `setargv.obj` . Se o seu programa usar uma `wmain` função, vincule com `wsetargv.obj` . Ambos têm comportamento equivalente.

Para vincular com `setargv.obj` ou `wsetargv.obj` , use a `/link` opção. Por exemplo:

```
cl example.c /link setargv.obj
```

Os caracteres curinga são expandidos da mesma maneira que os comandos do sistema operacional.

Confira também

Opções de link
`main` execução de função e programa

Analisando argumentos de linha de comando C

13/05/2021 • 2 minutes to read

Específico da Microsoft

O código de inicialização do Microsoft 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 primeiro argumento (`argv[0]`) é tratado especialmente. Ele representa o nome do programa. Como ele deve ser um nome de caminho válido, partes entre aspas duplas (`"`) são permitidas. As aspas duplas não são incluídas na `argv[0]` saída. As partes entre aspas duplas impedem a interpretação de um espaço ou caractere de tabulação como o final do argumento. As regras posteriores desta lista não se aplicam.
- Uma cadeia de caracteres entre aspas duplas é interpretada como um único argumento, independentemente de conter ou não espaço em branco. Uma cadeia de caracteres entre aspas pode ser inserida em um argumento. O cursor (`^`) não é reconhecido como um caractere de escape ou delimitador. Dentro de uma cadeia de caracteres entre aspas, um par de aspas duplas é interpretado como uma aspa dupla de escape simples. Se a linha de comando terminar antes de uma marca de aspas duplas de fechamento ser encontrada, todos os caracteres lidos até agora serão gerados como o último argumento.
- Uma aspa dupla precedida por uma barra invertida (`\"`) é interpretada como uma aspa dupla literal (`"`).
- As barras invertidas são interpretadas literalmente, a menos que precedam imediatamente uma aspa dupla.
- Se um número par de barras invertidas for seguido por uma aspa dupla, uma barra invertida (`\`) será colocada na `argv` matriz para cada par de barras invertidas (`\\`), e a marca de aspas duplas (`"`) será interpretada 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 `argv` matriz para cada par de barras invertidas (`\\`). A marca de aspas duplas é interpretada como uma sequência de escape pela barra invertida restante, fazendo com que uma aspa dupla literal (`"`) seja colocada `argv` .

Esta lista ilustra as regras acima mostrando o resultado interpretado passado para `argv` por vários exemplos de argumentos de linha de comando. O resultado listado na segunda, terceira e quarta coluna são do programa de ARGS.C que segue a lista.

ENTRADA DE LINHA DE COMANDO	ARGV[1]	ARGV[2]	ARGV[3]
`"a b c" d e`	`a b c`	`d`	`e`
`"ab\"c" "\\" d`	`ab"c`	`\`	`d`
`a\\\b d"e f"g h`	`a\\\b`	`de fg`	`h`

ENTRADA DE LINHA DE COMANDO	ARGV[1]	ARGV[2]	ARGV[3]
`a\\\"b c d`	`a\"b`	`c`	`d`
`a\\\\"b c" d e`	`a\\b c`	`d`	`e`
`a"b"" c d`	`ab" c d`		

Exemplo

Código

```
// ARGS.C illustrates the following variables used for accessing
// command-line arguments and environment variables:
// argc  argv  envp
//

#include <stdio.h>

int main( int argc, // Number of strings in array argv
char *argv[],      // Array of command-line argument strings
char **envp )      // Array of environment variable strings
{
    int count;

    // Display each command-line argument.
    printf_s( "\nCommand-line arguments:\n" );
    for( count = 0; count < argc; count++ )
        printf_s( "  argv[%d]   %s\n", count, argv[count] );

    // Display each environment variable.
    printf_s( "\nEnvironment variables:\n" );
    while( *envp != NULL )
        printf_s( "  %s\n", *(envp++) );

    return;
}
```

Comentários

Um exemplo de resultado deste programa é:

```
Command-line arguments:
  argv[0]   C:\MSC\TEST.EXE

Environment variables:
  COMSPEC=C:\NT\SYSTEM32\CMD.EXE

  PATH=c:\nt;c:\binb;c:\binr;c:\nt\system32;c:\word;c:\help;c:\msc;c:\;
  PROMPT=[$p]
  TEMP=c:\tmp
  TMP=c:\tmp
  EDITORS=c:\binr
  WINDIR=c:\nt
```

FINAL específico da Microsoft

Confira também

Execução principal da função e do programa

Personalizando processamento de linha de comando C

13/05/2021 • 2 minutes to read

Se o seu programa não usar argumentos de linha de comando, você poderá suprimir a rotina de processamento de linha de comando para economizar uma pequena quantidade de espaço. Para suprimir seu uso, inclua o `noarg.obj` arquivo (para `main` e `wmain`) nas `/link` Opções do compilador ou na `LINK` linha de comando.

Da mesma forma, se você nunca acessar a tabela de ambiente por meio do `envp` argumento, poderá suprimir a rotina interna de processamento de ambiente. Para suprimir seu uso, inclua o `noenv.obj` arquivo (para `main` e `wmain`) nas `/link` Opções do compilador ou na `LINK` linha de comando.

Para obter mais informações sobre as opções do vinculador de inicialização do tempo de execução, consulte Opções de link.

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

Confira também

`main` execução de função e programa
Opções de link.

Tempo de vida, escopo, visibilidade e ligação

13/05/2021 • 2 minutes to read

Para entender o funcionamento de um programa de C, você deve entender as regras que determinam como as variáveis e funções podem ser usadas no programa. Diversos conceitos a seguir são cruciais para entender essas regras:

- Tempo de vida
- Escopo e visibilidade
- Vinculação

Veja também

Estrutura do programa

Tempo de vida

13/05/2021 • 2 minutes to read

O "tempo de vida" é o período durante a execução de um programa em que uma variável ou uma função existe. A duração de armazenamento de identificador determina seu tempo de vida.

Um identificador declarado com o *especificador de classe de armazenamento* `static` tem duração de armazenamento estático. Os identificadores com a duração de armazenamento estático (também chamados de "globais") têm o armazenamento e um valor definido para a duração de um programa. O armazenamento é reservado e o valor armazenado do identificador é inicializado apenas uma vez, antes da inicialização do programa. Um identificador declarado com vinculação externa ou interna também tem duração de armazenamento estático (consulte Vinculação).

Um identificador declarado sem o `static` especificador de classe de armazenamento tem duração de armazenamento automática se for declarado dentro de uma função. Um identificador com duração automática de armazenamento (um "identificador local") tem armazenamento e um valor definido apenas dentro do bloco no qual que o identificador está definido ou declarado. Um identificador automático é alocado ao novo armazenamento sempre que o programa entrar nesse bloco, e perde seu armazenamento (e seu valor) quando o programa sai do bloco. Os identificadores declarados em uma função sem vinculação também têm a duração automática de armazenamento.

As regras a seguir especificam se um identificador tem tempo de vida global (estático) ou local (automático):

- Todas as funções têm tempo de vida estático. Portanto, elas existem em sempre durante a execução do programa. Os identificadores declarados no nível externo (ou seja, fora de todos os blocos no programa no mesmo nível das definições de função) sempre têm o tempo de vida global (estático).
- Se uma variável local tiver um inicializador, a variável será inicializada toda vez que for criada (a menos que seja declarada como `static`). Os parâmetros de função também têm o tempo de vida local. Você pode especificar o tempo de vida global para um identificador dentro de um bloco, incluindo o `static` especificador de classe de armazenamento em sua declaração. Depois de declarado `static` , a variável retém seu valor de uma entrada do bloco para o próximo.

Embora exista um identificador com um tempo de vida global durante a execução do programa de origem (por exemplo, uma variável declarada externamente ou uma variável local declarada com a `static` palavra-chave), ela pode não estar visível em todas as partes do programa. Consulte Escopo e visibilidade para obter informações sobre visibilidade; consulte Classes de armazenamento para visualizar uma discussão do não terminal *storage-class-specifier*.

A memória pode ser alocada conforme necessário (dinâmica) se criada com o uso de rotinas de biblioteca especiais como `malloc`. Como a alocação de memória dinâmica usa rotinas de biblioteca, não é considerado como parte da linguagem. Consulte a função malloc na *Referência da biblioteca em tempo de execução*.

Veja também

Tempo de vida, escopo, visibilidade e vinculação

Escopo e visibilidade

13/05/2021 • 2 minutes to read

A "visibilidade" de um identificador determina as partes do programa nas quais pode ser feita uma referência a ele — seu "escopo". Um identificador é visível (ou seja, pode ser usado) somente nas partes de um programa abrangidas por seu "escopo", que pode ser limitado (em ordem crescente de restrição) ao arquivo, função, bloco ou protótipo de função em que aparece. O escopo de um identificador é a parte do programa na qual o nome pode ser usado. Isso é às vezes chamado de "escopo léxico". Há quatro tipos de escopo: de função, de arquivo, de bloco e de protótipo de função.

Todos os identificadores, exceto os rótulos, têm seu escopo determinado pelo nível em que a declaração ocorre. As seguintes regras para cada tipo de escopo controlam a visibilidade dos identificadores em um programa:

Escopo do arquivo O declarador ou especificador de tipo de um identificador com escopo de arquivo aparece fora de qualquer bloco ou lista de parâmetros e está acessível de qualquer local da unidade de tradução após sua declaração. Os nomes de identificadores com escopo de arquivo são frequentemente denominados "globais" ou "externos". O escopo de um identificador global inicia no ponto da sua definição ou declaração e termina no final da unidade de tradução.

Escopo de função Um rótulo é o único tipo de identificador que tem escopo de função. Um rótulo é declarado implicitamente por seu uso em uma instrução. Os nomes de rótulos devem ser exclusivos dentro de uma função. (Para obter mais informações sobre rótulos e nomes de rótulo, consulte Instruções goto e identificadas.)

Escopo de bloco O declarador ou especificador de tipo de um identificador de escopo de bloco ocorre dentro de um bloco ou uma lista de declarações de parâmetros formais em uma definição de função. Só é visível do ponto de sua declaração ou definição até o final do bloco que contém sua declaração ou definição. O escopo é limitado a esse bloco e a todos os blocos aninhados naquele bloco e termina na chave que fecha o bloco associado. Esses identificadores às vezes são chamados de "variáveis locais".

Escopo de protótipo de função O declarador ou especificador de tipo de um identificador com escopo de protótipo de função aparece na lista de declarações de parâmetro em um protótipo de função (que não parte da declaração da função). O escopo termina no final do declarador da função.

As declarações apropriadas para tornar as variáveis visíveis em outros arquivos de origem são descritas em Classes de armazenamento. No entanto, variáveis e funções declaradas no nível externo com o `static` especificador de classe de armazenamento são visíveis somente dentro do arquivo de origem no qual elas são definidas. Todas as funções restantes são visíveis globalmente.

Veja também

Tempo de vida, escopo, visibilidade e vinculação

Resumo de tempo de vida e visibilidade

13/05/2021 • 2 minutes to read

A tabela a seguir é um resumo das características do tempo de vida e de visibilidade para a maioria dos identificadores. As três primeiras colunas dão os atributos que definem o tempo de vida e visibilidade. Um identificador com os atributos dados pelas três primeiras colunas tem o tempo de vida e a visibilidade mostrados na quarta e quinta colunas. No entanto, a tabela não abrange todos os casos possíveis. Consulte Classes de armazenamento para obter mais informações.

Resumo de tempo de vida e visibilidade

ATRIBUTOS: LEVEL	ITEM	STORAGE-CLASS ESPECIFICADOR	RESULTADO: TEMPO DE VIDA	VISIBILIDADE
Escopo de arquivo	Definição de variável	`static`	Global	Restante do arquivo de origem no qual ocorre
	Declaração de variável	`extern`	Global	Restante do arquivo de origem no qual ocorre
	Protótipo ou definição de função	`static`	Global	Arquivo único de origem
	Protótipo da função	`extern`	Global	Restante do arquivo de origem
Escopo de bloco	Declaração de variável	`extern`	Global	Bloquear
	Definição de variável	`static`	Global	Bloquear
	Definição de variável	`auto` ou `register`	Local	Bloquear

Exemplo

Descrição

O exemplo a seguir ilustra blocos, aninhamento e visibilidade das variáveis:

Código

```
// Lifetime_and_Visibility.c

#include <stdio.h>

int i = 1;  // i defined at external level

int main()  // main function defined at external level
{
    printf_s( "%d\n", i ); // Prints 1 (value of external level i)
    {                                  // Begin first nested block
        int i = 2, j = 3;          // i and j defined at internal level
        printf_s( "%d %d\n", i, j );  // Prints 2, 3
        {                              // Begin second nested block
            int i = 0;                 // i is redefined
            printf_s( "%d %d\n", i, j ); // Prints 0, 3
        }                              // End of second nested block
        printf_s( "%d\n", i );         // Prints 2 (outer definition
                                       //  restored)
    }                                  // End of first nested block
    printf_s( "%d\n", i );             // Prints 1 (external level
                                       // definition restored)
    return 0;
}
```

Comentários

Neste exemplo, há quatro níveis de visibilidade: o nível externo e três níveis de bloco. Os valores são impressos na tela como observado nos comentários após cada instrução.

Veja também

Tempo de vida, escopo, visibilidade e vinculação

Vinculação

13/05/2021 • 2 minutes to read

Os nomes de identificadores podem fazer referência aos identificadores diferentes em escopos diferentes. Um identificador declarado em escopos diferentes ou no mesmo escopo mais de uma vez pode ser feito para fazer referência ao mesmo identificador ou função por um processo chamado “vinculação”. A vinculação determina as partes do programa nas quais um identificador pode ser referenciado (sua "visibilidade"). Existem três tipos de vinculação: interna, externa e nenhuma vinculação.

Veja também

Usando extern para especificar a ligação

Ligação interna

13/05/2021 • 2 minutes to read

Se a declaração de um identificador de escopo de arquivo para um objeto ou uma função contiver o *especificador de classe de armazenamento* `static` , o identificador terá um vínculo interno. Caso contrário, o identificador terá vinculação externa. Consulte Classes de armazenamento para obter uma discussão do *storage-class-specifier* não terminal.

Em uma unidade de conversão, cada instância de um identificador com vinculação interna denota o mesmo identificador ou a mesma função. Os identificadores vinculados internamente são exclusivos de uma unidade de conversão.

Veja também

Usando extern para especificar a ligação

Ligação externa

13/05/2021 • 2 minutes to read

Se a primeira declaração no nível de escopo de arquivo para um identificador não usar o `static` especificador de classe de armazenamento, o objeto terá vínculo externo.

Se a declaração de um identificador para uma função não tiver um *especificador de classe de armazenamento*, seu vínculo será determinado exatamente como se ele fosse declarado com o *especificador de classe de armazenamento* `extern` . Se a declaração de um identificador para um objeto tiver escopo de arquivo e nenhum *storage-class-specifier*, a ligação será externa.

O nome de um identificador de vinculação externa designa a mesma função ou o mesmo objeto de dados como qualquer outra declaração para o mesmo nome com vinculação externa. As duas declarações podem estar na mesma unidade de tradução ou em unidades de tradução diferentes. Se o objeto ou a função também tiverem tempo de vida global, o objeto ou a função são compartilhados em todo o programa.

Veja também

Usando extern para especificar a ligação

Sem ligação

13/05/2021 • 2 minutes to read

Se uma declaração de um identificador dentro de um bloco não incluir o `extern` especificador de classe de armazenamento, o identificador não terá nenhum vínculo e será exclusivo para a função.

Os seguintes identificadores não têm nenhuma vinculação:

- Um identificador declarado como qualquer coisa que não seja um objeto ou uma função
- Um identificador declarado como um parâmetro de função
- Um identificador de escopo de bloco para um objeto declarado sem o `extern` especificador de classe de armazenamento

Se um identificador não tem nenhuma vinculação, declarar o mesmo nome novamente (em um declarador ou especificador de tipo) no mesmo nível de escopo gera um erro de redefinição de símbolo.

Veja também

Usando extern para especificar a ligação

Namespaces

13/05/2021 • 2 minutes to read

O compilador configura os "name spaces" para distinguir entre os identificadores usados para tipos diferentes de itens. Os nomes em cada name space devem ser exclusivos para evitar conflitos, mas um nome idêntico pode aparecer em mais de um name space. Isso significa que você pode usar o mesmo identificador de dois ou mais itens diferentes, contanto que os itens estejam em name spaces diferentes. O compilador pode resolver referências com base no contexto sintático do identificador no programa.

> **NOTE**
> Não confunda a noção limitada de um name space em C com o recurso "namespace" em C++. Consulte Namespaces, na referência da linguagem C++ para obter mais informações.

Esta lista descreve os name spaces usados em C.

Rótulos de instrução Rótulos de instrução nomeados fazem parte das instruções. As definições de rótulos de instrução são sempre seguidas por dois-pontos, mas não fazem parte dos `case` Rótulos. Os usos de rótulos de instrução sempre seguem imediatamente a palavra-chave `goto` . Os rótulos de instrução não precisam ser diferentes de outros nomes ou nomes de rótulo em outras funções.

Marcas de estrutura, União e enumeração essas marcas fazem parte dos especificadores de tipo de estrutura, União e enumeração e, se presentes, sempre seguem imediatamente as palavras reservadas `struct` , `union` ou `enum` . Os nomes de marcas devem ser diferentes de todas as outras marcas de estrutura, enumeração e união com a mesma visibilidade.

Membros de estruturas ou uniões Os nomes de membros são alocados em namespaces associados a cada tipo de estrutura e união. Ou seja, o mesmo identificador pode ser um nome de componente em várias estruturas ou uniões ao mesmo tempo. As definições de nomes de componente sempre ocorrem dentro dos especificadores do tipo estrutura ou união. Os usos de nomes de componentes sempre seguem imediatamente os operadores de seleção de Membros (`->` e `.`). O nome de um membro deve ser exclusivo dentro da estrutura ou da união, mas não precisa ser diferente de outros nomes no programa, inclusive os nomes dos membros de estruturas e uniões diferentes ou o nome da própria estrutura.

Identificadores comuns Todos os outros nomes ficam em um namespace que inclui variáveis, funções (incluindo parâmetros formais e variáveis locais) e constantes de enumeração. Os nomes de identificadores têm visibilidade aninhada, para que você possa redefini-los nos blocos.

Nomes Typedef Os nomes Typedef não podem ser usados como identificadores no mesmo escopo.

Por exemplo, como as marcas de estrutura, os membros de estrutura e os nomes de variável estão em três name spaces diferentes, os três itens nomeados `student` neste exemplo não estão em conflito. O contexto de cada item permite a interpretação correta de cada ocorrência de `student` no programa. (Para obter informações sobre estruturas, consulte Declarações de estrutura.)

```
struct student {
   char student[20];
   int class;
   int id;
   } student;
```

Quando `student` aparece após a `struct` palavra-chave, o compilador o reconhece como uma marca de estrutura. Quando `student` aparece depois de um operador de seleção de membro (`->` ou `.`), o nome refere-se ao membro da estrutura. Em outros contextos, `student` se refere à variável da estrutura. Porém, sobrecarregar a marca name space não é recomendado porque obscurece o significado.

Veja também

Estrutura do programa

Alinhamento (C11)

13/05/2021 • 4 minutes to read

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 da arquitetura de hardware.

As CPUs lêem e gravam a memória com mais eficiência quando os dados são armazenados em um endereço que é um múltiplo do tamanho dos dados. Por exemplo, um inteiro de 4 bytes será acessado com mais eficiência se estiver armazenado em um endereço que seja um múltiplo de 4. Quando os dados não estão alinhados, a CPU faz mais trabalho para acessar os dados.

Por padrão, o compilador alinha os dados com base em seu tamanho: `char` em um limite de 1 byte, `short` em um limite de 2 bytes, `int` , `long` e `float` em um limite de 4 bytes, `double` no limite de 8 bytes, e assim por diante.

Além disso, ao alinhar os dados usados com frequência com o tamanho de linha de cache do processador, você pode melhorar 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 as instâncias da estrutura sejam armazenadas em cache com eficiência.

Normalmente, você não precisa se preocupar com o alinhamento. O compilador geralmente alinha os dados em limites naturais que se baseiam no processador de destino e no tamanho dos dados. Os dados são alinhados em limites de até 4 bytes em processadores de 32 bits e limites de 8 bytes em processadores de 64 bits. Em alguns casos, no entanto, você pode obter melhorias de desempenho ou economia de memória, especificando um alinhamento personalizado para suas estruturas de dados.

Use a palavra-chave C11 `_Alignof` para obter o alinhamento preferencial de um tipo ou variável e `_Alignas` para especificar um alinhamento personalizado para uma variável ou tipo definido pelo usuário.

As macros de conveniência `alignof` e `alignas` , definidas no `<stdalign.h>` , são mapeadas diretamente para `_Alignof` e `_Alignas` , respectivamente. Essas macros correspondem às palavras-chave usadas em C++. Portanto, usar as macros em vez das palavras-chave C pode ser útil para portabilidade de código se você compartilhar qualquer código entre os dois idiomas.

`alignas` e `_Alignas` (C11)

Use `alignas` ou `_Alignas` para especificar o alinhamento personalizado para uma variável ou tipo definido pelo usuário. Eles podem ser aplicados a uma struct, União, enumeração ou variável.

Sintaxe de `alignas`

```
alignas(type)
alignas(constant-expression)
_Alignas(type)
_Alignas(constant-expression)
```

Comentários

`_Alignas` Não pode ser usado na declaração de um typedef, campo de bits, função, parâmetro de função ou um objeto declarado com o `register` especificador.

Especifique um alinhamento que seja uma potência de dois como 1, 2, 4, 8, 16 e assim por diante. Não use um valor menor do que o tamanho do tipo.

Structs e Unions têm um alinhamento igual ao maior alinhamento de qualquer membro. Os bytes de preenchimento são adicionados dentro de structs para garantir que os requisitos de alinhamento de membro individual sejam atendidos.

Se houver vários `alignas` especificadores em uma declaração (por exemplo, um struct com vários membros que têm `alignas` especificadores diferentes), o alinhamento da estrutura será aquele do maior.

`alignas` exemplo

Este exemplo usa a macro de conveniência `alignof` porque é portátil para C++. O comportamento é o mesmo se você usar `_Alignof` .

```
// Compile with /std:c11

#include <stdio.h>
#include <stdalign.h>

typedef struct
{
    int value; // aligns on a 4-byte boundary. There will be 28 bytes of padding between value and alignas
    alignas(32) char alignedMemory[32]; // assuming a 32 byte friendly cache alignment
} cacheFriendly; // this struct will be 32-byte aligned because alignedMemory is 32-byte alligned and is the
largest alignment specified in the struct

int main()
{
    printf("sizeof(cacheFriendly): %d\n", sizeof(cacheFriendly)); // 4 bytes for int value + 32 bytes for
alignedMemory[] + padding to ensure  alignment
    printf("alignof(cacheFriendly): %d\n", alignof(cacheFriendly)); // 32 because alignedMemory[] is aligned
on a 32-byte boundary

    /* output
        sizeof(cacheFriendly): 64
        alignof(cacheFriendly): 32
    */
}
```

`alignof` e `_Alignof` (C11)

`_Alignof` Retorna o alinhamento em bytes do tipo especificado. Ele retorna um valor do tipo `size_t` .

Sintaxe de `alignof`

```
alignof(type)
_Alignof(type)
```

`alignof` exemplo

Este exemplo usa a macro de conveniência `alignof` porque é portátil para C++. O comportamento é o mesmo se você usar `_Alignof` .

```
// Compile with /std:c11

#include <stdalign.h>
#include <stdio.h>

int main()
{
    size_t alignment = alignof(short);
    printf("alignof(short) = %d\n", alignment); // 2
    printf("alignof(int) = %d\n", alignof(int)); // 4
    printf("alignof(long) = %d\n", alignof(long)); // 4
    printf("alignof(float) = %d\n", alignof(float)); // 4
    printf("alignof(double) = %d\n", alignof(double)); // 8

    typedef struct
    {
        int a;
        double b;
    } test;

    printf("alignof(test) = %d\n", alignof(test)); // 8 because that is the alignment of the largest element
in the structure

    /* output

       alignof(short) = 2
       alignof(int) = 4
       alignof(long) = 4
       alignof(float) = 4
       alignof(double) = 8
       alignof(test) = 8
    */
}
```

Requisitos

std: c++ 11 ou posterior é necessário.

SDK do Windows versão 10.0.20201.0 ou posterior. Esta versão é atualmente uma compilação Insider, que pode ser baixada de downloads da versão prévia do Windows Insider. Consulte C11 e C17: introdução para obter instruções sobre como instalar e usar esse SDK.

Consulte também

`/std` (Especifique a versão padrão do idioma)
C++ `alignof` e `alignas`
Manipulação de alinhamento de dados do compilador

Declarações e tipos

13/05/2021 • 2 minutes to read

Esta seção descreve a declaração e a inicialização de variáveis, funções e tipos. A linguagem C inclui um conjunto padrão de tipos de dados básicos. Você também pode adicionar seus próprios tipos de dados, chamados "tipos derivados", declarando novos com base em tipos já definidos. Os seguintes tópicos são abordados:

- Visão geral das declarações
- Classes de armazenamento
- Especificadores de tipo
- Qualificadores de tipo
- Declaradores e declarações de variáveis
- Interpretando declaradores mais complexos
- Initialization
- Armazenamento de tipos básicos
- Tipos incompletos
- Declarações typedef
- Atributos de classe de armazenamento estendido

Veja também

Referência da linguagem C

Visão geral das declarações

13/05/2021 • 4 minutes to read

Uma "declaração" especifica a interpretação e os atributos de um conjunto de identificadores. Uma declaração que também causa a reserva do armazenamento para o objeto ou a função nomeada pelo identificador é chamada de uma "definição". As declarações de C para variáveis, funções e tipos têm esta sintaxe:

Sintaxe

`declaration`:
  `declaration-specifiers` optar por `attribute-seq` aceitar `init-declarator-list` `;`

/* `attribute-seq` a opção opt é específica da Microsoft */

`declaration-specifiers`:
  `storage-class-specifier` ** `declaration-specifiers` aceitar
  `type-specifier` ** `declaration-specifiers` aceitar
  `type-qualifier` ** `declaration-specifiers` aceitar

`init-declarator-list`:
  `init-declarator`
  `init-declarator-list` `,` `init-declarator`

`init-declarator`:
  `declarator`
  `declarator` `=` `initializer`

> **NOTE**
> Essa sintaxe para `declaration` não é repetida nas seções a seguir. A sintaxe nas seções a seguir geralmente começa com o não `declarator` terminal.

As declarações no `init-declarator-list` contêm os identificadores que estão sendo nomeados; `init` é uma abreviação para o inicializador. O `init-declarator-list` é uma sequência de declaradores separados por vírgula, cada um dos quais pode ter informações de tipo adicionais, ou um inicializador, ou ambos. O `declarator` contém os identificadores, se houver, que estão sendo declarados. O `declaration-specifiers` nonterminal consiste em uma sequência do tipo e especificadores de classe de armazenamento que indicam a ligação, a duração do armazenamento e, pelo menos, parte do tipo das entidades que os declaradores denotam. As declarações são feitas de alguma combinação de especificadores de classe de armazenamento, especificadores de tipo, qualificadores de tipo, declaradores e inicializadores.

Declarações podem conter um ou mais dos atributos opcionais listados em `attribute-seq` ; `seq` é uma abreviação para Sequence. Esses atributos específicos da Microsoft executam várias funções, que são discutidas em detalhes em todo este livro.

Na forma geral de uma declaração de variável, `type-specifier` fornece o tipo de dados da variável. O `type-specifier` pode ser um composto, como quando o tipo é modificado por `const` ou `volatile` . O `declarator` fornece o nome da variável, possivelmente modificado para declarar uma matriz ou um tipo de ponteiro. Por exemplo:

```
int const *fp;
```

declara uma variável chamada `fp` como um ponteiro para um valor não modificável (`const`) `int` . Você pode definir mais de uma variável em uma declaração usando vários declaradores, separados por vírgulas.

Uma declaração deve ter pelo menos um declarador ou seu especificador de tipo deve declarar uma marca de estrutura, marca de união ou membros de uma enumeração. Os declaradores fornecem toda quaisquer outras informações sobre um identificador. Um Declarador é um identificador que pode ser modificado com colchetes (`[ ]`), asteriscos (`*`) ou parênteses (`( )`) para declarar um tipo de matriz, ponteiro ou função, respectivamente. Quando você declara variáveis simples (como o caractere, inteiro e itens de ponto flutuante), ou estruturas e uniões de variáveis simples, `declarator` é apenas um identificador. Para obter mais informações sobre os declaradores, consulte Declaradores e declarações de variável.

Todas as definições são implicitamente declarações, mas nem todas as declarações são definições. Por exemplo, as declarações de variáveis que começam com o `extern` especificador de classe de armazenamento são "referenciando", em vez de "definindo" declarações. Se uma variável externa for referenciada antes de ser definida, ou se ela for definida em outro arquivo de origem daquele em que é usada, uma `extern` declaração será necessária. O armazenamento não é alocado "referenciando" declarações, nem as variáveis podem ser inicializados em declarações.

Uma classe de armazenamento ou um tipo (ou ambos) são necessários em declarações de variáveis. Exceto para `__declspec` , somente um especificador de classe de armazenamento é permitido em uma declaração e nem todos os especificadores de classe de armazenamento são permitidos em cada contexto. A `__declspec` classe de armazenamento é permitida com outros especificadores de classe de armazenamento e é permitida mais de uma vez. O especificador de classe de armazenamento de uma declaração afeta como o item declarado é armazenado e inicializado, e quais partes de um programa podem fazer referência ao item.

Os *`storage-class-specifier`* terminais definidos em C incluem `auto` , `extern` , `register` , `static` e `typedef` . O Microsoft C também inclui o *`storage-class-specifier`* terminal `__declspec` . Todos os *`storage-class-specifier`* terminais exceto `typedef` e `__declspec` são discutidos em classes de armazenamento. Para obter informações sobre o `typedef` , consulte `typedef` declarações. Para obter informações sobre o `__declspec` , consulte extended Storage-Class Attributes.

O local da declaração no programa de origem e da presença ou ausência de outras instruções da variável são fatores importantes na determinação de diversas variáveis. Pode haver várias redeclarações mas apenas uma definição. No entanto, uma definição pode aparecer em mais de uma unidade de tradução. Para objetos com vinculação interna, esta regra é aplicada separadamente a cada unidade de tradução, pois os objetos internamente vinculados são exclusivos para uma unidade de tradução. Para objetos com vinculação externa, essa regra se aplica ao programa inteiro. Para obter mais informações sobre visibilidade, consulte tempo de vida, escopo, visibilidade e vinculação.

Os especificadores do tipo fornecem algumas informações sobre os tipos de dados de identificadores. O especificador de tipo padrão é `int` . Para obter mais informações, consulte Especificadores de tipos. Os especificadores de tipo também podem definir marcas de tipo, estruturas e nomes de componentes da união, além de constantes da enumeração. Para obter mais informações, consulte declarações de enumeração, declarações de estruturae declarações de União.

Há dois *`type-qualifier`* terminais: `const` e `volatile` . Esses qualificadores especificam propriedades adicionais dos tipos que só são relevantes apenas no acesso de objetos desse tipo por valores l. Para obter mais informações sobre o `const` e o `volatile` , consulte qualificadores de tipo. Para uma definição de l-values, consulte Expressões de l-value e r-value.

Veja também

Classes de armazenamento C

13/05/2021 • 2 minutes to read

A "classe de armazenamento" de uma variável determina se o item tem um tempo de vida “global” ou “local”. C chama esse dois tempos de vida de “estático” e “automático”. Um item com um tempo de vida global existe e tem um valor durante toda a execução do programa. Todas as funções têm tempos de vida globais.

As variáveis automáticas, ou variáveis com tempo de vida local, recebem novos armazenamentos cada vez que o controle de execução é transmitido para o bloco no qual elas são definidas. Quando a execução retornar, as variáveis não terão mais valores significativos.

C fornece os seguintes especificadores de classe de armazenamento:

Sintaxe

storage-class-specifier:

`auto`
`register`
`static`
`extern`
`typedef`
** `__declspec (` **Extended-decl-Modifier-Seq* `)` / * Específico da Microsoft*/

Exceto para `__declspec` , você pode usar apenas um *especificador de classe de armazenamento* no *especificador de declaração* em uma declaração. Se nenhuma especificação de classe de armazenamento for feita, as declarações de um bloco criarão objetos automáticos.

Os itens declarados com o `auto` `register` especificador ou têm tempos de vida locais. Os itens declarados com o `static` `extern` especificador ou têm tempos de vida globais.

Como `typedef` e `__declspec` são semanticamente diferentes dos outros quatro terminais de *classe de armazenamento* , eles são discutidos separadamente. Para obter informações específicas sobre o `typedef` , consulte `typedef` declarações. Para obter informações específicas sobre o `__declspec` , consulte extended Storage-Class Attributes.

O posicionamento de declarações de variável e de função em arquivos de origem também afeta a classe e a visibilidade de armazenamento. As declarações fora de todas as definições de função devem aparecer no "nível externo". As declarações em definições de função aparecem no "nível interno".

O significado exato de cada especificador de classe de armazenamento depende de dois fatores:

- Se a declaração aparece no nível externo ou interno
- Se o item que está sendo declarado é uma variável ou uma função

Especificadores de classe de armazenamento para declarações de nível externo e Especificadores de classe de armazenamento para declarações de nível interno descrevem os terminais *storage-class-specifier* em cada tipo de declaração e explicam o comportamento padrão quando o *storage-class-specifier* é omitido de uma variável. Especificadores de classe de armazenamento com declarações de função discutem os especificadores de classe de armazenamento usados com as funções.

Veja também

Especificadores de classe de armazenamento para declarações de nível externo

13/05/2021 • 4 minutes to read

As variáveis externas são variáveis no escopo de arquivo. São definidas fora de qualquer função, e estão potencialmente disponíveis para muitas funções. As funções só podem ser definidas no nível externo e, consequentemente, não podem ser aninhadas. Por padrão, todas as referências a variáveis externas e funções de mesmo nome são referências ao mesmo objeto, o que significa que elas têm *vínculo externo*. (Você pode usar a `static` palavra-chave para substituir esse comportamento.)

As declarações de variáveis no nível externo são definições de variáveis (*definindo declarações*) ou referências a variáveis definidas em outro lugar (*declarações de referência*).

Uma declaração de variável externa que também inicializa a variável (implícita ou explicitamente) é uma declaração de definição da variável. Uma definição no nível externo pode ter diversos formatos:

- Uma variável que você declara com o `static` especificador de classe de armazenamento. Você pode inicializar explicitamente a `static` variável com uma expressão constante, conforme descrito em inicialização. Se você omitir o inicializador, a variável será inicializado com 0 por padrão. Por exemplo, essas duas instruções são consideradas definições da variável `k`.

  ```
  static int k = 16;
  static int k;
  ```

- Uma variável inicializada explicitamente no nível externo. Por exemplo, `int j = 3;` é uma definição da variável `j`.

Em declarações de variáveis no nível externo (ou seja, fora de todas as funções), você pode usar o `static` `extern` especificador de classe de armazenamento ou ou omitir o especificador de classe de armazenamento inteiramente. Você não pode usar `auto` os `register` *`storage-class-specifier`* terminais e no nível externo.

Depois que uma variável é definida no nível externo, ela é visível durante o resto da unidade de tradução. A variável não é visível antes de sua declaração no mesmo arquivo de origem. Além disso, ela não é visível em outros arquivos de origem do programa, a menos que uma declaração de referência torne-a visível, como descrito a seguir.

As regras relacionadas a `static` incluem:

- Variáveis declaradas fora de todos os blocos sem a `static` palavra-chave sempre retêm seus valores em todo o programa. Para restringir o acesso a uma unidade de tradução específica, você deve usar a `static` palavra-chave. Isso lhes dá *vínculo interno*. Para torná-los globais a um programa inteiro, omita a classe de armazenamento explícita ou use a palavra-chave `extern` (consulte as regras na próxima lista). Isso lhes dá *vínculo externo*. As vinculações interna e externa também são discutidas em Vinculação.

- Você pode definir uma variável no nível externo apenas uma vez em um programa. Você pode definir outra variável com o mesmo nome e o `static` especificador de classe de armazenamento em uma unidade de tradução diferente. Como cada `static` definição é visível somente dentro de sua própria unidade de tradução, nenhum conflito ocorre. Ele fornece uma maneira útil de ocultar nomes de identificadores que devem ser compartilhados entre as funções de uma única unidade de tradução, mas não visíveis para outras unidades de tradução.

- O `static` especificador de classe de armazenamento também pode se aplicar a funções. Se você declarar uma função `static`, seu nome será invisível fora do arquivo no qual ele foi declarado.

As regras para uso do `extern` são:

- O `extern` especificador de classe de armazenamento declara uma referência a uma variável definida em outro lugar. Você pode usar uma `extern` declaração para tornar uma definição em outro arquivo de origem visível, ou para tornar uma variável visível antes de sua definição no mesmo arquivo de origem. Depois de declarar uma referência à variável no nível externo, a variável será visível no restante da unidade de tradução em que ocorre a referência declarada.
- Para que uma `extern` referência seja válida, a variável à qual ela se refere deve ser definida uma vez e somente uma vez no nível externo. Essa definição (sem a `extern` classe de armazenamento) pode estar em qualquer uma das unidades de tradução que compõem o programa.

Exemplo

O exemplo a seguir ilustra declarações externas:

```
/*************************************************************
                     SOURCE FILE ONE
*************************************************************/
#include <stdio.h>

extern int i;                // Reference to i, defined below
void next( void );           // Function prototype

int main()
{
    i++;
    printf_s( "%d\n", i );   // i equals 4
    next();
}

int i = 3;                  // Definition of i

void next( void )
{
    i++;
    printf_s( "%d\n", i );  // i equals 5
    other();
}

/*************************************************************
                     SOURCE FILE TWO
*************************************************************/
#include <stdio.h>

extern int i;              // Reference to i in
                           // first source file
void other( void )
{
    i++;
    printf_s( "%d\n", i ); // i equals 6
}
```

Os dois arquivos de origem neste exemplo contêm um total de três declarações externas de `i`. Apenas uma das declarações é uma "declaração de definição". Esta declaração,

```
int i = 3;
```

define a variável global `i` e a inicializa com valor inicial de 3. A declaração "referenciando" de `i` na parte superior do primeiro arquivo de origem usando `extern` torna a variável global visível antes de sua declaração de definição no arquivo. A declaração de referência de `i` no segundo arquivo de origem também torna a variável visível no arquivo de origem. Se uma instância de definição para uma variável não for fornecida na unidade de tradução, o compilador presume que existe uma

```
extern int x;
```

declaração de referência que uma referência de definição

```
int x = 0;
```

aparece em outra unidade de tradução do programa.

Todas as três funções, `main`, `next`, e `other`, executadas na mesma tarefa: aumentam `i` e a imprimem. Os valores 4, 5 e 6, são impressos.

Se a variável não `i` tiver sido inicializada, ela teria sido definida como 0 automaticamente. Nesse caso, os valores 1, 2, e 3 seriam impressos. Consulte Inicialização para obter mais informações sobre inicialização de variáveis.

Veja também

Classes de armazenamento C

Especificadores de classe de armazenamento para declarações de nível interno

13/05/2021 • 2 minutes to read

Você pode usar qualquer um dos quatro `storage-class-specifier` terminais para declarações de variáveis no nível interno. Quando você omite o `storage-class-specifier` de tal declaração, a classe de armazenamento padrão é `auto` . Portanto, a palavra-chave `auto` raramente é vista em um programa em C.

Veja também

Classes de armazenamento C

`auto` Especificador de Storage-Class

13/05/2021 • 2 minutes to read

O `auto` especificador de classe de armazenamento declara uma variável automática, uma variável com um tempo de vida local. Uma `auto` variável é visível somente no bloco no qual ela é declarada. Declarações de `auto` variáveis podem incluir inicializadores, conforme discutido na inicialização. Como as variáveis com `auto` classe de armazenamento não são inicializadas automaticamente, você deve inicializá-las explicitamente ao declará-las ou atribuí-las a valores iniciais em instruções dentro do bloco. Os valores das variáveis não inicializadas `auto` são indefinidos. (Uma variável local `auto` ou `register` classe de armazenamento é inicializada toda vez que chegar no escopo se um inicializador for fornecido.)

Uma `static` variável interna (uma variável estática com escopo de bloco ou local) pode ser inicializada com o endereço de qualquer `static` item ou externo, mas não com o endereço de outro `auto` Item, porque o endereço de um `auto` item não é uma constante.

Veja também

`auto` Chaves

Especificador de classe de armazenamento de registro

13/05/2021 • 2 minutes to read

Específico da Microsoft

O compilador do Microsoft C/C++ não honra as solicitações de variáveis de registro feitas pelo usuário. No entanto, para portabilidade, todas as demais semânticas associadas à `register` palavra-chave são respeitadas pelo compilador. Por exemplo, você não pode aplicar o operador address-of unário (**&**) a um objeto Register, nem a `register` palavra-chave ser usada em matrizes.

FINAL específico da Microsoft

Veja também

Especificadores de classe de armazenamento para declarações de Internal-Level

Especificador de classe de armazenamento estático

13/05/2021 • 2 minutes to read

Uma variável declarada no nível interno com o `static` especificador de classe de armazenamento tem um tempo de vida global, mas só é visível dentro do bloco no qual é declarada. Para cadeias de caracteres constantes, o uso do `static` é útil porque minimiza a sobrecarga da inicialização frequente em funções chamadas com frequência.

Comentários

Se você não inicializar uma variável explicitamente `static` , ela será inicializada como 0 por padrão. Dentro de uma função, `static` faz com que o armazenamento seja alocado e serve como uma definição. As variáveis estáticas internas fornecem armazenamento particular e permanente que só pode ser visto por uma única função.

Veja também

Classes de armazenamento C
Classes de armazenamento (C++)

Especificador de classe de armazenamento externa

13/05/2021 • 2 minutes to read

Uma variável declarada com o `extern` especificador de classe de armazenamento é uma referência a uma variável com o mesmo nome definido em outro arquivo de origem. Ela é usada para tornar visível a definição da variável de nível externo. Uma variável declarada como `extern` não tem armazenamento alocado para si mesma; é apenas um nome.

Exemplo

Este exemplo ilustra declarações de nível interno e externo:

```
// Source1.c

int i = 1;

// Source2. c

#include <stdio.h>

// Refers to the i that is defined in Source1.c:
extern int i;

void func(void);

int main()
{
    // Prints 1:
    printf_s("%d\n", i);
    func();
    return;
}

void func(void)
{
    // Address of global i assigned to pointer variable:
    static int *external_i = &i;

    // This definition of i hides the global i in Source.c:
    int i = 16;

    // Prints 16, 1:
    printf_s("%d\n%d\n", i, *external_i);
}
```

Neste exemplo, a variável `i` é definida em Source1.c com um valor inicial de 1. Uma `extern` declaração em origem2. c torna ' i ' visível nesse arquivo.

Na `func` função, o endereço da variável global `i` é usado para inicializar a variável de `static` ponteiro `external_i` . Isso funciona porque a variável global tem `static` tempo de vida, o que significa que seu endereço não é alterado durante a execução do programa. Em seguida, uma variável `i` é definida dentro do escopo de `func` como uma variável local com o valor inicial 16. Essa definição não afeta o valor da variável `i` de nível externo, que é ocultada pelo uso do respectivo nome para a variável local. O valor de `i` global agora está acessível apenas através do ponteiro `external_i` .

Veja também

Especificadores de classe de armazenamento para declarações de Internal-Level

Especificadores de classe de armazenamento com declarações de função

13/05/2021 • 2 minutes to read

Você pode usar o `static` ou o `extern` especificador de classe de armazenamento em declarações de função. As funções sempre têm tempos de vida globais.

Específico da Microsoft

As declarações de função no nível interno têm o mesmo significado que as declarações de função no nível externo. Isso significa que uma função é visível do ponto de declaração durante o restante da unidade de conversão mesmo que seja declarada no escopo local.

FINAL específico da Microsoft

As regras de visibilidade para funções variam ligeiramente das regras para as variáveis, como segue:

- Uma função declarada para ser `static` visível somente dentro do arquivo de origem no qual está definida. As funções no mesmo arquivo de origem podem chamar a `static` função, mas as funções em outros arquivos de origem não podem acessá-la diretamente pelo nome. Você pode declarar outra `static` função com o mesmo nome em um arquivo de origem diferente sem conflito.
- As funções declaradas como `extern` estão visíveis em todos os arquivos de origem no programa (a menos que você redeclare posteriormente tal função como `static`). Qualquer função pode chamar uma `extern` função.
- As declarações de função que omitem o especificador de classe de armazenamento são `extern` por padrão.

Específico da Microsoft

A Microsoft permite redefinição de um `extern` identificador como `static` .

FINAL específico da Microsoft

Veja também

Classes de armazenamento C

Especificadores de tipo C

13/05/2021 • 2 minutes to read

Os especificadores de tipo em declarações definem o tipo de uma declaração de função ou variável.

Sintaxe

tipo-especificador: `void` `char` `short` `int` `long` `float` `double` `signed` `unsigned` *struct-or-Union-do especificador* do *especificador* *-nome do typedef-Name*

Os `signed char` `signed int` `signed short int` tipos **int Long assinados** „ e com suas `unsigned` contrapartes e `enum` , são chamados de tipos *integrais* . Os `float` `double` especificadores„ e de `long double` tipo são chamados de tipos *flutuantes* ou *de ponto flutuante* . Você pode usar qualquer especificador do tipo integral ou de ponto flutuante em uma declaração de variável ou função. Se um *especificador de tipo* não for fornecido em uma declaração, ele será levado a ser `int` .

As palavras-chave opcionais `signed` e `unsigned` podem preceder ou seguir qualquer um dos tipos integrais, exceto `enum` , e também podem ser usadas sozinhas como especificadores de tipo; nesse caso, eles são compreendidos como `signed int` e `unsigned int` , respectivamente. Quando usado sozinho, a palavra-chave `int` é considerada `signed` . Quando usado sozinho, as palavras-chave `long` e `short` são compreendidas como **Long int** e `short int` .

Os tipos de enumeração são considerados tipos básicos. Os especificadores de tipo para tipos de enumeração são discutidos em Declarações de enumeração.

A palavra-chave `void` tem três usos: para especificar um tipo de retorno de função, para especificar uma lista de tipo de argumento para uma função que não usa argumentos e para especificar um ponteiro para um tipo não especificado. Você pode usar o `void` tipo para declarar funções que não retornam nenhum valor ou para declarar um ponteiro para um tipo não especificado. Consulte argumentos para obter informações sobre `void` quando ele aparece sozinho dentro dos parênteses após um nome de função.

Específico da Microsoft

A verificação de tipo agora está em conformidade com ANSI, o que significa que tipo `short` e tipo `int` são tipos distintos. Por exemplo, esta é uma redefinição do compilador do Microsoft C que foi aceita por versões anteriores do compilador.

```
int   myfunc();
short myfunc();
```

O exemplo a seguir também gerencia um aviso sobre a ação indireta a diferentes tipos:

```
int *pi;
short *ps;

ps = pi;  /* Now generates warning */
```

O compilador do Microsoft C também gerencia avisos para as diferenças no sinal. Por exemplo:

```
signed int *pi;
unsigned int *pu

pi = pu;  /* Now generates warning */
```

`void` Expressões de tipo são avaliadas para efeitos colaterais. Você não pode usar o valor (não existente) de uma expressão que tenha tipo `void` de qualquer forma, nem pode converter uma `void` expressão (por conversão implícita ou explícita) em qualquer tipo, exceto `void` . Se você usar uma expressão de qualquer outro tipo em um contexto em que uma `void` expressão é necessária, seu valor será Descartado.

Para estar em conformidade com a especificação ANSI, **void * *** não pode ser usado como **int * *** . Somente `void` * pode ser usado como um ponteiro para um tipo não especificado.

FINAL específico da Microsoft

Você pode criar especificadores de tipo adicionais com `typedef` declarações, conforme descrito em declarações de typedef. Consulte Armazenamento de tipos básicos para obter informações sobre o tamanho de cada tipo.

Veja também

Declarações e tipos

Especificadores de tipo de dados e equivalentes

13/05/2021 • 2 minutes to read

Essa documentação geralmente usa os formulários dos especificadores de tipo listados na tabela a seguir, em vez de formulários longos. Ele também pressupõe que o `char` tipo é assinado por padrão. Em toda esta documentação, `char` é equivalente a `signed char` .

Especificadores de tipo e equivalentes

ESPECIFICADOR DE TIPO	EQUIVALENTE(S)
`signed char` [1]	`char`
`signed int`	`signed` , `int`
`signed short int`	`short` , `signed short`
`signed long int`	`long` , `signed long`
`unsigned char`	—
`unsigned int`	`unsigned`
`unsigned short int`	`unsigned short`
`unsigned long int`	`unsigned long`
`float`	—
`long double` [2]	—

[1] quando você torna o `char` tipo não assinado por padrão (especificando a opção do `/J` compilador), não é possível abreviar `signed char` como `char` .

[2] em sistemas operacionais de 32 bits e 64 bits, o compilador do Microsoft C é mapeado `long double` para o tipo `double` .

específica da Microsoft

Você pode especificar a `/J` opção do compilador para alterar o `char` tipo padrão de `signed char` para `unsigned char` . Quando essa opção estiver em vigor, `char` significa o mesmo que `unsigned char` , e você deve usar a `signed` palavra-chave para declarar um valor de caractere assinado. Se um `char` valor for declarado explicitamente `signed` , a `/J` opção não o afetará, e o valor será estendido ao se expandir para um `int` tipo. O `char` tipo é estendido com zero quando ampliado para o `int` tipo.

FINAL específico da Microsoft

Veja também

Qualificadores de tipo

13/05/2021 • 3 minutes to read

Os qualificadores de tipo fornecem uma de duas propriedades a um identificador. O `const` qualificador de tipo declara um objeto que não poderá ser modificado. O `volatile` qualificador de tipo declara um item cujo valor pode ser alterado legitimamente por algo além do controle do programa no qual ele aparece, como um thread executando simultaneamente.

Os qualificadores de tipo„ `const` `restrict` e `volatile` , podem aparecer apenas uma vez em uma declaração. Qualificadores de tipo podem aparecer com qualquer especificador de tipo; no entanto, eles não podem aparecer após a primeira vírgula em uma declaração de vários itens. Por exemplo, as seguintes declarações são aceitáveis:

```
typedef volatile int VI;
const int ci;
```

Essas declarações não são legais:

```
typedef int *i, volatile *vi;
float f, const cf;
```

Os qualificadores de tipo são relevantes apenas ao acessar identificadores como l-values nas expressões. Consulte Expressões L-Value e R-Value para obter informações sobre l-values e expressões.

Sintaxe

`type-qualifier` :

`const`

`restrict`

`volatile`

`const` e `volatile`

As seguintes são as `const` declarações e legais `volatile` :

```
int const *p_ci;      // Pointer to constant int
int const (*p_ci);   // Pointer to constant int
int *const cp_i;     // Constant pointer to int
int (*const cp_i);   // Constant pointer to int
int volatile vint;     // Volatile integer
```

Se a especificação de um tipo de matriz incluir qualificadores de tipo, o elemento será qualificada, não o tipo de matriz. Se a especificação de tipo de função incluir qualificadores, o comportamento será indefinido. `volatile` e `const` não afetam o intervalo de valores ou Propriedades aritméticas do objeto.

- A `const` palavra-chave pode ser usada para modificar qualquer tipo fundamental ou agregado, ou um ponteiro para um objeto de qualquer tipo, ou um `typedef` . Se um item for declarado apenas com o `const` qualificador de tipo, seu tipo será usado como **const int**. Uma `const` variável pode ser inicializada ou pode ser colocada em uma região somente leitura do armazenamento. A `const` palavra-

chave é útil para declarar ponteiros `const`, pois isso requer que a função não altere o ponteiro de forma alguma.

- O compilador pressupõe que, em qualquer ponto do programa, uma `volatile` variável pode ser acessada por um processo desconhecido que usa ou modifica seu valor. Independentemente das otimizações especificadas na linha de comando, o código para cada atribuição ou referência de uma `volatile` variável deve ser gerado mesmo que pareça não ter efeito.

Se `volatile` for usado sozinho, `int` será assumido. O `volatile` especificador de tipo pode ser usado para fornecer acesso confiável a locais de memória especiais. Use `volatile` com objetos de dados que podem ser acessados ou alterados por manipuladores de sinais, executando programas simultaneamente ou por hardware especial, como registros de controle de e/s mapeados pela memória. Você pode declarar uma variável como `volatile` por seu tempo de vida, ou você pode converter uma única referência para ser `volatile`.

- Um item pode ser `const` e `volatile`, nesse caso, o item não pôde ser modificado legitimamente por seu próprio programa, mas pode ser modificado por algum processo assíncrono.

`restrict`

O `restrict` qualificador de tipo, introduzido em C99, pode ser aplicado a declarações de ponteiro. Ele qualifica o ponteiro, não o que aponta para.

`restrict` é uma dica de otimização para o compilador que nenhum outro ponteiro no escopo atual refere-se ao mesmo local de memória. Ou seja, somente o ponteiro ou um valor derivado dele (como o ponteiro + 1) é usado para acessar o objeto durante o tempo de vida do ponteiro. Isso ajuda o compilador a produzir código mais otimizado. C++ tem um mecanismo equivalente, `__restrict`

Tenha em mente que se trata `restrict` de um contrato entre você e o compilador. Se você fizer um alias com um ponteiro marcado com `restrict`, o resultado será indefinido.

Veja um exemplo que usa `restrict`:

```
void test(int* restrict first, int* restrict second, int* val)
{
    *first += *val;
    *second += *val;
}

int main()
{
    int i = 1, j = 2, k = 3;
    test(&i, &j, &k);

    return 0;
}

// Marking union members restrict tells the compiler that
// only z.x or z.y will be accessed in any scope, which allows
// the compiler to optimize access to the members.
union z
{
    int* restrict x;
    double* restrict y;
};
```

Confira também

`/std` (Especifique a versão padrão do idioma)

Declaradores e declarações de variáveis

13/05/2021 • 2 minutes to read

O restante desta seção descreve o formato e o significado das declarações para os tipos de variável resumidos nesta lista. Em particular, as seções restantes explicam como declarar:

TIPO DE VARIÁVEL	DESCRIÇÃO
Variáveis simples	Variáveis de valor único com tipo integral ou de ponto flutuante
matrizes	Variáveis compostas de uma coleção de elementos com o mesmo tipo
Ponteiros	Variáveis que apontam para outras variáveis e contêm locais de variáveis (na forma de endereços) em vez de valores
Variáveis de enumeração	Variáveis simples com tipo integral que mantêm um único valor de um conjunto de constantes de inteiro nomeadas
Estruturas	Variáveis compostas de uma coleção de valores que podem ter tipos diferentes
Uniões	Variáveis compostas de vários valores de tipos diferentes que ocupam o mesmo espaço de armazenamento

Um *Declarador* é a parte de uma declaração que especifica o nome a ser introduzido no programa. Ele pode incluir modificadores como `*` (ponteiro-para) e qualquer uma das palavras-chave da Convenção de chamada da Microsoft.

Específico da Microsoft

Neste Declarador,

```
__declspec(thread) char *var;
```

`char` é o especificador de tipo, `__declspec(thread)` e `*` são os modificadores, e `var` é o nome do identificador.

FINAL específico da Microsoft

Você usa declaradores para declarar matrizes de valores, ponteiros para valores e funções que retornam valores de um tipo especificado. Os declaradores aparecem nas declarações de matrizes e de ponteiros descritas posteriormente nesta seção.

Sintaxe

declarator:
pointer aceitar *direct-declarator*

direct-declarator:
identifier

`(` *declarator* `)`

direct-declarator `[` *constant-expression* aceitar `]`

direct-declarator `(` *parameter-type-list* `)`

direct-declarator `(` *identifier-list* aceitar `)`

pointer:

`*` *type-qualifier-list* aceitar

`*` *type-qualifier-list* aceitar *pointer*

type-qualifier-list:

type-qualifier

type-qualifier-list *type-qualifier*

> **NOTE**
> Consulte a sintaxe de *declaration* em visão geral de declarações ou Resumo da sintaxe da linguagem C para a sintaxe que faz referência a um *declarator* .

Quando um declarador consiste em um identificador não modificado, o item que está sendo declarado tem um tipo de base. Se um asterisco (`*`) aparecer à esquerda de um identificador, o tipo será modificado para um tipo de ponteiro. Se o identificador for seguido por colchetes (`[ ]`), o tipo será modificado para um tipo de matriz. Se os parênteses seguirem o identificador, o tipo será modificado para um tipo de função. Para obter mais informações sobre como interpretar a precedência em declarações, consulte interpretando declaradores mais complexos.

Cada declarador declara pelo menos um identificador. Um declarador deve incluir um especificador de tipo para ser uma declaração completa. O especificador de tipo fornece: o tipo dos elementos de um tipo de matriz, o tipo de objeto endereçado por um tipo de ponteiro ou o tipo de retorno de uma função.

As declarações de matrizes e de ponteiros são discutidas em mais detalhes posteriormente nesta seção. Os exemplos a seguir ilustram alguns formatos simples de declaradores:

```
int list[20]; // Declares an array of 20 int values named list
char *cp;     // Declares a pointer to a char value
double func( void ); // Declares a function named func, with no
                     // arguments, that returns a double value
int *aptr[10] // Declares an array of 10 pointers
```

Específico da Microsoft

O compilador do Microsoft C não limita o número de declaradores que podem modificar um tipo aritmético, de estrutura ou de União. O número é limitado somente pela memória disponível.

FINAL específico da Microsoft

Confira também

Declarações e tipos

Declarações de variável simples

13/05/2021 • 2 minutes to read

A declaração de uma variável simples, a forma mais simples de um declarador direto, especifica o nome e o tipo da variável. Ela também especifica a classe e o tipo de dados de armazenamento da variável.

Classes ou tipos de armazenamento (ou ambos) são necessários em declarações de variável. As variáveis sem tipo (como `var;`) geram avisos.

Sintaxe

Declarador:
  pointer$_{opt}$ *direct-declarator*

Declarador direto:
  ID

identificador:
  Não dígito
  identifier nondigit
  identifier digit

Para aritmética, estrutura, União, enumerações e tipos void, e para tipos representados por `typedef` nomes, declaradores simples podem ser usados em uma declaração, uma vez que o especificador de tipo fornece todas as informações de digitação. Tipos de ponteiro, matriz e função requerem declaradores mais complicados.

Você pode usar uma lista de identificadores separados por vírgulas (,) para especificar diversas variáveis na mesma declaração. Todas as variáveis definidas na declaração têm o mesmo tipo base. Por exemplo:

```
int x, y;        /* Declares two simple variables of type int */
int const z = 1; /* Declares a constant value of type int */
```

As variáveis `x` e `y` podem conter qualquer valor no conjunto definido pelo `int` tipo de uma implementação específica. O objeto simples `z` é inicializado com o valor 1 e não pode ser modificado.

Se a declaração de `z` fosse para uma variável estática não inicializada ou estava no escopo do arquivo, ela receberia um valor inicial de 0, e esse valor não seria modificável.

```
unsigned long reply, flag; /* Declares two variables
                              named reply and flag     */
```

Neste exemplo, as variáveis `reply` e `flag` têm `unsigned long` tipo e contêm valores integrais não assinados.

Veja também

Declaradores e declarações de variáveis

Declarações de enumeração C

13/05/2021 • 4 minutes to read

Uma enumeração consiste em um conjunto de constantes de número inteiro nomeadas. Uma declaração de tipo de enumeração fornece o nome da marca de enumeração (opcional). E define o conjunto de identificadores de inteiro nomeados (chamado de *conjunto de enumeração*, *constantes de enumerador*, *enumeradores*ou *Membros*). Uma variável do tipo de enumeração armazena um dos valores do conjunto de enumeração definido por esse tipo.

Variáveis do `enum` tipo podem ser usadas em expressões de indexação e como operandos de todos os operadores aritméticos e relacionais. As enumerações fornecem uma alternativa à política de pré-processador de `#define` com a vantagem de que os valores podem ser gerados para você e obedecer regras normais de escopo.

No ANSI C, as expressões que definem o valor de uma constante de enumerador sempre têm o `int` tipo. Isso significa que o armazenamento associado a uma variável de enumeração é o armazenamento necessário para um único `int` valor. Uma constante de enumeração ou um valor do tipo enumerado podem ser usados em qualquer lugar em que a linguagem C permita uma expressão de inteiro.

Sintaxe

`enum-specifier`:

`enum` *`identifier`* opt `{` aceitar *`enumerator-list`* `}`

`enum` *`identifier`*

`enumerator-list`:

`enumerator`

`enumerator-list` `,` *`enumerator`*

`enumerator`:

`enumeration-constant`

`enumeration-constant` `=` *`constant-expression`*

`enumeration-constant`:

`identifier`

Os nomes opcionais *`identifier`* do tipo de enumeração definido por *`enumerator-list`* . Esse identificador é geralmente chamado de a "marca" de enumeração especificada pela lista. Um especificador de tipo declara `identifier` ser a marca da enumeração especificada pelo não *`enumerator-list`* terminal, como visto aqui:

```
enum identifier
{
    // enumerator-list
}
```

O *`enumerator-list`* define os membros do conjunto de enumeração.

Se a declaração de uma marca estiver visível, as declarações posteriores que usam a marca, mas omitam, *`enumerator-list`* especificam o tipo enumerado declarado anteriormente. A marca deve se referir a um tipo definido da enumeração, e esse tipo de enumeração deve estar no escopo atual. Como o tipo de enumeração é definido em outro lugar, o *`enumerator-list`* não aparece nessa declaração. Declarações de tipos derivados de

enumerações e `typedef` declarações para tipos de enumeração podem usar a marca de enumeração antes que o tipo de enumeração seja definido.

Cada *`enumeration-constant`* *`enumerator-list`* um deles nomeia um valor do conjunto de enumeração. Por padrão, a primeira *`enumeration-constant`* é associada ao valor 0. O próximo *`enumeration-constant`* na lista é associado ao valor de (*`constant-expression`* + 1), a menos que você o associe explicitamente a outro valor. O nome de um *`enumeration-constant`* é equivalente ao seu valor.

Você pode usar *`enumeration-constant`* = *`constant-expression`* para substituir a sequência padrão de valores. Ou seja, se *`enumeration-constant`* = *`constant-expression`* aparecer no *`enumerator-list`* , o *`enumeration-constant`* será associado ao valor fornecido por *`constant-expression`* . O *`constant-expression`* deve ter o `int` tipo e pode ser negativo.

As regras a seguir se aplicam aos membros de um conjunto de enumerações:

- Um conjunto de enumerações pode conter valores duplicados de constantes. Por exemplo, você pode associar o valor 0 a dois identificadores diferentes, por exemplo, membros chamados `null` e `zero` , no mesmo conjunto.
- Os identificadores na lista de enumeração devem ser diferentes de outros identificadores no mesmo escopo com a mesma visibilidade. Isso inclui nomes de variáveis e identificadores comuns em outras listas de enumeração.
- As marcas de enumeração obedecem as regras normais de escopo. Elas devem ser distintas de outras marcas de enumerações, estruturas, e união com a mesma visibilidade.

Exemplos

Esses exemplos ilustram declarações de enumeração:

```
enum DAY            /* Defines an enumeration type    */
{
    saturday,       /* Names day and declares a       */
    sunday = 0,     /* variable named workday with    */
    monday,         /* that type                      */
    tuesday,
    wednesday,      /* wednesday is associated with 3 */
    thursday,
    friday
} workday;
```

O valor 0 é associado a `saturday` por padrão. O identificador `sunday` é explicitamente definido como 0. Os identificadores restantes recebem valores de 1 a 5 por padrão.

Neste exemplo, um valor do conjunto `DAY` é atribuído à variável `today` .

```
enum DAY today = wednesday;
```

O nome da constante de enumeração é usado para atribuir o valor. Como o tipo de enumeração `DAY` foi declarado anteriormente, somente a marca de enumeração `DAY` é necessária.

Para atribuir explicitamente um valor inteiro a uma variável de tipo de dados enumerado, use uma conversão de tipo:

```
workday = ( enum DAY ) ( day_value - 1 );
```

Essa conversão é recomendada em C, mas não é necessária.

```
enum BOOLEAN  /* Declares an enumeration data type called BOOLEAN */
{
    false,     /* false = 0, true = 1 */
    true
};

enum BOOLEAN end_flag, match_flag; /* Two variables of type BOOLEAN */
```

Essa declaração também pode ser especificada como

```
enum BOOLEAN { false, true } end_flag, match_flag;\
```

ou como

```
enum BOOLEAN { false, true } end_flag;
enum BOOLEAN match_flag;
```

Um exemplo que usa essas variáveis pode ter esta aparência:

```
if ( match_flag == false )
    {
     .
     .   /* statement */
     .
    }
    end_flag = true;
```

Os tipos de dados de enumerador sem nome também podem ser declarados. O nome do tipo de dados é omitido, mas as variáveis podem ser declaradas. A variável `response` é uma variável de tipo definida:

```
enum { yes, no } response;
```

Confira também

Enumerações

Declarações de estrutura

13/05/2021 • 5 minutes to read

Uma "declaração de estrutura" nomeia um tipo e especifica uma sequência de valores variáveis (chamados de "membros" ou "campos" da estrutura) que podem ter tipos diferentes. Um identificador opcional, chamado de "marca", fornece o nome do tipo de estrutura e pode ser usado em referências subsequentes ao tipo. Uma variável desse tipo de estrutura mantém a sequência inteira definida por esse tipo. As estruturas em C são semelhantes aos tipos conhecidos como "registros" em outras linguagens.

Sintaxe

especificador de struct-ou-Union:
identificador struct-or-Union { *struct-declaração-List* }
struct-or-union identifier

struct-or-union:
`struct`
`union`

struct-declaration-list:
declaração de struct
struct-declaration-list struct-declaration

declaração de struct:
specifier-qualifier-list struct-declarator-list ;

specifier-qualifier-list:
type-specifier specifier-qualifier-list$_{opt}$
type-qualifier specifier-qualifier-list$_{opt}$

struct-declarator-list:
struct - Declarador *struct-declarator-List* , *struct-declarator*

estrutura-declaradora:
Declarador
tipo de Declarador de especificador $_{opt}$: *expressão Constant*

A declaração de um tipo de estrutura não reserva espaço para uma estrutura. É apenas um modelo para declarações posteriores de variáveis da estrutura.

Um *identifier* (marca) definido anteriormente pode ser usado para fazer referência a um tipo de estrutura definido em outro lugar. Nesse caso, *struct-declaration-list* não poderá ser repetida enquanto a definição estiver visível. Declarações de ponteiros para estruturas e typedefs de tipos de estrutura podem usar a marca de estrutura antes que o tipo de estrutura seja definido. No entanto, a definição da estrutura deve ser encontrada antes de qualquer uso real do tamanho dos campos. Essa é uma definição incompleta do tipo e da marca de tipo. Para que essa definição fique completa, uma definição de tipo deve aparecer depois no mesmo escopo.

A *struct-declaration-list* especifica os tipos e nomes dos membros da estrutura. Um argumento *struct-declaration-list* contém uma ou mais declarações de variável ou de campo de bits.

Cada variável declarada em *struct-declaration-list* é definida como um membro do tipo de estrutura. As declarações de variáveis em *struct-declaration-list* têm o mesmo formato que outras declarações de variável abordadas nesta seção, com exceção de que as declarações não podem conter inicializadores nem

especificadores de classe de armazenamento. Os membros da estrutura podem ter quaisquer tipos de variáveis `void`, exceto tipo, um tipo incompleto ou um tipo de função.

Um membro não pode ser declarado para ter o tipo da estrutura em que aparece. Porém, um membro pode ser declarado como um ponteiro para o tipo de estrutura em que aparece, desde que o tipo tenha uma marca. Isso permite criar listas vinculadas de estruturas.

As estruturas seguem o mesmo escopo que outros identificadores. Os identificadores de estruturas devem ser distintos de outras marcas de estrutura, união e enumeração com a mesma visibilidade.

Cada *struct-declaration* em uma *struct-declaration-list* deve ser exclusivo na lista. No entanto, os nomes de identificadores em uma *struct-declaration-list* não precisam ser distintos de nomes de variáveis comuns ou de identificadores em outras listas de declarações de estrutura.

Também é possível acessar estruturas aninhadas como se fossem declaradas no nível do escopo do arquivo. Por exemplo, dada esta declaração:

```
struct a
{
    int x;
    struct b
    {
      int y;
    } var2;
} var1;
```

estas duas declarações são válidas:

```
struct a var3;
struct b var4;
```

Exemplos

Estes exemplos ilustram declarações de estruturas:

```
struct employee   /* Defines a structure variable named temp */
{
    char name[20];
    int id;
    long class;
} temp;
```

A estrutura `employee` tem três membros: `name`, `id` e `class`. O `name` membro é uma matriz de 20 elementos e `id` e `class` são membros simples com `int` and `long` Type, respectivamente. O identificador `employee` é o identificador da estrutura.

```
struct employee student, faculty, staff;
```

Esse exemplo define três variáveis de estrutura: `student`, `faculty` e `staff`. Cada estrutura tem a mesma lista de três membros. Os membros são declarados para ter o tipo de estrutura `employee`, definido no exemplo anterior.

```
struct           /* Defines an anonymous struct and a */
{                /* structure variable named complex  */
    float x, y;
} complex;
```

A `complex` estrutura tem dois membros com o `float` tipo `x` e `y` . O tipo de estrutura não tem marcas e, portanto, é não nomeado ou anônimo.

```
struct sample   /* Defines a structure named x */
{
    char c;
    float *pf;
    struct sample *next;
} x;
```

Os dois primeiros membros da estrutura são uma `char` variável e um ponteiro para um `float` valor. O terceiro membro, `next` , é declarado como um ponteiro para o tipo de estrutura que está sendo definido (`sample`).

As estruturas anônimas podem ser úteis quando a marca nomeada não é necessária. Esse é o caso quando uma declaração define todas as instâncias da estrutura. Por exemplo:

```
struct
{
    int x;
    int y;
} mystruct;
```

As estruturas inseridas são frequentemente anônimas.

```
struct somestruct
{
    struct    /* Anonymous structure */
    {
        int x, y;
    } point;
    int type;
} w;
```

Específico da Microsoft

O compilador permite uma matriz não dimensionada ou de tamanho zero como o último membro de uma estrutura. Isso poderá ser útil se uma matriz constante tiver tamanhos diferentes quando usada em situações variadas. A declaração de uma estrutura assim é semelhante a esta:

`struct` *identificador* { *conjunto de declarações tipo array-Name*[ ];};

As matrizes não dimensionadas só podem aparecer como o último membro de uma estrutura. As estruturas que contêm declarações de matrizes não dimensionadas podem ser aninhadas em outras estruturas, desde que nenhum membro adicional seja declarado em nenhuma das estruturas de inclusão. Matrizes dessas estruturas não são permitidas. O `sizeof` operador, quando aplicado a uma variável desse tipo ou ao próprio tipo, assume 0 para o tamanho da matriz.

Declarações de estruturas também podem ser especificadas sem um declarador quando são membros de outra estrutura ou união. Os nomes de campos são promovidos na estrutura de inclusão. Por exemplo, uma estrutura sem nome tem aparência semelhante a esta:

```
struct s
{
    float y;
    struct
    {
        int a, b, c;
    };
    char str[10];
} *p_s;
.
.
.
p_s->b = 100;  /* A reference to a field in the s structure */
```

Consulte Membros de estruturas e uniões para obter informações sobre referências de estrutura.

FINAL específico da Microsoft

Veja também

Declaradores e declarações de variáveis

Campos de bit C

13/05/2021 • 2 minutes to read

Além dos declaradores para membros de uma estrutura ou união, um declarador de estrutura também pode ser um número especificado de bits, chamado de "campo de bits". O comprimento é definido fora do declarador do nome de campo por dois pontos. Um campo de bits é interpretado como um tipo integral.

Sintaxe

estrutura-declaradora:
  Declarador
  tipo de Declarador de especificador $_{opt}$: *expressão Constant*

A *constant-expression* especifica a largura do campo em bits. O *especificador de tipo* para `declarator` deve ser `unsigned int` , `signed int` , ou `int` , e a *expressão de constante* deve ser um valor inteiro não negativo. Se o valor for zero, a declaração não tem nenhum `declarator` . As matrizes de campos de bits, os ponteiros para campos de bits, e as funções que retornam campos de bits não são permitidas. O `declarator` opcional nomeia o campo de bits. Os campos de bits só podem ser declarados como parte de uma estrutura. O operador address-of (`&`) não pode ser aplicado a componentes de campo de bits.

Não é possível fazer referência a campos de bits sem nome e seus conteúdos no tempo de execução são imprevisíveis. Podem ser usados como campos "fictícios", para fins de alinhamento. Um campo de bits sem nome cuja largura é especificada como 0 garante que o armazenamento do membro que o segue na *lista de declarações de struct* comece em um `int` limite.

Os campos de bits devem também devem ser longos o bastante para conter o padrão de bits. Por exemplo, essas duas instruções não são válidas:

```
short a:17;        /* Illegal! */
int long y:33;     /* Illegal! */
```

Este exemplo define uma matriz de estruturas bidimensional nomeada `screen` .

```
struct
{
    unsigned short icon : 8;
    unsigned short color : 4;
    unsigned short underline : 1;
    unsigned short blink : 1;
} screen[25][80];
```

A matriz contém 2.000 elementos. Cada elemento é uma estrutura individual que contém quatro membros de campo de bits: `icon` , `color` , `underline` e `blink` . O tamanho de cada estrutura são dois bytes.

Os campos de bits têm a mesma semântica do tipo inteiro. Isso significa que um campo de bits é usado em expressões exatamente da mesma forma que uma variável do mesmo tipo de base, independentemente de quantos bits houver no campo de bits.

Específico da Microsoft

Os campos de bits definidos como `int` são tratados como `signed` . Uma extensão da Microsoft para o padrão ANSI C permite `char` e `long` tipos (`signed` e `unsigned`) para campos de bits. Campos de bits não nomeados

com tipo base `long` , `short` ou `char` (`signed` ou `unsigned`) forçam o alinhamento a um limite apropriado para o tipo base.

Os campos de bit são alocados em um inteiro do bit menos significativo ao bit mais significativo. No código a seguir

```
struct mybitfields
{
    unsigned short a : 4;
    unsigned short b : 5;
    unsigned short c : 7;
} test;

int main( void );
{
    test.a = 2;
    test.b = 31;
    test.c = 0;
}
```

os bits seriam organizados como segue:

```
00000001 11110010
ccccccccb bbbbaaaa
```

Como a família de processadores 8086 armazena o byte inferior dos valores inteiros antes do byte superior, o inteiro `0x01F2` acima seria armazenado na memória física como `0xF2` seguido por `0x01` .

FINAL específico da Microsoft

Veja também

Declarações de estrutura

Armazenamento e alinhamento de estruturas

13/05/2021 • 2 minutes to read

Específico da Microsoft

Os membros da estrutura são armazenados em sequência na ordem em que são declarados: o primeiro membro tem o endereço de memória mais baixo e o último membro, o mais alto.

Cada objeto de dados tem um *requisito de alinhamento*. Para estruturas, o requisito é o maior dos respectivos membros. Cada objeto é alocado um *deslocamento* para que

deslocamento `%` *requisito* `==` de alinhamento 0

Os campos de bits adjacentes serão empacotados na mesma unidade de alocação de 1, 2, ou 4 bytes se os tipos integrais forem do mesmo tamanho e se o campo de bit seguinte se encaixar na unidade de alocação atual sem cruzar o limite imposto pelos requisitos comuns de alinhamento dos campos de bits.

Para conservar espaço ou ficar em conformidade com as estruturas de dados existentes, pode ser conveniente armazenar estruturas de forma mais ou menos compacta. A opção do compilador /Zp[*n*] e o #pragma pack controlam o modo como os dados da estrutura são "empacotados" na memória. Quando você usa a opção /Zp[*n*], em que *n* é 1, 2, 4, 8 ou 16, cada membro da estrutura depois do primeiro é armazenado em limites de bytes que são o requisito de alinhamento do campo ou o tamanho de empacotamento (*n*), o que for menor. Expressos como uma fórmula, os limites de bytes são

```
min( n, sizeof( item ) )
```

em que *n* é o tamanho de empacotamento expresso com a opção /Zp[*n*] e *item* é o membro da estrutura. O tamanho de empacotamento padrão é /Zp8.

Para usar o pragma `pack` para especificar um empacotamento que não seja aquele especificado na linha de comando para determinada estrutura, atribua o pragma `pack`, onde o tamanho de empacotamento é 1, 2, 4, 8 ou 16, antes da estrutura. Para restabelecer o empacotamento atribuído na linha de comando, especifique o pragma `pack` sem argumentos.

Os campos de bits assumem o tamanho padrão `long` para o compilador do Microsoft C. Os membros da estrutura são alinhados no tamanho do tipo ou no tamanho de /Zp[*n*], o que for menor. O tamanho padrão é 4.

FINAL específico da Microsoft

Veja também

Declarações de estrutura

Declarações de união

13/05/2021 • 2 minutes to read

Uma "declaração de união" especifica um conjunto de valores de variáveis e, opcionalmente, uma tag que nomeia a união. Os valores de variáveis são chamados de "membros" da união e podem ter tipos diferentes. Uniões são semelhantes a "registros variantes" em outras linguagens.

Sintaxe

especificador de struct-ou-Union:
identificador struct-or-Union **{** *struct-declaração-List* **}**
struct-or-union identificador

struct-or-union:
`struct`
`union`

struct-declaration-list:
declaração de struct
struct-declaration-list struct-declaration

O conteúdo da união é definido para ser

declaração de struct:
specifier-qualifier-list struct-declarator-list **;**

specifier-qualifier-list:
type-specifier specifier-qualifier-list$_{opt}$
type-qualifier specifier-qualifier-list$_{opt}$

struct-declarator-list:
estrutura-declaradora
struct-declarator-List **,** *struct-declarator*

Uma variável com o `union` tipo armazena um dos valores definidos por esse tipo. As mesmas regras controlam declarações da estrutura e de união. Uniões também podem ter campos de bits.

Os membros de uniões não podem ter tipo, tipo `void` ou função incompleto. Portanto, os membros não podem ser uma instância de união, mas podem ser ponteiros ao tipo de união que está sendo declarado.

Uma declaração de tipo de união é somente um modelo. A memória não é reservada até que a variável seja declarada.

> **NOTE**
> Se uma união de dois tipos é declarada e um valor é armazenado, mas a união é acessada com o outro tipo, os resultados são não confiáveis. Por exemplo, uma União de `float` e `int` é declarada. Um `float` valor é armazenado, mas o programa mais tarde acessa o valor como um `int` . Nessa situação, o valor dependeria do armazenamento interno de `float` valores. O valor inteiro não seria confiável.

Exemplos

Veja a seguir alguns exemplos de uniões:

```
union sign   /* A definition and a declaration */
{
    int svar;
    unsigned uvar;
} number;
```

Este exemplo define uma variável de união com tipo `sign` e declara uma variável nomeada `number` que tem dois membros: `svar`, um inteiro assinado, e `uvar`, um inteiro sem sinal. Esta declaração permite que o valor atual de `number` seja armazenado como um valor assinado ou não assinado. A marca associada a esse tipo de união é `sign`.

```
union               /* Defines a two-dimensional */
{                   /*  array named screen */
    struct
    {
      unsigned int icon : 8;
      unsigned color : 4;
    } window1;
    int screenval;
} screen[25][80];
```

A matriz `screen` contém 2.000 elementos. Cada elemento da matriz é uma união individual com dois membros: `window1` e `screenval`. O membro `window1` é uma estrutura com dois membros de campos de bits, `icon` e `color`. O `screenval` membro é um `int` . A qualquer momento, cada elemento Union mantém o `int` representado por `screenval` ou a estrutura representada por `window1` .

Específico da Microsoft

Uniões aninhadas podem ser declaradas anonimamente quando são membros de outra estrutura ou união. Este é um exemplo de uma união sem nome:

```
struct str
{
    int a, b;
    union            / * Unnamed union */
    {
      char c[4];
      long l;
      float f;
   };
   char c_array[10];
} my_str;
.
.
.
my_str.l == 0L;  /* A reference to a field in the my_str union */
```

Uniões frequentemente são aninhadas em uma estrutura que inclui um campo que fornece o tipo de dados contido na união de qualquer horário específico. Este é um exemplo de uma declaração para essa união:

```
struct x
{
    int type_tag;
    union
    {
      int x;
      float y;
    }
}
```

Consulte Membros de estruturas e uniões para obter informações sobre uniões de referência.

FINAL específico da Microsoft

Veja também

Declaradores e declarações de variáveis

Armazenamento de uniões

13/05/2021 • 2 minutes to read

O armazenamento associado a uma variável de união é o armazenamento necessário para o maior membro da união. Quando um membro menor é armazenado, a variável de união pode conter espaço de memória não utilizado. Todos os membros são armazenados no mesmo espaço de memória e começam no mesmo endereço. O valor armazenado é substituído sempre que um valor é atribuído a um membro diferente. Por exemplo:

```
union         /* Defines a union named x */
{
    char *a, b;
    float f[20];
} x;
```

Os membros da `x` União são, em ordem de sua declaração, um ponteiro para um `char` valor, um `char` valor e uma matriz de `float` valores. O armazenamento alocado para `x` é o armazenamento necessário para a matriz `f` de 20 elementos, pois `f` é o membro mais longo da união. Como nenhuma marca é associada à união, seu tipo é sem nome ou "anônimo".

Veja também

Declarações Union

Declarações de matriz

13/05/2021 • 2 minutes to read

Uma "declaração de matriz" nomeia a matriz e especifica o tipo dos respectivos elementos. Também pode definir o número de elementos na matriz. Uma variável com tipo de matriz é considerada um ponteiro para o tipo dos elementos da matriz.

Sintaxe

declaração:
  declaration-specifiers init-declarator-list$_{opt}$;

init-declarator-list:
  init-Declarador
  init-declarator-List , *init-declarator*

init-Declarador:
  Declarador
  Declarador = *inicializador*

Declarador:
  pointer$_{opt}$ *direct-declarator*

Declarador direto:/ * um Declarador de função */
  *Declarador direto***[****opt-expressão de constante*]

Como *constant-expression* é opcional, a sintaxe tem dois formatos:

- O primeiro formato define uma variável de matriz. O argumento *constant-expression* entre colchetes especifica o número de elementos na matriz. O *constant-expression*, se estiver presente, deve ter um tipo integral e um valor maior que zero. Cada elemento tem o tipo fornecido pelo *especificador de tipo*, que pode ser qualquer tipo, exceto `void` . Um elemento de matriz não pode ser um tipo de função.
- O segundo formato declara uma variável que foi definida em outro lugar. Ele omite o argumento *constant-expression* entre colchetes, mas não os colchetes. Você só poderá usar esse formato se tiver inicializado a matriz anteriormente, se a tiver declarado como um parâmetro ou se a tiver declarado como uma referência a uma matriz definida explicitamente em outro lugar no programa.

Nos dois formatos, *direct-declarator* nomeia a variável e pode modificar o tipo dela. Os colchetes ([ ]) depois de *direct-declarator* modificam o declarador para um tipo de matriz.

Qualificadores de tipo podem aparecer na declaração de um objeto de tipo de matriz, mas se aplicam aos elementos, e não à própria matriz.

Você pode declarar uma matriz de matrizes (uma matriz "multidimensional") colocando uma lista de expressões de constantes entre colchetes após o declarador de matriz, neste formato:

```
Declarador de tipo-especificador [ expressão constante ] [ expressão constante ] ...
```

Cada *constant-expression* entre colchetes define o número de elementos em uma determinada dimensão: as matrizes bidimensionais têm duas expressões entre colchetes, as matrizes tridimensionais têm três e assim por diante. Você poderá omitir a primeira expressão de constante se tiver inicializado a matriz, se a tiver declarado como um parâmetro ou se a tiver declarado como uma referência a uma matriz definida explicitamente em

outro lugar no programa.

É possível definir matrizes de ponteiros para diversos tipos de objetos usando declaradores complexos, conforme descrito em Interpretar declaradores mais complexos.

As matrizes são armazenadas por linha. Por exemplo, a seguinte matriz consiste em duas linhas com três colunas cada:

```
char A[2][3];
```

As três colunas da primeira linha são armazenadas primeiro, seguidas pelas três colunas da segunda linha. Isso significa que o último subscrito varia mais rapidamente.

Para fazer referência a um elemento individual de uma matriz, use uma expressão de subscrito, conforme descrito em Operadores pós-fixados.

Exemplos

Estes exemplos ilustram declarações de matrizes:

```
float matrix[10][15];
```

A matriz bidimensional chamada `matrix` tem 150 elementos, cada um com `float` tipo.

```
struct {
    float x, y;
} complex[100];
```

Essa é uma declaração de uma matriz de estruturas. Essa matriz tem 100 elementos; cada elemento é uma estrutura que contém dois membros.

```
extern char *name[];
```

Essa instrução declara o tipo e o nome de uma matriz de ponteiros para `char` . A definição real de `name` ocorre em outro lugar.

Específico da Microsoft

O tipo de inteiro necessário para manter o tamanho máximo de uma matriz é o tamanho de **size_t**. Definido no arquivo de cabeçalho STDDEF. H, **size_t** é um `unsigned int` com o intervalo 0x00000000 a 0x7CFFFFFF.

FINAL específico da Microsoft

Veja também

Declaradores e declarações de variáveis

Armazenamento de matrizes

13/05/2021 • 2 minutes to read

O armazenamento associado a um tipo de matriz é o armazenamento obrigatório para todos os seus elementos. Os elementos de uma matriz são armazenados em locais contíguos e de memória crescente, do primeiro elemento ao último.

Veja também

Declarações de matriz

Declarações de ponteiro

13/05/2021 • 4 minutes to read

Uma *declaração de ponteiro* nomeia uma variável de ponteiro e especifica o tipo de objeto para o qual a variável aponta. Uma variável declarada como um ponteiro contém um endereço de memória.

Sintaxe

Declarador:
  pointer$_{opt}$ *direct-declarator*

Declarador direto:
  ID
  (*Declarador* **)**
  Declarador direto **[** *opt-expressão de constante* **]**
  Declarador direto **(** *tipo de parâmetro-lista* **)**
  Declarador direto **(** *opção identificador-lista* **)**

ponteiro:
  ***** $_{opção}$ *de qualificador de tipo-lista*
  ***** *ponteiro* de *tipo-qualificador de lista de tipos*

type-qualifier-list:
  qualificador de tipo
  type-qualifier-list type-qualifier

O *type-specifier* informa o tipo de objeto, que pode ser qualquer tipo básico, de estrutura ou união. As variáveis de ponteiro também podem apontar para funções, matrizes e outros ponteiros. (Para obter informações sobre como declarar e interpretar tipos de ponteiro mais complexos, consulte Interpretar declaradores mais complexos.)

Ao fazer o *especificador de tipo* `void` , você pode atrasar a especificação do tipo ao qual o ponteiro se refere. Esse item é conhecido como "ponteiro para `void` " e é escrito como `void *` . Uma variável declarada como um ponteiro para *nulo* pode ser usada para apontar para um objeto de qualquer tipo. No entanto, para realizar a maioria das operações no ponteiro ou no objeto para o qual ele aponta, o tipo para o qual ele aponta deve ser explicitamente especificado para cada operação. (Variáveis do tipo `char` * e Type `void` * são compatíveis com atribuição sem uma conversão de tipo.) Essa conversão pode ser realizada com uma conversão de tipo (Confira conversões de conversão de tipo para obter mais informações).

O *qualificador de tipo* pode ser `const` ou ou `volatile` ambos. Eles especificam, respectivamente, que o ponteiro não pode ser modificado pelo próprio programa (`const`), ou que o ponteiro pode ser modificado legitimamente por algum processo além do controle do programa (`volatile`). (Consulte qualificadores de tipo para obter mais informações sobre `const` e `volatile` .)

O *declarator* atribui um nome à variável e pode incluir um modificador de tipo. Por exemplo, se o *declarator* representa uma matriz, o tipo do ponteiro é modificado para ser um ponteiro para uma matriz.

Você pode declarar um ponteiro para uma estrutura, união ou tipo de enumeração antes de definir a estrutura, a união ou o tipo de enumeração. Você declara o ponteiro com a marca de estrutura ou de união, conforme mostrado nos exemplos abaixo. Essas declarações são permitidas porque o compilador não precisa saber o tamanho da estrutura ou da união para alocar espaço para a variável de ponteiro.

Exemplos

Os exemplos a seguir ilustram as declarações de ponteiro.

```
char *message; /* Declares a pointer variable named message */
```

O ponteiro de *mensagem* aponta para uma variável com o `char` tipo.

```
int *pointers[10];  /* Declares an array of pointers */
```

A matriz de *ponteiros* tem 10 elementos; cada elemento é um ponteiro para uma variável com o `int` tipo.

```
int (*pointer)[10]; /* Declares a pointer to an array of 10 elements */
```

A variável *pointer* aponta para uma matriz com 10 elementos. Cada elemento nesta matriz tem `int` tipo.

```
int const *x;      /* Declares a pointer variable, x,
                      to a constant value */
```

O ponteiro *x* pode ser modificado para apontar para um `int` valor diferente, mas o valor para o qual ele aponta não pode ser modificado.

```
const int some_object = 5 ;
int other_object = 37;
int *const y = &fixed_object;
int volatile *const z = &some_object;
int *const volatile w = &some_object;
```

A variável *y* nessas declarações é declarada como um ponteiro constante para um `int` valor. O valor para o qual ele aponta pode ser modificado, mas o ponteiro em si deve sempre apontar para o mesmo local: o endereço de *fixed_object*. Da mesma forma, *z* é um ponteiro constante, mas também é declarado para apontar para um `int` cujo valor não pode ser modificado pelo programa. O especificador adicional `volatile` indica que, embora o valor da **const int** apontada por *z* não possa ser modificado pelo programa, ele pode ser modificado legitimamente por um processo em execução simultaneamente com o programa. A declaração de *w* especifica que o programa não pode alterar o valor para o qual está apontado e que o programa não pode modificar o ponteiro.

```
struct list *next, *previous; /* Uses the tag for list */
```

Este exemplo declara duas variáveis de ponteiro, *next* e *previous*, que apontam para a o tipo de estrutura *list*. Essa declaração pode aparecer antes da definição do tipo de estrutura *list* (consulte o próximo exemplo), desde que a definição de tipo *list* tenha a mesma visibilidade que a declaração.

```
struct list
{
    char *token;
    int count;
    struct list *next;
} line;
```

A variável *line* tem o tipo de estrutura chamado *list*. O tipo de estrutura de *lista* tem três membros: o primeiro

membro é um ponteiro para um `char` valor, o segundo é um `int` valor e o terceiro é um ponteiro para outra estrutura de *lista* .

```
struct id
{
    unsigned int id_no;
    struct name *pname;
} record;
```

O *registro* de variável tem a *ID* de tipo de estrutura. Observe que *pname* é declarado como um ponteiro para outro tipo de estrutura chamado *Name*. Essa declaração pode aparecer antes que o tipo *name* seja definido.

Veja também

Declaradores e declarações de variáveis

Armazenamento de endereços

13/05/2021 • 2 minutes to read

A quantidade de armazenamento necessária para um endereço e o significado do endereço dependem da implementação do compilador. Não há garantia de que ponteiros para tipos diferentes tenham o mesmo tamanho. Portanto, **sizeof(char *)** não é necessariamente igual a **sizeof(int *)**.

Específico da Microsoft

Para o compilador Microsoft C, **sizeof(char *)** é igual a **sizeof(int *)**.

FINAL específico da Microsoft

Veja também

Declarações de ponteiro

Ponteiros baseados (C)

13/05/2021 • 2 minutes to read

Específico da Microsoft

__based (referência do C++)

Para os compiladores C de 32 bits e de 64 bits da Microsoft, um ponteiro baseado é um deslocamento de 32 bits ou de 64 bits a partir de um base de ponteiro de 32 bits ou de 64 bits. O endereçamento baseado é útil para controlar as seções nas quais os objetos são alocados, diminuindo assim o tamanho do arquivo executável e aumentando a velocidade de execução. Em geral, o formato para especificar um ponteiro baseado é

> *Declarador* de *tipo* **__based (*base*)**

A variante "ponteiro de base" do endereçamento baseado permite especifica um ponteiro como base. O ponteiro baseado, portanto, é um deslocamento para a seção de memória que começa no início do ponteiro no qual ele é baseado. Ponteiros baseados em endereços de ponteiro são a única forma da `__based` palavra-chave válida em compilações de 32 bits e 64 bits. Nessas compilações, ele são deslocamentos de 32 bits ou de 64 bits a partir de uma base de 32 bits ou 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.

O exemplo a seguir mostra um ponteiro com base em um ponteiro.

```
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`.

FINAL específico da Microsoft

Veja também

Declaradores e declarações de variáveis

Declaradores abstratos C

13/05/2021 • 2 minutes to read

Um declarador abstrato é um declarador sem um identificador, consistindo em um ou mais modificadores de ponteiro, matriz ou função. O modificador de ponteiro (*****) sempre precede o identificador em um Declarador; a matriz (**[]**) e os modificadores de função (**()**) seguem o identificador. Sabendo disso, você pode determinar onde o identificador apareceria em um declarador abstrato e interpretar o declarador corretamente. Consulte Interpretar declaradores mais complexos para obter mais informações e exemplos de declaradores complexos. Geralmente, `typedef` pode ser usado para simplificar os declaradores. Consulte Declarações de Typedef.

Os declaradores abstratos podem ser complexos. Os parênteses em um declarador abstrato complexo especificam uma determinada interpretação, como fazem no caso dos declaradores complexos em declarações.

Estes exemplos ilustram declaradores abstratos:

```
int *         // The type name for a pointer to type int:

int *[3]      // An array of three pointers to int

int (*) [5]   // A pointer to an array of five int

int *()       // A function with no parameter specification
              // returning a pointer to int

// A pointer to a function taking no arguments and
// returning an int

int (*) ( void )

// An array of an unspecified number of constant pointers to
// functions each with one parameter that has type unsigned int
// and an unspecified number of other parameters returning an int

int (*const []) ( unsigned int, ... )
```

> **NOTE**
> O declarador abstrato que consiste em um conjunto de parênteses vazios, **( )**, não é permitido porque é ambíguo. É impossível determinar se o lugar do identificador implícito é dentro dos parênteses (nesse caso, ele é um tipo não modificado) ou antes dos parênteses (nesse caso, ele é um tipo de função).

Veja também

Declaradores e declarações de variáveis

Interpretando declaradores mais complexos

13/05/2021 • 4 minutes to read

Você pode incluir qualquer declarador entre parênteses para especificar uma interpretação de um "declarador complexo" específico. Um declarador complexo é um identificador qualificado por mais de um modificador de função, matriz ou ponteiro. Você pode aplicar várias combinações de modificadores de função, matriz e ponteiro a um único identificador. Geralmente, `typedef` pode ser usado para simplificar declarações. Consulte Declarações de Typedef.

Na interpretação de declaradores complexos, os colchetes e parênteses (ou seja, os modificadores à direita do identificador) têm precedência sobre os asteriscos (modificadores à esquerda do identificador). Os colchetes e os parênteses têm a mesma precedência e são associados da esquerda para a direita. Depois que o declarador for completamente interpretado, o especificador do tipo é aplicado como a última etapa. Usando parênteses, você pode substituir a ordem padrão de associação e forçar uma interpretação específica. No entanto, nunca use parênteses ao redor de um nome de identificador por si só. Ele pode ser interpretado erradamente como uma lista de parâmetros.

Uma maneira simples de interpretar declaradores complexos é lê-los de "dentro para fora" usando as quatro etapas a seguir:

1. Comece com o identificador e o procure pelos colchetes ou parênteses (se houver) diretamente à direita.
2. Interprete esses colchetes ou parênteses, então procure pelos asteriscos à esquerda.
3. Se você encontrar um parêntese direito em qualquer momento, retorne e aplique as regras 1 e 2 a todos os elementos entre parênteses.
4. Aplique o especificador do tipo.

```
char *( *(*var)() )[10];
 ^   ^  ^ ^ ^   ^    ^
 7   6  4 2 1   3    5
```

Neste exemplo, as etapas estão em ordem numérica e podem ser interpretadas da seguinte maneira:

1. O identificador `var` é declarado como
2. um ponteiro para
3. uma função que retorna
4. um ponteiro para
5. uma matriz de 10 elementos, que são
6. ponteiros para
7. `char` os.

Exemplos

Os exemplos a seguir ilustram outras instruções complexas e mostram como os parênteses podem afetar o significado de uma declaração.

```
int *var[5]; /* Array of pointers to int values */
```

O modificador da matriz tem prioridade maior que o modificador do ponteiro, portanto, `var` é declarado como matriz. O modificador de ponteiro se aplica ao tipo dos elementos da matriz; Portanto, os elementos de matriz são ponteiros para `int` valores.

```
int (*var)[5]; /* Pointer to array of int values */
```

Nesta declaração para `var` , parênteses fornecem o modificador de ponteiro com prioridade mais alta do que o modificador de matriz e `var` é declarado como um ponteiro para uma matriz de cinco `int` valores.

```
long *var( long, long ); /* Function returning pointer to long */
```

Os modificadores de função também têm prioridade mais alta do que os modificadores de ponteiro, portanto, essa declaração para `var` declara `var` ser uma função que retorna um ponteiro para um `long` valor. A função é declarada para usar dois `long` valores como argumentos.

```
long (*var)( long, long ); /* Pointer to function returning long */
```

Este exemplo é semelhante ao anterior. Os parênteses fornecem ao modificador de ponteiro maior prioridade do que o modificador de função e `var` é declarado como um ponteiro para uma função que retorna um `long` valor. Novamente, a função usa dois `long` argumentos.

```
struct both       /* Array of pointers to functions */
{                 /*   returning structures         */
    int a;
    char b;
} ( *var[5] )( struct both, struct both );
```

Os elementos de uma matriz não podem ser funções, mas esta declaração demonstra como declarar uma matriz dos ponteiros às funções. Neste exemplo, `var` é declarado como uma matriz de cinco ponteiros para as funções que retornam estruturas com dois membros. Os argumentos para as funções são declarados como duas estruturas com o mesmo tipo de estrutura, `both` . Observe que os parênteses que cercam `*var[5]` são necessários. Sem eles, a declaração é uma tentativa inválida de declarar uma matriz de funções, conforme mostrado abaixo:

```
/* ILLEGAL */
struct both *var[5](struct both, struct both);
```

A instrução a seguir declara uma matriz de ponteiros.

```
unsigned int *(* const *name[5][10] ) ( void );
```

A matriz `name` tem 50 elementos organizados em uma matriz multidimensional. Os elementos são ponteiros para um ponteiro que é uma constante. Esse ponteiro constante aponta para uma função que não tem parâmetros e retorna um ponteiro para um tipo sem assinatura.

Este exemplo a seguir é uma função que retorna um ponteiro para uma matriz de três `double` valores.

```
double ( *var( double (*)[3] ) )[3];
```

Nesta declaração, uma função retorna um ponteiro para uma matriz, uma vez que as funções que retornam matrizes são inválidas. Aqui `var` está declarado como uma função que retorna um ponteiro para uma matriz de três `double` valores. A função `var` usa um argumento. O argumento, como o valor de retorno, é um ponteiro para uma matriz de três `double` valores. O tipo de argumento é determinado por um *abstract-declarator* complexo. Os parênteses em volta do asterisco no tipo de argumento são obrigatórios; sem eles, o tipo de argumento seria uma matriz de três ponteiros para `double` valores. Para ver uma discussão e exemplos de declaradores abstratos, consulte Declaradores abstratos.

```
union sign         /* Array of arrays of pointers */
{                  /* to pointers to unions       */
     int x;
     unsigned y;
} **var[5][5];
```

Como mostra o exemplo acima, um ponteiro pode apontar para outro ponteiro e uma matriz pode conter matrizes como elementos. Aqui, `var` é uma matriz de cinco elementos. Cada elemento é uma matriz de cinco elementos de ponteiros para ponteiros para uniões com dois membros.

```
union sign *(*var[5])[5]; /* Array of pointers to arrays
                             of pointers to unions        */
```

Este exemplo mostra como a colocação de parênteses altera o significado da declaração. Neste exemplo, `var` é uma matriz de cinco elementos de ponteiros para as matrizes de cinco elementos de ponteiros para uniões. Para obter exemplos de como usar o `typedef` para evitar declarações complexas, consulte declarações de typedef.

Veja também

Declarações e tipos

Inicialização

13/05/2021 • 2 minutes to read

Um "inicializador" é um valor ou uma sequência de valores a ser atribuído à variável sendo declarada. Você pode definir uma variável como um valor inicial aplicando um inicializador ao declarador na declaração variável. O valor ou valores do inicializador são atribuídos à variável.

As seções a seguir descrevem como inicializar variáveis dos tipos escalares, agregadores e cadeia de caracteres. Os "tipos escalares" incluem todos os tipos aritméticos, além dos ponteiros. Os "tipos de agregação" incluem matrizes, estruturas e uniões.

Veja também

Declarações e tipos

Inicializando tipos escalares

13/05/2021 • 3 minutes to read

Ao inicializar tipos escalares, o valor de `assignment-expression` é atribuído à variável. As regras de conversão para a atribuição se aplicam. (Consulte Conversões de tipos para obter informações sobre regras de conversão.)

Sintaxe

`declaration`:
`declaration-specifiers` ** `init-declarator-list` aceitar `;`

`declaration-specifiers`:
`storage-class-specifier` ** `declaration-specifiers` aceitar
`type-specifier` ** `declaration-specifiers` aceitar
`type-qualifier` ** `declaration-specifiers` aceitar

`init-declarator-list`:
`init-declarator`
`init-declarator-list` `,` `init-declarator`

`init-declarator`:
`declarator`
`declarator` `=` `initializer` /* Para inicialização escalar*/

`initializer`:
`assignment-expression`

Você pode inicializar variáveis de qualquer tipo, desde que obedeça às seguintes regras:

- Variáveis declaradas no nível de escopo de arquivo podem ser inicializadas. Se você não inicializar uma variável explicitamente no nível externo, ela será inicializada como 0 por padrão.
- Uma expressão constante pode ser usada para inicializar qualquer variável global declarada com o `static` `storage-class-specifier` . Variáveis declaradas para serem `static` inicializadas quando a execução do programa é iniciada. Se você não inicializar explicitamente uma variável global `static` , ela será inicializada como 0 por padrão e todos os membros que tiverem o tipo de ponteiro receberão um ponteiro nulo.
- Variáveis declaradas com o `auto` `register` especificador de classe de armazenamento ou são inicializadas cada vez que o controle de execução passa para o bloco no qual elas são declaradas. Se você omitir um inicializador da declaração de `auto` uma `register` variável ou, o valor inicial da variável será indefinido. Para valores automáticos e de registro, o inicializador não está restrito a ser uma constante; ele pode ser qualquer expressão que envolva valores definidos anteriormente, inclusive chamadas de função.
- Os valores iniciais para declarações de variáveis externas e para todas as `static` variáveis, sejam elas externas ou internas, devem ser expressões constantes. (Para obter mais informações, consulte expressões constantes.) Como o endereço de qualquer variável estática ou declarada externamente é constante, ele pode ser usado para inicializar uma variável de ponteiro declarada internamente `static` . No entanto, o endereço de uma `auto` variável não pode ser usado como um inicializador estático porque pode ser diferente para cada execução do bloco. Você pode usar valores constantes ou variáveis para inicializar `auto` e `register` variáveis.

- Se a declaração de um identificador tem o escopo do bloco e o identificador tem vinculação externa, a declaração não pode ter uma inicialização.

Exemplos

Os exemplos a seguir ilustram inicializações:

```
int x = 10;
```

A variável de inteiro `x` é inicializada como a expressão de constante `10`.

```
register int *px = 0;
```

O ponteiro `px` é inicializado como 0, gerando um ponteiro "nulo".

```
const int c = (3 * 1024);
```

Este exemplo usa uma expressão constante `(3 * 1024)` para inicializar `c` um valor constante que não pode ser modificado devido à `const` palavra-chave.

```
int *b = &x;
```

Essa instrução inicializa o ponteiro `b` com o endereço de outra variável, `x`.

```
int *const a = &z;
```

O ponteiro `a` é inicializado com o endereço de uma variável nomeada `z`. No entanto, como ele é especificado para ser um `const`, a variável `a` só pode ser inicializada, nunca modificada. Ela sempre aponta para o mesmo local.

```
int GLOBAL ;

int function( void )
{
    int LOCAL ;
    static int *lp = &LOCAL;   /* Illegal initialization */
    static int *gp = &GLOBAL;  /* Legal initialization   */
    register int *rp = &LOCAL; /* Legal initialization   */
}
```

A variável global `GLOBAL` é declarada no nível externo e, por isso, tem tempo de vida global. A variável local `LOCAL` tem a `auto` classe de armazenamento e só tem um endereço durante a execução da função na qual ela é declarada. Portanto, `static` não é permitido tentar inicializar a variável `lp` de ponteiro com o endereço de `LOCAL`. A `static` variável `gp` de ponteiro pode ser inicializada para o endereço de `GLOBAL` porque esse endereço é sempre o mesmo. Da mesma forma, `*rp` pode ser inicializado porque `rp` é uma variável local e pode ter um inicializador não constante. Cada vez que se entra no bloco, `LOCAL` tem um novo endereço, que é então atribuído a `rp`.

Veja também

Initialization

Inicializando tipos agregados

13/05/2021 • 6 minutes to read

Um tipo de *agregação* é um tipo de estrutura, união ou matriz. Se um tipo de agregação contém membros de tipos de agregação, as regras de inicialização são aplicadas recursivamente.

Sintaxe

inicializador:
  { *inicializador-lista* **}** /* para inicialização de agregação */
  { *inicializador-lista* **,}**

initializer-list:
  initializer
  inicializador-lista **,** *inicializador*

A *initializer-list* é uma lista de inicializadores separados por vírgulas. Cada inicializador na lista é uma expressão constante ou uma lista de inicializadores. Em virtude disso, as listas de inicializadores podem ser aninhadas. Esse formato é útil para inicializar membros de agregações de um tipo de agregação, como mostrado nos exemplos desta seção. No entanto, se o inicializador de um identificador automático for uma expressão única, ele não precisa ser uma expressão constante, apenas ter o tipo apropriado para a atribuição ao identificador.

Para cada lista de inicializadores, os valores das expressões constantes são atribuídos, em ordem, aos membros correspondentes da variável de agregação.

Se *initializer-list* tiver menos valores do que um tipo de agregação, os membros ou os elementos restantes do tipo de agregação serão inicializados como 0. O valor inicial de um identificador automático não inicializado explicitamente será indefinido. Se *initializer-list* tiver mais valores do que um tipo de agregação, ocorrerá um erro. Essas regras se aplicam a cada lista de inicializadores inserida, bem como à agregação como um todo.

O inicializador de uma estrutura é uma expressão do mesmo tipo ou uma lista de inicializadores para seus membros incluídos entre chaves ({ }). Os membros sem nome de campo de bits não são inicializados.

Quando uma união é inicializada, *initializer-list* deve ser uma expressão constante única. O valor da expressão constante é atribuído ao primeiro membro de união.

Se o tamanho de uma matriz for desconhecido, o número de inicializadores determina o tamanho da matriz, e seu tipo se torna completo. Não há nenhuma maneira de especificar a repetição de um inicializador em C, ou de inicializar um elemento no meio de uma matriz sem fornecer também todos os valores acima. Se você precisar dessa operação em seu programa, escreva a rotina na linguagem Assembly.

Observe que o número de inicializadores pode definir o tamanho da matriz:

```
int x[ ] = { 0, 1, 2 }
```

No entanto, se você especificar o tamanho e fornecer o número incorreto de inicializadores, o compilador gerará um erro.

Específico da Microsoft

O tamanho máximo para uma matriz é definido por **size_t**. Definido no arquivo de cabeçalho STDDEF. H, **size_t** é um `unsigned int` com o intervalo 0x00000000 a 0x7CFFFFFF.

FINAL específico da Microsoft

Exemplos

Este exemplo mostra inicializadores para uma matriz.

```
int P[4][3] =
{
    { 1, 1, 1 },
    { 2, 2, 2 },
    { 3, 3, 3,},
    { 4, 4, 4,},
};
```

Essa instrução declara `P` como uma matriz quatro-por-três e inicializa os elementos da primeira linha como 1, os elementos da sua segunda linha como 2, e assim por diante até a quarta linha. Observe que a lista de inicializadores para a terceira e a quarta linhas contém vírgulas depois da última expressão constante. A última lista de inicializadores (`{4, 4, 4,},`) também é seguida por uma vírgula. Essas vírgulas adicionais são permitidas, mas não são obrigatórias. Somente vírgulas que separam expressões constantes e listas de inicializadores são necessárias.

Se um membro de agregação não tiver qualquer lista de inicializadores inserida, os valores serão simplesmente atribuídos, em ordem, a cada membro da subagregação. Portanto, a inicialização no exemplo anterior equivale ao seguinte:

```
int P[4][3] =
{
   1, 1, 1, 2, 2, 2, 3, 3, 3, 4, 4, 4
};
```

As chaves também podem delimitar inicializadores individuais na lista e ajudam a esclarecer o exemplo anterior.

Quando você inicializar uma variável de agregação, deve ter cuidado para usar corretamente as chaves e as listas de inicializadores. O exemplo a seguir ilustra com mais detalhes a interpretação das chaves pelo compilador:

```
typedef struct
{
    int n1, n2, n3;
} triplet;

triplet nlist[2][3] =
{
    { {  1, 2, 3 }, {  4, 5, 6 }, {  7, 8, 9 } },  /* Row 1 */
    { { 10,11,12 }, { 13,14,15 }, { 16,17,18 } }   /* Row 2 */
};
```

Neste exemplo, `nlist` é declarado como uma matriz de estruturas 2-por-3, cada estrutura com três membros. A linha 1 da inicialização atribui valores à primeira linha de `nlist` , como segue:

1. A primeira chave à esquerda na linha 1 sinaliza ao compilador que a inicialização do primeiro membro da agregação de `nlist` (isto é, `nlist[0]`) foi iniciada.

2. A segunda chave à esquerda indica a inicialização do primeiro membro da agregação de `nlist[0]` (isto é, a estrutura em `nlist[0][0]`) foi iniciada.

3. A primeira chave à direita termina a inicialização da estrutura `nlist[0][0]` ; a chave à esquerda a seguir

inicia a inicialização de `nlist[0][1]`.

4. O processo continua até o final da linha, onde a chave de fechamento à direita encerra a inicialização de `nlist[0]`.

A linha 2 atribui valores à segunda linha de `nlist` de forma semelhante. Observe que os conjuntos de chaves externos que delimitam os inicializadores nas linhas 1 e 2 são obrigatórios. A construção a seguir, que omite as chaves externas, provocará um erro:

```
triplet nlist[2][3] =  /* THIS CAUSES AN ERROR */
{
     {  1, 2, 3 },{  4, 5, 6 },{  7, 8, 9 },   /* Line 1 */
     { 10,11,12 },{ 13,14,15 },{ 16,17,18 }    /* Line 2 */
};
```

Nessa construção, a primeira chave deixada na linha 1 inicia a inicialização de `nlist[0]`, que é uma matriz de três estruturas. Os valores 1, 2 e 3, são atribuídos aos três membros da primeira estrutura. Quando a chave à direita seguinte é encontrada (após o valor 3), a inicialização de `nlist[0]` está concluída, e as duas estruturas restantes na matriz de três estruturas são iniciadas automaticamente como 0. Da mesma forma, `{ 4,5,6 }` inicializa a primeira estrutura na segunda linha de `nlist`. As duas estruturas de `nlist[1]` restantes são definidas como 0. Quando o compilador localizar a lista de inicializadores seguinte (`{ 7,8,9 }`), ele tentará inicializar `nlist[2]`. Como a `nlist` tem apenas duas linhas, essa tentativa causará um erro.

Neste próximo exemplo, os três `int` membros de `x` são inicializados como 1, 2 e 3, respectivamente.

```
struct list
{
    int i, j, k;
    float m[2][3];
} x = {
        1,
        2,
        3,
       {4.0, 4.0, 4.0}
      };
```

Na estrutura `list` acima, os três elementos na primeira linha de `m` são inicializados em 4.0; os elementos da linha restante de `m` são inicializados em 0.0 por padrão.

```
union
{
    char x[2][3];
    int i, j, k;
} y = { {
            {'1'},
            {'4'}
        }
      };
```

A variável de união `y`, neste exemplo, é inicializada. O primeiro elemento da união é uma matriz, assim, o inicializador é um inicializador de agregação. A lista de inicializadores `{'1'}` atribui valores para a primeira linha da matriz. Como apenas um valor aparece na lista, o elemento na primeira coluna é inicializado com o caractere `1`, e os dois elementos restantes na linha são inicializados com o valor 0 por padrão. Da mesma forma, o primeiro elemento da segunda linha de `x` é inicializado com o caractere `4`, e os dois elementos restantes na linha são inicializados com o valor 0.

Veja também

Initialization

Inicializando cadeias de caracteres

13/05/2021 • 2 minutes to read

Você pode inicializar uma matriz de caracteres (ou caracteres largos) com uma literal de cadeia de caracteres (ou literal de cadeia de caracteres largos). Por exemplo:

```
char code[ ] = "abc";
```

inicializa `code` como uma matriz de caracteres de quatro elementos. O quarto elemento é caractere nulo, que termina todos os literais de cadeia de caracteres.

Uma lista de identificadores só pode ter o mesmo número de identificadores a serem inicializados. Se você especificar um tamanho da matriz menor do que a cadeia de caracteres; os caracteres adicionais serão ignorados. Por exemplo, a seguinte declaração inicializa `code` como uma matriz de caracteres de três elementos:

```
char code[3] = "abcd";
```

Somente os três primeiros caracteres do inicializador são atribuídos a `code`. O caractere `d` e o caractere de terminação nula da cadeia de caracteres são descartados. Observe que isso cria uma cadeia de caracteres não terminada (ou seja, sem um valor 0 para marcar seu fim) e gera uma mensagem de diagnóstico que indica essa condição.

Esta declaração

```
char s[] = "abc", t[3] = "abc";
```

é idêntica a

```
char s[]  = {'a', 'b', 'c', '\0'},
     t[3] = {'a', 'b', 'c' };
```

Se a cadeia de caracteres for menor que o tamanho de matriz especificado, os elementos restantes da matriz serão inicializados como 0.

Específico da Microsoft

No Microsoft C, os literais de cadeia de caracteres podem ter até 2048 bytes de comprimento.

FINAL específico da Microsoft

Veja também

Initialization

Armazenamento de tipos básicos

13/05/2021 • 2 minutes to read

A tabela a seguir resume o armazenamento associado a cada tipo básico.

Tamanhos de tipos fundamentais

TIPO	ARMAZENAMENTO
`char` , `unsigned char` , `signed char`	1 byte
`short` , `unsigned short`	2 bytes
`int` , `unsigned int`	4 bytes
`long` , `unsigned long`	4 bytes
`long long` , `unsigned long long`	8 bytes
`float`	4 bytes
`double`	8 bytes
`long double`	8 bytes

Os tipos de dados C se enquadram nas categorias gerais. Os *tipos integrais* incluem,,, `int` `char` `short` `long` e `long long` . Esses tipos podem ser qualificados com `signed` ou e `unsigned` , e `unsigned` por si só podem ser usados como abreviados para `unsigned int` . Os tipos de enumeração (`enum`) também são tratados como tipos integrais para a maioria das finalidades. Os *tipos flutuantes* incluem `float` , `double` e `long double` . Os *tipos aritméticos* incluem todos os tipos flutuantes e integrais.

Veja também

Declarações e tipos

Tipo char

13/05/2021 • 2 minutes to read

O `char` tipo é usado para armazenar o valor inteiro de um membro do conjunto de caracteres representável. Esse valor inteiro é o código ASCII que corresponde ao caractere especificado.

Específico da Microsoft

Os valores de caractere do tipo `unsigned char` têm um intervalo de 0 a 0xFF hexadecimal. Um `signed char` tem um intervalo de 0x80 a 0x7f. Esses intervalos podem ser traduzidos em de 0 a 255 decimal e -128 a +127 decimal, respectivamente. A opção/J Compiler altera o padrão de `signed` para `unsigned` .

FINAL específico da Microsoft

Veja também

Armazenamento de tipos básicos

Tipo int

13/05/2021 • 2 minutes to read

O tamanho de um `signed int` `unsigned int` item ou é o tamanho padrão de um inteiro em um determinado computador. Por exemplo, em sistemas operacionais de 16 bits, o `int` tipo geralmente é 16 bits ou 2 bytes. Em sistemas operacionais de 32 bits, o `int` tipo geralmente é de 32 bits ou 4 bytes. Portanto, o `int` tipo é equivalente ao `short int` `long int` tipo ou, e o `unsigned int` tipo é equivalente ao `unsigned short` ou ao `unsigned long` tipo, dependendo do ambiente de destino. Os `int` tipos todos representam valores assinados, a menos que especificado de outra forma.

Os especificadores de tipo `int` e `unsigned int` (ou simplesmente `unsigned`) definem determinados recursos da linguagem C (por exemplo, o `enum` tipo). Nesses casos, as definições de `int` e `unsigned int` para uma implementação específica determinam o armazenamento real.

Específico da Microsoft

Os inteiros com sinal são representados no formato de dois complementos. O bit mais significativo contém o sinal: 1 para o negativo, 0 para o sinal positivo e zero. O intervalo de valores é fornecido em limites de inteiro C e C++, que é obtido dos limites. Arquivo de cabeçalho H.

FINAL específico da Microsoft

> **NOTE**
> Os `int` `unsigned int` especificadores e de tipo são amplamente usados em programas C porque permitem que um computador específico manipule valores inteiros da maneira mais eficiente para esse computador. No entanto, como os tamanhos `int` dos `unsigned int` tipos e variam, os programas que dependem de um `int` tamanho específico podem não ser portáteis para outros computadores. Para tornar os programas mais portáteis, você pode usar expressões com o `sizeof` operador (conforme discutido no `sizeof` operador) em vez de tamanhos de dados embutidos em código.

Veja também

Armazenamento de tipos básicos

Tipos de inteiro dimensionados C

13/05/2021 • 2 minutes to read

Específico da Microsoft

Suporte dos recursos do Microsoft C 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 o tamanho, em bits, da variável de inteiro. O valor de *n* pode ser 8, 16, 32 ou 64. O exemplo a seguir declara uma variável de cada um dos quatro 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 três primeiros tipos de inteiros de tamanho são sinônimos para os tipos ANSI que têm o mesmo tamanho. Eles são úteis para escrever código portátil 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` , `__int32` é sinônimo de tipo `int` e `__int64` é sinônimo de tipo `long long` .

FINAL específico da Microsoft

Veja também

Armazenamento de tipos básicos

Tipo float

13/05/2021 • 4 minutes to read

Os números de ponto flutuante usam o formato IEEE (Instituto de Engenheiros Eletricistas e Eletrônicos). Os valores de precisão simples com tipo float têm 4 bytes, que consistem em um bit de sinal, um expoente binário de 8 bits no formato de 127 em excesso e uma mantissa de 23 bits. A mantissa representa um número entre 1,0 e 2,0. Como o bit de ordem superior da mantissa é sempre 1, ele não é armazenado no número. Essa representação fornece um intervalo de aproximadamente 3,4E-38 a 3,4E+38 para o tipo float.

Você pode declarar variáveis como float ou double, dependendo das necessidades de seu aplicativo. As principais diferenças entre os dois tipos são a significação que podem representar, o armazenamento que exigem e o intervalo que ocupam. A tabela a seguir mostra a relação entre a significação e os requisitos de armazenamento.

Tipos de ponto flutuante

TIPO	DÍGITOS SIGNIFICATIVOS	NÚMERO DE BYTES
FLOAT	6 - 7	4
double	15 - 16	8

As variáveis de ponto flutuante são representadas por uma mantissa, que contém o valor do número, e um expoente, que contém a ordem de grandeza do número.

A tabela a seguir mostra o número de bits alocados à mantissa e ao expoente para cada tipo de ponto flutuante. O bit mais significativo de qualquer float ou double é sempre o bit de sinal. Se ele for 1, o número será considerado negativo; caso contrário, será considerado um número positivo.

Comprimentos de expoentes e de mantissas

TIPO	COMPRIMENTO DO EXPOENTE	COMPRIMENTO DA MANTISSA
FLOAT	8 bits	23 bits
double	11 bits	52 bits

Como são armazenados em um formato sem sinal, os expoentes são deslocados pela metade do valor possível. Para o tipo float, o deslocamento é 127; para o tipo double, é 1023. Você pode calcular o valor real do expoente subtraindo o valor do deslocamento do valor do expoente.

A mantissa é armazenada como uma fração binária maior ou igual a 1 e menor que 2. Para os tipos float e double, existe um 1 à esquerda implícito na mantissa na posição de bit mais significativa; assim, na verdade, as mantissas têm 24 e 53 bits de comprimento, respectivamente, embora o bit mais significativo nunca seja armazenado na memória.

Em vez do método de armazenamento descrito acima, o pacote de ponto flutuante pode armazenar números de ponto flutuante binários como números desnormalizados. "Números desnormalizados" são números de ponto flutuante diferentes de zero com valores de expoente reservados nos quais o bit mais significativo da mantissa é 0. Usando o formato desnormalizado, o intervalo de um número de ponto flutuante pode ser estendido, perdendo em precisão. Você não pode controlar se um número de ponto flutuante é representado no formato normalizado ou desnormalizado; o pacote de ponto flutuante determina a representação. O pacote de ponto

flutuante nunca usa um formato desnormalizado, a menos que o expoente se torne menor do que o valor mínimo que pode ser representado em um formato normalizado.

A tabela a seguir mostra os valores mínimo e máximo que você pode armazenar em variáveis de cada tipo de ponto flutuante. Os valores listados na tabela só se aplicam aos números de ponto flutuante normalizados; os números de ponto flutuante desnormalizados têm um valor mínimo menor. Observe que os números retidos nos registros 80 *x* 87 são sempre representados no formato normalizado de 80 bits; os números só podem ser representados no formato desnormalizado quando armazenados em variáveis de ponto flutuante de 32 bits ou de 64 bits (variáveis do tipo float e do tipo long).

Intervalo de tipos de ponto flutuante

TIPO	VALOR MÍNIMO	VALOR MÁXIMO
FLOAT	1.175494351 E - 38	3,402823466 E + 38
double	2.2250738585072014 E - 308	1,7976931348623158 E + 308

Se a precisão preocupa menos que o armazenamento, considere usar o tipo float para as variáveis de ponto flutuante. Ao contrário, se a precisão é o critério mais importante, use o tipo double.

As variáveis de ponto flutuante podem ser promovidas a um tipo de maior significação (do tipo float para o tipo double). Muitas vezes, a promoção ocorre quando você executa uma aritmética em variáveis de ponto flutuante. Essa aritmética é sempre realizada no mesmo grau de precisão que a variável com o grau de precisão mais alto. Por exemplo, considere as seguintes declarações de tipo:

```
float f_short;
double f_long;
long double f_longer;

f_short = f_short * f_long;
```

No exemplo acima, a variável `f_short` é promovida para o tipo double e multiplicada por `f_long`; o resultado é arredondado para o tipo float antes de ser atribuído a `f_short`.

No exemplo a seguir (que usa as declarações do exemplo anterior), a aritmética é realizada com precisão de float (32 bits) nas variáveis; o resultado é promovido para o tipo double:

```
f_longer = f_short * f_short;
```

Veja também

Armazenamento de tipos básicos

Tipo duplo

13/05/2021 • 2 minutes to read

Os valores de precisão double com tipo double têm 8 bytes. O formato é semelhante ao formato de float, exceto que tem um expoente excess-1023 de 11 bits e uma mantissa de 52 bits, mais 1 bit implícito de ordem alta. Esse formato fornece um intervalo de aproximadamente 1.7E-308 a 1.7E+308 para o tipo double.

Específico da Microsoft

O tipo double contém 64 bits: 1 para o sinal, 11 para o expoente e 52 para a mantissa. O intervalo é +/-1.7E308 com pelo menos 15 dígitos de precisão.

FINAL específico da Microsoft

Veja também

Armazenamento de tipos básicos

Tipo long double

13/05/2021 • 2 minutes to read

O `long double` tipo é idêntico ao tipo Double .

Veja também

Armazenamento de tipos básicos

Tipos incompletos

13/05/2021 • 2 minutes to read

Um *tipo incompleto* é um tipo que descreve um identificador, mas que não tem informações necessárias para determinar o tamanho do identificador. Um tipo incompleto pode ser:

- Um tipo de estrutura cujos membros você ainda não especificou.
- Um tipo de união cujos membros você ainda não especificou.
- Um tipo de matriz cuja dimensão você ainda não especificou.

O `void` tipo é um tipo incompleto que não pode ser concluído. Para concluir um tipo incompleto, especifique as informações ausentes. Os exemplos a seguir mostram como criar e concluir os tipos incompletos.

- Para criar um tipo incompleto de estrutura, declare um tipo de estrutura sem especificar seus membros. Neste exemplo, o ponteiro `ps` aponta para um tipo incompleto de estrutura chamado `student`.

  ```
  struct student *ps;
  ```

- Para concluir um tipo incompleto de estrutura, declare o mesmo tipo de estrutura posteriormente no mesmo escopo com seus membros especificados, como em

  ```
  struct student
  {
      int num;
  }                   /* student structure now completed */
  ```

- Para criar um tipo incompleto de matriz, declare um tipo de matriz sem especificar sua contagem de repetições. Por exemplo:

  ```
  char a[];  /* a has incomplete type */
  ```

- Para concluir um tipo incompleto de matriz, declare o mesmo nome posteriormente no mesmo escopo com sua contagem de repetições especificada, como em

  ```
  char a[25]; /* a now has complete type */
  ```

Veja também

Declarações e tipos

Declarações typedef

13/05/2021 • 3 minutes to read

Uma declaração de typedef é uma declaração com typedef como a classe de armazenamento. O declarador se torna um novo tipo. Você pode usar declarações de typedef para construir nomes mais curtos ou significativos para os tipos já definidos por C ou para os tipos que você declarou. Os nomes de typedef permitem que você encapsule os detalhes da implementação que podem ser alterados.

Uma declaração de typedef é interpretada da mesma maneira que uma declaração de variável ou de função, mas o identificador, em vez de assumir o tipo especificado pela declaração, se tornará um sinônimo para o tipo.

Sintaxe

declaração:
  declaration-specifiers init-declarator-list$_{opt}$;

declaration-specifiers:
  declaração de classe de armazenamento-especificadores-$_{opcional}$
  declaração de especificador de tipo – aceitar especificadores
  declaração de qualificador de tipo – aceitar especificadores

storage-class-specifier:
  `typedef`

type-specifier:
  `void`
  `char`
  `short`
  `int`
  `long`
  `float`
  `double`
  `signed`
  `unsigned`
  especificador struct-ou-Union-
  especificador de enumeração
  typedef-Name

typedef-Name:
  ID

Observe que uma declaração de typedef não cria tipos. Ela cria sinônimos para tipos existentes ou nomes para os tipos que podem ser especificados de outras formas. Quando um nome de typedef é usado como um especificador de tipo, ele pode ser combinado com determinados especificadores de tipo, mas não outros. Os modificadores aceitáveis incluem `const` e `volatile` .

Os nomes de typedef compartilham o namespace com identificadores comuns (consulte Namespaces para obter mais informações). Portanto, um programa pode ter um nome de typedef e um identificador de escopo local com o mesmo nome. Por exemplo:

```
typedef char FlagType;

int main()
{
}

int 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. Este exemplo ilustra essa restrição:

```
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.

Os nomes de typedef podem ser usados para melhorar a legibilidade de código. As três declarações a seguir de `signal` especificam exatamente do mesmo tipo. O primeiro não usa os nomes de typedef.

```
typedef void fv( int ), (*pfv)( int );  /* typedef declarations */

void ( *signal( int, void (*) (int)) ) ( int );
fv *signal( int, fv * );   /* Uses typedef type */
pfv signal( int, pfv );    /* Uses typedef type */
```

Exemplos

Os exemplos a seguir ilustram as declarações de typedef:

```
typedef int WHOLE; /* Declares WHOLE to be a synonym for int */
```

Observe que agora `WHOLE` pode ser usado em uma declaração variável, como `WHOLE i;` ou `const WHOLE i;`. No

entanto, a declaração `long WHOLE i;` seria inválida.

```
typedef struct club
{
    char name[30];
    int size, year;
} GROUP;
```

Essa instrução declara `GROUP` como um tipo de estrutura com três membros. Como uma marca de estrutura, `club`, também está especificada, o nome de typedef (`GROUP`) ou a marca de estrutura pode ser usadas em declarações. Você deve usar a palavra-chave da estrutura com a marca e não pode usar a palavra-chave da estrutura com o nome de typedef.

```
typedef GROUP *PG; /* Uses the previous typedef name
                      to declare a pointer            */
```

O tipo `PG` é declarado como um ponteiro para o tipo `GROUP`, que por sua vez é definido como um tipo de estrutura.

```
typedef void DRAWF( int, int );
```

Esse exemplo fornece o tipo `DRAWF` para uma função que não retorna valor e que usa dois argumentos int. Isso significa, por exemplo, se a declaração

```
DRAWF box;
```

for equivalente à declaração

```
void box( int, int );
```

Veja também

Declarações e tipos

Atributos de classe de armazenamento estendido C

13/05/2021 • 2 minutes to read

Específico da Microsoft

Mais informações atualizadas sobre atributos de classe de armazenamento podem ser encontradas em `__declspec` (referência de C++).

A sintaxe de atributo estendido simplifica e padroniza as extensões específicas da Microsoft para a linguagem C. Os atributos de classe de armazenamento que usam a sintaxe de atributo estendida incluem `thread` , `naked` , `dllimport` e `dllexport` .

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 (`thread` , `naked` , `dllimport` ou `dllexport`). Exemplos de outros modificadores de classe de armazenamento incluem `static` as `extern` palavras-chave e. No entanto, essas palavras-chave fazem parte do padrão ISO C e não são cobertas pela sintaxe de atributo estendido.

Sintaxe

storage-class-specifier :
    `__declspec (` *extended-decl-modifier-seq* `)` /* Específico da Microsoft*/

extended-decl-modifier-seq : / * Específico da Microsoft*/
    extended-decl-modifier opt
    extended-decl-modifier-seq *extended-decl-modifier*

extended-decl-modifier : / * Específico da Microsoft*/
    `thread`
    `naked`
    `dllimport`
    `dllexport`

O espaço em branco separa os modificadores de declaração. Um *extended-decl-modifier-seq* pode estar vazio; nesse caso, `__declspec` não tem efeito.

Os `thread` atributos de classe de armazenamento,, e `naked` `dllimport` `dllexport` são uma propriedade somente da declaração dos dados ou da função à qual eles são aplicados. Eles não redefinem os atributos de tipo da própria função. O `thread` atributo afeta somente os dados. O `naked` atributo afeta apenas funções. Os `dllimport` `dllexport` atributos e afetam as funções e os dados.

FINAL específico da Microsoft

Confira também

Declarações e tipos

Importação e exportação de DLL

13/05/2021 • 2 minutes to read

Específico da Microsoft

Os modificadores da classe de armazenamento dllimport e dllexport são extensões específicas da Microsoft para a linguagem C. Esses modificadores definem a interface das DLLs para o cliente (o arquivo executável ou outra DLL). Para obter informações específicas sobre como usar esses modificadores, consulte dllexport, dllimport.

FINAL específico da Microsoft

Veja também

Atributos de Storage-Class estendidos C

Naked (C)

13/05/2021 • 2 minutes to read

Específico da Microsoft

O atributo de classe de armazenamento naked é uma extensão específica da Microsoft na linguagem C. O compilador gera um código sem código de prólogo e de epílogo para funções declaradas com o atributo de classe de armazenamento naked. As funções naked são úteis quando você precisa escrever suas próprias sequências de código de prólogo/epílogo usando o código de assembler embutido. As funções naked são úteis para escrever drivers para dispositivo virtuais.

Para obter informações específicas sobre como usar o atributo naked, consulte Funções naked.

FINAL específico da Microsoft

Veja também

Atributos de Storage-Class estendidos C

Armazenamento local de thread

13/05/2021 • 2 minutes to read

Específico da Microsoft

O armazenamento local de thread (TLS) é o mecanismo pelo qual cada thread em um processo multithread determinado aloca o armazenamento para dados específicos de threads. 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, confira Processos e threads no SDK do Windows.

A linguagem Microsoft C inclui o atributo de classe de armazenamento estendida, thread, que é usado com a palavra-chave __declspec para declarar uma variável local de thread. 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;
```

As seguintes diretrizes devem ser observadas ao declarar variáveis locais de threads associados estaticamente:

- Variáveis locais de thread com inicialização dinâmica são inicializadas somente no thread que faz com que a DLL seja carregada e nos threads que já estão em execução no processo. Para obter mais informações, veja thread.
- Você pode aplicar o atributo thread somente a declarações e definições de dados. Ele não pode ser usado em declarações ou definições de função. Por exemplo, o código a seguir gera um erro de compilador:

  ```
  #define Thread   __declspec( thread )
  Thread void func();      /* Error */
  ```

- Você pode especificar o atributo thread apenas em itens de dados com duração de armazenamento estática. Isso inclui dados globais (estáticos e externos) e dados estáticos locais. Você não pode declarar dados automáticos com o atributo thread. Por exemplo, o código a seguir gera erros de compilador:

  ```
  #define Thread   __declspec( thread )
  void func1()
  {
      Thread int tls_i;            /* Error */
  }

  int func2( Thread int tls_i )    /* Error */
  {
     return tls_i;
  }
  ```

- Você deve usar o atributo thread para a declaração e a definição de dados locais de thread, independentemente da declaração e a definição ocorrem no mesmo arquivo ou em arquivos separados. Por exemplo, o código a seguir gera um erro:

  ```
  #define Thread   __declspec( thread )
  extern int tls_i;     /* This generates an error, because the   */
  int Thread tls_i;     /* declaration and the definition differ. */
  ```

- Você não pode usar o atributo thread como um modificador de tipo. Por exemplo, o código a seguir gera um erro de compilador:

  ```
  char *ch __declspec( thread );      /* Error */
  ```

- O endereço de uma variável local de thread não é considerado constante, e nenhuma expressão que envolva esse endereço é considerada uma expressão constante. Isso significa que você não pode usar o endereço de uma variável local de thread como um inicializador para um ponteiro. Por exemplo, o compilador sinaliza o código a seguir como um erro:

  ```
  #define Thread   __declspec( thread )
  Thread int tls_i;
  int *p = &tls_i;      /* Error */
  ```

- O C permite a inicialização de uma variável com uma expressão que envolva uma referência a si mesma, mas apenas para objetos de extensão não estática. Por exemplo:

  ```
  #define Thread   __declspec( thread )
  Thread int tls_i = tls_i;             /* Error */
  int j = j;                            /* Error */
  Thread int tls_i = sizeof( tls_i )    /* Okay  */
  ```

  Observe que uma expressão sizeof que inclua a variável que está sendo inicializada não constitui uma referência a si mesma e é permitida.

- O uso de **__declspec (thread)** pode interferir no carregamento do atraso de importações de dll.

Para obter mais informações sobre como usar o atributo thread, consulte Tópicos de multithreading.

FINAL específico da Microsoft

Veja também

Atributos de Storage-Class estendidos C

Expressões e atribuições

13/05/2021 • 2 minutes to read

Esta seção descreve como formar expressões e atribuir valores na linguagem C. As constantes, os identificadores, as cadeias de caracteres e as chamadas de função são operandos manipulados em expressões. A linguagem C tem todos os operadores comuns de linguagem. Esta seção aborda esses operadores, bem como operadores que são exclusivos de C ou do Microsoft C. Os tópicos abordados incluem:

- Expressões l-value e r-value
- Expressões constantes
- Efeitos colaterais
- Pontos de sequência
- Operadores
- Precedência de operador
- Conversões de tipo
- Conversões de tipo

Veja também

Referência da linguagem C

Operandos e expressões

13/05/2021 • 2 minutes to read

Um "operando" é uma entidade em que um operador atua. Uma "expressão" é uma sequência de operadores e de operandos que executa qualquer combinação destas ações:

- Computa um valor
- Designa um objeto ou uma função
- Gera efeitos colaterais

Os operandos em C incluem constantes, identificadores, cadeias de caracteres, chamadas de função, expressões subscritas, expressões de seleção de membros e expressões complexas formadas combinando operandos com operadores ou incluindo operandos entre parênteses. A sintaxe desses operandos é determinada em Expressões primárias.

Veja também

Expressões e atribuições

Expressões primárias C

13/05/2021 • 2 minutes to read

As expressões primárias são os blocos de construção de expressões mais complexas. Elas podem ser constantes, identificadores, uma seleção genéricaou uma expressão entre parênteses.

Sintaxe

`primary-expression`:

`identifier`

`constant`

`string-literal`

(`expression`)

`generic-selection`

expressão:

`assignment-expression`

`expression` , `assignment-expression`

Veja também

Seleção genérica Operandos e expressões

Identificadores em expressões primárias

13/05/2021 • 2 minutes to read

Os identificadores podem ter um `float` tipo integral„ `enum` „„ de `struct` `union` matriz, de ponteiro ou de função. Um identificador é uma expressão primária contanto que tenha sido declarada como a designação de um objeto (nesse caso, um l-value) ou como uma função (nesse caso, um designador de função). Para obter uma definição de l-value, consulte Expressões de L-Value e R-Value.

O valor do ponteiro representado por um identificador de matriz não é uma variável, portanto, um identificador de matriz não pode formar o operando esquerdo de uma operação de atribuição e, em virtude disso, não é um l-value modificável.

Um identificador declarado como uma função representa um ponteiro cujo valor é o endereço da função. O ponteiro aborda uma função que retorna um valor de um tipo especificado. Portanto, os identificadores da função também não podem ser l-values em operações de atribuição. Para obter mais informações, consulte Identificadores.

Veja também

Expressões primárias C

Constantes em expressões primárias

13/05/2021 • 2 minutes to read

Um operando constante tem o valor e o tipo do valor da constante que representa. Uma constante de caractere tem `int` tipo. Uma constante de inteiro tem `int` , `long` , `unsigned int` ou `unsigned long` tipo, dependendo do tamanho do inteiro e da maneira como o valor é especificado. Consulte Constantes para obter mais informações.

Veja também

Expressões primárias C

literais String em expressões primárias

13/05/2021 • 2 minutes to read

Um "literal de cadeia de caracteres" é um caractere, um caractere largo ou uma sequência de caracteres adjacentes entre aspas duplas. Como não são variáveis, nem literais de cadeia de caracteres nem seus elementos podem ser o operando esquerdo em uma operação de atribuição. O tipo de um literal de cadeia de caracteres é uma matriz de `char` (ou uma matriz de `wchar_t` para literais de cadeia de caracteres largos). As matrizes nas expressões são convertidas em ponteiros. Consulte Literais de cadeia de caracteres para obter mais informações sobre cadeias de caracteres.

Veja também

Expressões primárias C

Expressões em parênteses

13/05/2021 • 2 minutes to read

É possível colocar qualquer operando entre parênteses sem alterar o tipo ou o valor da expressão dentro deles. Por exemplo, na expressão:

```
( 10 + 5 ) / 5
```

os parênteses `10 + 5` significam que o valor de `10 + 5` é avaliado primeiro e torna-se o operando esquerdo do operador Division (**/**). O resultado de `( 10 + 5 ) / 5` será 3. Sem os parênteses, `10 + 5 / 5` seria avaliado como 11.

Embora os parênteses afetem a maneira como os operandos são agrupados em uma expressão, eles não podem assegurar uma determinada ordem de avaliação em todos os casos. Por exemplo, nem os parênteses ou o agrupamento da esquerda para a direita da seguinte expressão assegura que o valor de `i` esteja em qualquer uma das subexpressões:

```
( i++ +1 ) * ( 2 + i )
```

O compilador é livre para avaliar os dois lados da multiplicação em qualquer ordem. Se o valor inicial de `i` for zero, toda a expressão pode ser avaliada como qualquer uma destas duas instruções:

```
( 0 + 1 + 1 ) * ( 2 + 1 )
( 0 + 1 + 1 ) * ( 2 + 0 )
```

As exceções que resultam dos efeitos colaterais são discutidas em Efeitos colaterais.

Veja também

Expressões primárias C

Seleção genérica (C11)

13/05/2021 • 2 minutes to read

Use a `_Generic` palavra-chave para escrever um código que selecione uma expressão em tempo de compilação com base no tipo do argumento. É semelhante ao sobrecarregamento em C++ em que o tipo do argumento seleciona qual função deve ser chamada, exceto que o tipo do argumento seleciona qual expressão deve ser avaliada.

Por exemplo, a expressão `_Generic(42, int: "integer", char: "character", default: "unknown");` avalia o tipo de `42` e procura o tipo correspondente, `int` , na lista. Ele o encontra e retorna `"integer"` .

Sintaxe

`generic-selection`:
  `_Generic` **(** *`assignment-expression`* **,** *`assoc-list`* **)**

`assoc-list`:
  `association`
  `assoc-list` **,** *`association`*

`association`:
  `type-name` **:** *`assignment-expression`*
  `default` **:** *`assignment-expression`*

A primeira *`assignment-expression`* é chamada de expressão de controle. O tipo da expressão de controle é determinado no momento da compilação e é correspondido com o *`assoc-list`* para localizar a expressão a ser avaliada e retornada. A expressão de controle não é avaliada. Por exemplo, `_Generic(intFunc(), int: "integer", default: "error");` não resulta em uma chamada em tempo de execução para `intFunc()` .

Quando o tipo da expressão de controle é determinado, `const` , `volatile` e é `restrict` removido antes da correspondência *`assoc-list`* .

As entradas no `assoc-list` que não são escolhidas não são avaliadas.

Restrições

- O *`assoc-list`* não pode especificar o mesmo tipo mais de uma vez.
- O *`assoc-list`* não pode especificar tipos que são compatíveis entre si, como uma enumeração e o tipo subjacente dessa enumeração.
- Se uma seleção genérica não tiver um padrão, a expressão de controle deverá ter apenas um nome de tipo compatível na lista de associação genérica.

Exemplo

Uma maneira de usar `_Generic` o está em uma macro. O arquivo de cabeçalho <tgmath. h> usa **_Generic** para chamar a função Math Right dependendo do tipo de argumento. Por exemplo, a macro para `cos()` mapeia uma chamada com um float para `cosf()` , ao mapear uma chamada com um Double complexo para `ccos()` .

O exemplo a seguir mostra como escrever uma macro que identifica o tipo do argumento que você passa para ele. Ele produz `"unknown"` se nenhuma entrada em *`assoc-list`* corresponde à expressão de controle:

```
// Compile with /std:c11

#include <stdio.h>

/* Get a type name string for the argument x */
#define TYPE_NAME(X) _Generic((X), \
      int: "int", \
      char: "char", \
      double: "double", \
      default: "unknown")

int main()
{
    printf("Type name: %s\n", TYPE_NAME(42.42));

    // The following would result in a compile error because
    // 42.4 is a double, doesn't match anything in the list,
    // and there is no default.
    // _Generic(42.4, int: "integer", char: "character"));
}

/* Output:
Type name: double
*/
```

Veja também

`/std` (Especifique a versão padrão do idioma)

Matemática do tipo genérico

Expressões L-Value e R-Value

13/05/2021 • 2 minutes to read

Expressões que fazem referência a locais de memória são chamadas de expressões "l-value". Um l-value representa o valor "localizador" de uma região de armazenamento ou um valor "esquerdo", indicando que ele pode aparecer à esquerda do sinal de igual (=). Os l-values frequentemente são identificadores.

Expressões que fazem referência aos locais modificáveis são chamadas "l-values modificáveis". Um l-Value modificável não pode ter um tipo de matriz, um tipo incompleto ou um tipo com o `const` atributo. Para estruturas e uniões terem valores l modificáveis, elas não devem ter nenhum membro com o `const` atributo. O nome do identificador denota um local de armazenamento, enquanto o valor da variável é o valor armazenado nesse local.

Um identificador é um l-value modificável se ele fizer referência a um local de memória e se seu tipo for aritmético, de estrutura, de união ou ponteiro. Por exemplo, se `ptr` for um ponteiro para uma região de armazenamento, então `*ptr` é um l-value modificável que designa a região de armazenamento para a qual `ptr` aponta.

Qualquer uma das seguintes expressões de C podem ser expressões l-value:

- Um identificador do tipo integral, flutuante, ponteiro, de estrutura ou de união
- Uma expressão subscrita (**[]**) que não seja avaliada como uma matriz
- Uma expressão de seleção de membro (`->` ou `.`)
- Uma expressão unário (*****) que não se refere a uma matriz
- Uma expressão l-value entre parênteses
- Um `const` objeto (um l-value não modificável)

O termo "r-value" é usado às vezes para descrever o valor de uma expressão e para diferenciá-lo de um l-value. Todos os l-values são r-values mas nem todos r-values são l-values.

Específico da Microsoft

O Microsoft C inclui uma extensão do padrão ANSI C que permite que as conversões de l- values sejam usadas como l-values, desde que o tamanho do objeto não seja alongado na conversão. (Consulte conversões de conversão de tipo para obter mais informações.) O exemplo a seguir ilustra esse recurso:

```
char *p ;
short  i;
long l;

(long *) p = &l ;       /* Legal cast   */
(long) i = l ;          /* Illegal cast */
```

O padrão do Microsoft C é que as extensões da Microsoft sejam habilitadas. Use a opção de compilador /Za para desativar essas extensões.

FINAL específico da Microsoft

Veja também

Expressões de constante C

13/05/2021 • 2 minutes to read

Uma expressão constante é avaliada no momento da compilação, não no tempo de execução e pode ser usada em qualquer lugar que uma constante possa ser usada. A expressão constante deve ser avaliada como uma constante que está no intervalo de valores representáveis para esse tipo. Os operandos de uma expressão constante podem ser constantes inteiras, constantes de caractere, constantes de ponto flutuante, constantes de enumeração, conversões de tipo, `sizeof` expressões e outras expressões de constante.

Sintaxe

constant-expression:
- *conditional-expression*

conditional-expression:
- *logical-OR-expression*
- *logical-OR-expression* `?` *expression* `:` *conditional-expression*

expression:
- *assignment-expression*
- *expression* `,` *assignment-expression*

assignment-expression:
- *conditional-expression*
- *unary-expression* *assignment-operator* *assignment-expression*

assignment-operator: um de
- `=` `*=` `/=` `%=` `+=` `-=` `<<=` `>>=` `&=` `^=` `|=`

Os não terminais para Declarador de struct, enumerador, Declarador direto, Declarador de resumo direto e instrução rotulada contêm o não *constant-expression* terminal.

Uma expressão constante integral deve ser usada para especificar o tamanho de um membro de campo de bits de uma estrutura, o valor de uma constante de enumeração, o tamanho de uma matriz ou o valor de uma `case` constante.

Expressões constantes usadas em diretivas de pré-processador estão sujeitas a várias restrições. Eles são conhecidos como expressões de constantes *restritas* . Uma expressão constante restrita não pode conter `sizeof` expressões, constantes de enumeração, conversões de tipo para qualquer tipo ou constantes de tipo flutuante. No entanto, ele pode conter o identificador de expressão constante especial `defined (` `)` .

Confira também

Operandos e expressões

Avaliação de expressão (C)

13/05/2021 • 2 minutes to read

Expressões que envolvem atribuição, incremento unário, diminuição unária ou chamando de uma função podem ter consequências incidentais à sua avaliação (efeitos colaterais). Quando um "ponto de sequência" é alcançado, tudo que precede o ponto de sequência, inclusive todos os efeitos colaterais, obrigatoriamente foram avaliados antes que a avaliação seja iniciada em qualquer coisa posterior ao ponto de sequência.

Os "efeitos colaterais" são alterações causadas pela avaliação de uma expressão. Os efeitos colaterais ocorrem sempre que o valor de uma variável é alterado por uma avaliação da expressão. Todas as operações de atribuição têm efeitos colaterais. As chamadas de função também poderão ter efeitos colaterais se alterarem o valor de um item externamente visível, seja pela atribuição direta ou indireta por meio de um ponteiro.

Veja também

Operandos e expressões

Efeitos colaterais

13/05/2021 • 2 minutes to read

A ordem de avaliação das expressões é definida pela implementação específica, exceto quando a linguagem garante uma determinada ordem de avaliação (conforme descrito em Precedência e ordem de avaliação). Por exemplo, os efeitos colaterais ocorrem nas seguintes chamadas de função:

```
add( i + 1, i = j + 2 );
myproc( getc(), getc() );
```

Os argumentos de uma chamada de função podem ser avaliados em qualquer ordem. A expressão `i + 1` pode ser avaliada antes de `i = j + 2`, ou `i = j + 2` pode ser avaliada antes de `i + 1`. O resultado é diferente em cada caso. De maneira similar, não é possível garantir quais caracteres serão passados para `myproc`. Como os incrementos unários e operações de redução envolvem atribuições, essas operações podem causar efeitos colaterais, conforme mostrado no seguinte exemplo:

```
x[i] = i++;
```

Neste exemplo, o valor de `x` alterado é imprevisível. O valor do subscrito pode ser o valor novo ou antigo de `i`. O resultado pode variar em compiladores diferentes ou níveis de otimização diferentes.

Como C não define a ordem de avaliação de efeitos colaterais, ambos os métodos de avaliação discutidos anteriormente estão corretos e qualquer um pode ser implementado. Para garantir que o seu código seja portátil e claro, evite as instruções que dependem de um pedido específico de avaliação quanto aos efeitos colaterais.

Veja também

Avaliação de expressão

Pontos de sequência C

13/05/2021 • 2 minutes to read

Entre os "pontos de sequência" consecutivos, o valor de um objeto pode ser modificado apenas uma vez por uma expressão. A linguagem C define os seguintes pontos de sequência:

- Operando esquerdo do operador AND lógico (`&&`). O operando da esquerda do operador AND lógico é completamente avaliado e todos os efeitos colaterais são concluídos antes de continuar. Se o operando da esquerda for avaliado como falso (0), o outro operando não será avaliado.
- O operando da esquerda do operador OR lógica (`||`). O operando da esquerda do operador OR lógica é completamente avaliado e todos os efeitos colaterais são concluídos antes de continuar. Se o operando da esquerda for avaliado como true (diferente de zero), o outro operando não será avaliado.
- Operando esquerdo do operador vírgula O operando da esquerda 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. Observe que o operador vírgula em uma chamada de função não garante a ordem de avaliação.
- Operador de chamada de função. Todos os argumentos para uma função são avaliados e todos os efeitos colaterais são concluídos antes de entrarem na função. Nenhuma ordem de avaliação entre os argumentos é especificada.
- 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 fim de uma expressão de inicialização completa (ou seja, uma expressão que não faz parte de outra expressão, como o fim 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 avaliada por seus efeitos colaterais e há um ponto de sequência que segue esta avaliação.
- A expressão de controle em uma instrução Selection (`if` ou `switch`). A expressão será completamente avaliada e todos os efeitos colaterais serão concluídos antes que o código dependente da seleção seja executado.
- A expressão de controle de `while` uma `do` instrução ou. A expressão é completamente avaliada e todos os efeitos colaterais são concluídos antes de qualquer instrução na próxima iteração do `while` `do` loop ou ser executada.
- Cada uma das três expressões de uma `for` instrução. As expressões são completamente avaliadas e todos os efeitos colaterais são concluídos antes que qualquer instrução na próxima iteração do `for` loop seja executada.
- A expressão em uma `return` instrução. A expressão será completamente avaliada e todos os efeitos colaterais serão concluídos antes que o controle retorne para a função de chamada.

Veja também

Avaliação de expressão

Operadores C

13/05/2021 • 2 minutes to read

Os operadores C são um subconjunto de operadores embutidos de C++.

Há três tipos de operadores. Uma expressão unário consiste em um operador unário que foi anexado a um operando ou à `sizeof` palavra-chave seguida por uma expressão. A expressão pode ser o nome de uma variável ou uma expressão de conversão. Se a expressão for uma expressão de conversão, ela deverá ser colocada entre parênteses. Uma expressão binária consiste em dois operandos unidos por um operador binário. Uma expressão ternária consiste em três operandos unidos pelo operador de expressão condicional.

O C inclui os seguintes operadores unários:

SÍMBOLO	NAME
- ~ !	Operadores de negação e de complemento
*****&	Operadores de indireção e address-of
`sizeof`	Operador de tamanho
+	Operador de adição de unário
++ --	Operadores unários de incremento e de decremento

Operadores binários associados da esquerda para a direita. O C fornece os seguintes operadores binários:

SÍMBOLO	NAME
* /****%	Operadores de multiplicação
+ -	Operadores aditivos
<< >>	Operadores shift
< >****<= >= == !=	Operadores relacionais
&|^	Operadores bit a bit
&&****||	Operadores lógicos
,	Operador de avaliação sequencial

O operador de base (:>), com suporte em versões anteriores do compilador de 16 bits do Microsoft C, é descrito em Resumo da sintaxe da linguagem C.

O operador de expressão condicional tem uma precedência mais baixa do que as expressões binárias e difere delas por ser associado à direita.

As expressões com operadores também incluem as expressões de atribuição, que usam operadores de atribuição unários ou binários. Os operadores de atribuição unários são os operadores Increment (++) e

Decrement (--); os operadores de atribuição Binary são o operador de atribuição simples (=) e os operadores de atribuição composta. Cada operador de atribuição composta é uma combinação de outro operador binário com o operador de atribuição simples.

Veja também

- Expressões e atribuições

Precedência e ordem da avaliação

13/05/2021 • 5 minutes to read

A precedência e a associatividade dos operadores C afetam o agrupamento e a avaliação dos operandos nas expressões. A precedência de um operador é significante somente quando outros operadores com precedência maior ou menor estão presentes. As expressões com operadores de maior precedência são avaliadas primeiro. A precedência também pode ser descrita pela palavra "associação". Os operadores com maior precedência são ditos ter uma associação mais próxima.

A tabela a seguir resume a precedência e a associatividade (a ordem em que os operandos são avaliados) dos operadores C, listando-os em ordem decrescente de precedência. Onde vários operadores aparecem juntos, eles têm a mesma precedência e são avaliados de acordo com sua associatividade. Os operadores na tabela são descritos nas seções que começam com Operadores de sufixo. O restante desta seção fornece informações gerais sobre a precedência e associatividade.

Precedência e associatividade de operadores C

SÍMBOLO [1]	TIPO DE OPERAÇÃO	CAPACIDADE DE ASSOCIAÇÃO
`[` `]` `(` `)` `.` `->` `++` `--` (sufixo)	Expression	Da esquerda para a direita
`sizeof` `&` `*` `+` `-` `~` `!` `++` `--` (prefixo)	Unário	Da direita para a esquerda
typecasts	Unário	Da direita para a esquerda
`*` `/` `%`	Multiplicativo	Da esquerda para a direita
`+` `-`	Aditiva	Da esquerda para a direita
`<<` `>>`	Deslocamento bit a bit	Da esquerda para a direita
`<` `>` `<=` `>=`	Relacional	Da esquerda para a direita
`==` `!=`	Igualitário	Da esquerda para a direita
`&`	Bitwise-AND	Da esquerda para a direita
`^`	Bitwise-exclusive-OR	Da esquerda para a direita
`|`	Bitwise-inclusive-OR	Da esquerda para a direita
`&&`	Logical-AND	Da esquerda para a direita
`||`	Logical-OR	Da esquerda para a direita
`? :`	Expressão condicional	Da direita para a esquerda

SÍMBOLO	TIPO DE OPERAÇÃO	CAPACIDADE DE ASSOCIAÇÃO
`=` `*=` `/=` `%=` `+=` `-=` `<<=` `>>=` `&=` `^=` `|=`	Atribuição simples e composta [2]	Da direita para a esquerda
`,`	Avaliação sequencial	Da esquerda para a direita

[1] Os operadores são listados em ordem decrescente de precedência. Quando vários operadores aparecem na mesma linha ou em um grupo, eles têm a mesma precedência.

[2] Todos os operadores de atribuição simples e composta têm a mesma precedência.

Uma expressão pode conter vários operadores com a mesma precedência. Quando vários desses operadores aparecem no mesmo nível em uma expressão, a avaliação procede de acordo com a associatividade do operador, da direita para a esquerda ou da esquerda para a direita. A direção da avaliação não afeta os resultados das expressões que incluem mais de um operador de multiplicação (`*`), adição (`+`) ou binário bit a bit (`&`, `|` ou `^`) no mesmo nível. A ordem das operações não é definida pela linguagem. O compilador é livre para avaliar essas expressões em qualquer ordem, se ele puder garantir um resultado consistente.

Somente os operadores de avaliação sequencial (`,`), AND lógico (`&&`), OR lógico (`||`), de expressão condicional (`? :`) e de chamada de função constituem pontos de sequência e, portanto, garantem uma ordem específica de avaliação de seus operandos. O operador da chamada de função é o conjunto de parênteses depois do identificador da função. O operador de avaliação sequencial (`,`) é garantido para avaliar seus operandos da esquerda para a direita. (O operador de vírgula em uma chamada de função não é o mesmo que o operador de avaliação sequencial e não fornece nenhuma garantia.) Para obter mais informações, consulte pontos de sequência.

Os operadores lógicos também garantem a avaliação de seus operandos da esquerda para a direita. No entanto, eles avaliam o menor número de operandos necessários para determinar o resultado da expressão. Isso é chamado de avaliação de" curto-circuito". Assim, alguns operandos da expressão não podem ser avaliados. Por exemplo, na expressão

```
x && y++
```

o segundo operando, `y++`, será avaliado somente se `x` for true (diferente de zero). Assim, `y` não será incrementado se `x` for false (0).

Exemplos

A lista a seguir mostra como o compilador associa automaticamente várias expressões de exemplo:

EXPRESSION	ASSOCIAÇÃO AUTOMÁTICA
`a & b || c`	`(a & b) || c`
`a = b || c`	`a = (b || c)`
`q && r || s--`	`(q && r) || s--`

Na primeira expressão, o operador AND bit a bit (`&`) tem uma maior precedência que o operador OR lógico (`||`), então `a & b` forma o primeiro operando da operação OR lógica.

Na segunda expressão, o operador OR lógico (`||`) tem maior precedência que o operador de atribuição simples (`=`), então `b || c` é agrupado como o operando à direita na atribuição. Observe que o valor atribuído

a `a` é 0 ou 1.

A terceira expressão mostra uma expressão corretamente formada que pode gerar um resultado inesperado. O operador AND lógico (`&&`) tem maior precedência que o operador OR lógico (`||`), então `q && r` é agrupado como um operando. Como os operadores lógicos asseguram a avaliação dos operandos da esquerda para a direita, `q && r` é avaliado antes de `s--` . No entanto, se `q && r` for avaliado como um valor diferente de zero, `s--` não será avaliado e `s` não será diminuído. Se não diminuir `s` for causar um problema em seu programa, `s--` deverá aparecer como o primeiro operando da expressão ou `s` deverá ser diminuído em uma operação separada.

A expressão a seguir é ilegal e gera uma mensagem de diagnóstico no tempo de compilação:

EXPRESSÃO ILEGAL	AGRUPAMENTO PADRÃO
`p == 0 ? p += 1: p += 2`	`( p == 0 ? p += 1 : p ) += 2`

Nessa expressão, o operador de igualdade (`==`) tem a precedência mais alta, então `p == 0` é agrupado como um operando. O operador de expressão condicional (`? :`) tem a próxima precedência mais alta. Seu primeiro operando é `p == 0` , e o segundo operando é `p += 1` . No entanto, o último operando do operador de expressão condicional é considerado `p` em vez de `p += 2` , já que essa ocorrência de `p` se associa ainda mais ao operador de expressão condicional do que ao operador de atribuição composta. Um erro de sintaxe ocorre porque `+= 2` não tem um operando à esquerda. Você deve usar parênteses para evitar erros desse tipo e gerar um código mais legível. Por exemplo, você pode usar parênteses como mostrado abaixo para corrigir e esclarecer o exemplo anterior:

```
( p == 0 ) ? ( p += 1 ) : ( p += 2 )
```

Veja também

Operadores de C

Conversões aritméticas normais

13/05/2021 • 2 minutes to read

A maioria dos operadores C executam conversões de tipos para avançar os operandos de uma expressão para um tipo comum ou estender valores resumidos para o tamanho do inteiro usado em operações do computador. As conversões realizadas por operadores C dependem do operador específico e do tipo dos operandos. Porém, muitos operadores executam conversões semelhantes em operandos dos tipos integral e flutuante. Essas conversões são conhecidas como "conversões aritméticas". A conversão de um valor de operando em um tipo compatível não faz qualquer alteração em seu valor.

As conversões aritméticas resumidas abaixo são chamadas "conversões aritméticas usuais". Essas etapas são aplicadas apenas a operadores binários que esperam o tipo aritmético. A finalidade é gerar um tipo comum que também seja o tipo do resultado. Para determinar quais conversões ocorrem realmente, o compilador aplica o seguinte algoritmo a operações binárias à expressão. As etapas abaixo não são uma ordem de precedência.

1. Se qualquer operando for do tipo `long double`, o outro operando será convertido em tipo `long double`.
2. Se a condição acima não for atendida e o operando for do tipo `double`, o outro operando será convertido em tipo `double`.
3. Se as duas condições acima não forem atendidas e o operando for do tipo `float`, o outro operando será convertido em tipo `float`.
4. Se as três condições anteriores não forem atendidas (nenhum dos operandos é dos tipos flutuantes), as conversões integrais são executadas nos operandos como segue:
   - Se qualquer operando for do tipo `unsigned long`, o outro operando será convertido em tipo `unsigned long`.
   - Se a condição acima não for atendida e o operando for do tipo `long` e o outro do tipo `unsigned int`, ambos os operandos serão convertidos para o tipo `unsigned long`.
   - Se as duas condições acima não forem atendidas e o operando for do tipo `long`, o outro operando será convertido em tipo `long`.
   - Se as três condições acima não forem atendidas e o operando for do tipo `unsigned int`, o outro operando será convertido em tipo `unsigned int`.
   - Se nenhuma das condições acima for atendida, ambos os operandos serão convertidos para o tipo `int`.

O código a seguir ilustra essas regras de conversão:

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

dVal = iVal * ulVal; /* iVal converted to unsigned long
                      * Uses step 4.
                      * Result of multiplication converted to double
                      */
dVal = ulVal + fVal; /* ulVal converted to float
                      * Uses step 3.
                      * Result of addition converted to double
                      */
```

Veja também

Operadores de C

Operadores pós-fixados

13/05/2021 • 2 minutes to read

Os operadores pós-fixados tem a precedência mais alta (a associação mais estreita) na avaliação da expressão.

Sintaxe

expressão de sufixo:
  primary-expression
  sufixo-expressão **[** *expressão* **]**
  *sufixo-expressão***(**opção de expressão de argumento-lista**)**
  expressão de sufixo **.** *identifier*
  expressão **->** de sufixo *identificador* do
  expressão de sufixo **++**
  expressão de sufixo **--**

Os operadores nesse nível de precedência são: subscritos de matriz, chamadas de função, membros de estrutura e união, e operadores de incremento e decremento pós-fixados.

Veja também

Operadores de C

Matrizes unidimensionais

13/05/2021 • 2 minutes to read

Uma expressão de pós-fixação seguida por uma expressão entre colchetes ([ ]) é uma representação assinada de um elemento de um objeto da matriz. Uma expressão subscrita representa o valor no endereço que está na posição *expression*, além de *postfix-expression* quando expresso como

```
postfix-expression [ expression ]
```

Geralmente, o valor representado por *postfix-expression* é um valor do ponteiro, como um identificador de matriz e *expression* é um valor integral. No entanto, tudo o que é necessário sintaticamente é que uma das expressões seja do tipo ponteiro e a outra seja do tipo integral. Assim, o valor integral pode estar na posição de *postfix-expression* e o valor do ponteiro pode estar entre colchetes na posição *expression* ou "subscrita". Por exemplo, esse código é válido:

```
// one_dimensional_arrays.c
int sum, *ptr, a[10];
int main() {
   ptr = a;
   sum = 4[ptr];
}
```

As expressões subscritas geralmente são usadas para se referir a elementos da matriz, mas você pode aplicar um subscrito a qualquer ponteiro. Independentemente da ordem dos valores, *expression* deve estar entre colchetes ([ ]).

A expressão de subscrito é avaliada adicionando o valor integral ao valor do ponteiro e, em seguida, aplicando o operador de indireção (*) ao resultado. (Consulte indireção e operadores de endereço para uma discussão sobre o operador de indireção.) Em vigor, para uma matriz unidimensional, as quatro expressões a seguir são equivalentes, supondo que `a` seja um ponteiro e `b` seja um inteiro:

```
a[b]
*(a + b)
*(b + a)
b[a]
```

De acordo com as regras de conversão para o operador de adição (determinadas em Operadores aditivos), o valor integral é convertido em um deslocamento do endereço mediante a multiplicação dele pelo comprimento do tipo resolvido pelo ponteiro.

Por exemplo, suponha que o identificador `line` se refere a uma matriz de `int` valores. O procedimento a seguir é usado para avaliar a expressão subscrita `line[ i ]`:

1. O valor inteiro `i` é multiplicado pelo número de bytes definidos como o comprimento de um `int` Item. O valor convertido de `i` representa `i` `int` posições.

2. Esse valor convertido é adicionado ao valor de ponteiro original (`line`) para produzir um endereço `i` `int` de deslocamento de posições `line` .

3. O operador de indireção é aplicado ao novo endereço. O resultado é o valor do elemento da matriz nessa posição (intuitivamente, `line [ i ]`).

A expressão subscrita `line[0]` representa o valor do primeiro elemento de linha, desde que o deslocamento de endereço representado por `line` seja 0. Similarmente, uma expressão como `line[5]` se refere ao deslocamento de cinco posições do elemento da linha, ou o sexto elemento da matriz.

Veja também

Operador de subscrito:

Matrizes multidimensionais (C)

13/05/2021 • 2 minutes to read

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, *expressão1* **[** *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 de ponteiro final atenda a um tipo de matriz (veja os exemplos abaixo).

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.

Exemplos

Para os exemplos a seguir, uma matriz chamada `prop` é declarada com três elementos, cada um deles é uma matriz de valores 4-by-6 `int` .

```
int prop[3][4][6];
int i, *ip, (*ipp)[6];
```

Uma referência à matriz `prop` é semelhante a esta:

```
i = prop[0][0][1];
```

O exemplo acima mostra como se referir ao segundo `int` elemento individual de `prop` . As matrizes são armazenadas por linha, de modo que o último subscrito varie mais rapidamente; a expressão `prop[0][0][2]` se refere ao próximo (terceiro) elemento da matriz, e assim por diante.

```
i = prop[2][1][3];
```

Essa instrução é uma referência mais complexa a um elemento individual de `prop`. A expressão é avaliada como segue:

1. O primeiro subscrito, `2` , é multiplicado pelo tamanho de uma matriz 4-por-6 `int` e adicionado ao valor do ponteiro `prop` . O resultado aponta para a terceira matriz 4 por 6 de `prop` .
2. O segundo subscrito, `1` , é multiplicado pelo tamanho da matriz de 6 elementos `int` e adicionado ao endereço representado por `prop[2]` .
3. Cada elemento da matriz de 6 elementos é um `int` valor, portanto, o subscrito final, `3` , é multiplicado pelo tamanho de um antes de ser `int` adicionado ao `prop[2][1]` . O ponteiro resultante é pertinente ao quarto elemento da matriz de 6 elementos.

4. O operador de indireção é aplicado ao valor do ponteiro. O resultado é o `int` elemento nesse endereço.

Os dois próximos exemplos mostra casos em que o operador de indireção não é aplicado.

```
ip = prop[2][1];

ipp = prop[2];
```

Na primeira dessas instruções, a expressão `prop[2][1]` é uma referência válida para a matriz tridimensional `prop`; refere-se a uma matriz de 6 elementos (declarada acima). Como o valor do ponteiro é pertinente a uma matriz, o operador de indireção não é aplicado.

Da mesma forma, o resultado da expressão `prop[2]` na segunda instrução `ipp = prop[2];` é um valor de ponteiro pertinente a uma matriz bidimensional.

Veja também

Operador de subscrito:

Chamada de função (C)

13/05/2021 • 2 minutes to read

Uma *chamada de função* é uma expressão que inclui o nome da função que está sendo chamada ou o valor de um ponteiro de função e, opcionalmente, os argumentos que estão sendo passados para a função.

Sintaxe

expressão de sufixo:
*sufixo-expressão***(****opção de expressão de argumento-lista***)**

argument-expression-list:
expressão de atribuição
argumento-expressão-lista **,** *expressão de atribuição*

O *postfix-expression* deve ser avaliado para um endereço de função (por exemplo, um identificador de função ou o valor de um ponteiro de função) e o *argument-expression-list* é uma lista de expressões (separadas por vírgulas) cujos valores (os "argumentos") são passados para a função. O argumento *argument-expression-list* pode estar vazio.

Uma expressão de chamada de função tem o valor e o tipo do valor de retorno da função. Uma função não pode retornar um objeto do tipo de matriz. Se o tipo de retorno da função for `void` (ou seja, a função tiver sido declarada nunca para retornar um valor), a expressão de chamada de função também terá o `void` tipo. (Consulte Chamadas de função para obter mais informações.)

Veja também

Operador de chamada de função: ()

Membros de união e estrutura

13/05/2021 • 2 minutes to read

Uma "expressão de seleção de membros" faz referência a membros de estruturas e de uniões. Essas expressões têm o valor e o tipo do membro selecionado.

> *expressão de sufixo* . *ID*
> *expressão* -> de sufixo *identificador* do

A lista a seguir descreve os dois formatos de expressões de seleção de membros:

1. No primeiro formulário, o *sufixo-expressão* representa um valor `struct` ou `union` tipo, e o *identificador* nomeia um membro da estrutura ou União especificada. O valor da operação é o de *identificador* e é um l-value se *postfix-expression* for um l-value. Consulte Expressões de L-value e R-value para obter mais informações.

2. No segundo formato, *postfix-expression* representa um ponteiro para uma estrutura ou união e *identificador* nomeia um membro da estrutura ou união especificada. O valor é o de *identificador* e é um l-value.

Os dois formatos de expressões de seleção de membros têm efeitos semelhantes.

Na verdade, uma expressão que envolve o operador de seleção de membro (`->`) é uma versão abreviada de uma expressão usando o ponto final (.) se a expressão antes do período consistir no operador de indireção (*****) aplicado a um valor de ponteiro. Portanto,

```
expression->identifier
```

é equivalente a

```
(*expression).identifier
```

quando *expressão* for um valor do ponteiro.

Exemplos

Os exemplos a seguir fazem referência a essa declaração de estrutura. Para obter informações sobre o operador de indireção (*****) usado nesses exemplos, consulte indireção e operadores de endereço.

```
struct pair
{
    int a;
    int b;
    struct pair *sp;
} item, list[10];
```

Uma expressão de seleção de membros para a estrutura `item` é semelhante a:

```
item.sp = &item;
```

No exemplo anterior, o endereço da estrutura `item` é atribuído ao membro `sp` da estrutura. Isso significa que `item` contém um ponteiro para si mesmo.

```
(item.sp)->a = 24;
```

Neste exemplo, a expressão de ponteiro `item.sp` é usada com o operador de seleção de membro (`->`) para atribuir um valor ao membro `a` .

```
list[8].b = 12;
```

Essa instrução mostra como selecionar um membro individual da estrutura em uma matriz de estruturas.

Veja também

Operadores de acesso de membro: . e ->

Operadores de incremento e de decremento pós-fixados C

13/05/2021 • 2 minutes to read

Os operandos dos operadores de incremento e decremento pós-fixados são tipos escalares que são l-values modificáveis.

Sintaxe

expressão de sufixo:
  expressão de sufixo **++**
  expressão de sufixo **--**

O resultado da operação de incremento ou decremento pós-fixada é o valor do operando. Depois que o resultado é obtido, o valor do operando é incrementado (ou decrementado). O código a seguir ilustra o operador de incremento pós-fixado.

```
if( var++ > 0 )
    *p++ = *q++;
```

Nesse exemplo, a variável `var` é comparada a 0 e depois incrementada. Se `var` era positiva antes de ser incrementada, a próxima instrução é executada. Primeiro, o valor do objeto para o qual `q` aponta é atribuído ao objeto para o qual `p` aponta. Em seguida, `q` e `p` são incrementados.

Veja também

Operadores de incremento e decréscimo de sufixo: + + e--

Operadores unários C

13/05/2021 • 2 minutes to read

Os operadores unários aparecem antes do respectivo operando e são associados da direita para a esquerda.

Sintaxe

unary-expression: *postfix-expression*

++ *expressão unário*

`--` *expressão-unária*

unary-operator cast-expression

`sizeof` *expressão unário*

sizeof (*type-name* **)**

operador unário: um de **& * + - ~ !**

Veja também

Operadores de C

Operadores de incremento e de decremento pré-fixados

13/05/2021 • 2 minutes to read

Os operadores unários (`++` e --) são chamados de incrementos de "prefixo" ou diminuição de operadores quando os operadores de aumento ou diminuição aparecem antes do operando. O incremento e a diminuição de pós-fixação têm precedência maior que o incremento e a diminuição de prefixo. O operando deve ter um tipo integral, flutuante ou de ponteiro e deve ser uma expressão l-Value modificável (uma expressão sem o `const` atributo). O resultado é um l-value.

Quando o operador aparecer antes de seu operando, o operando será incrementado ou diminuído, e seu novo valor será o resultado da expressão.

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.

Exemplo

Este exemplo ilustra o operador unário de diminuição de prefixo:

```
if( line[--i] != '\n' )
    return;
```

Neste exemplo, a variável `i` é diminuída antes de ser usada como um subscrito para `line`.

Veja também

Operadores unários C

Operadores de indireção e endereço de

13/05/2021 • 2 minutes to read

O operador unário de indireção (*) acessa um valor indiretamente, por meio de um ponteiro. O operando deve ser um tipo de ponteiro. O resultado da operação é o valor resolvido pelo operando; isto é, o valor no endereço para o qual o operando aponta. O tipo do resultado é o tipo endereçado pelo operando.

O resultado do operador de indireção será *tipo* se o operando for do tipo *ponteiro para tipo*. Se o operando apontar para uma função, o resultado será um designador de função. Se ele apontar para um objeto, o resultado será um lvalue que designa o objeto.

Se o valor do ponteiro não for válido, o resultado do operador de indireção será indefinido. Estas são algumas das condições mais comuns que invalidam um valor de ponteiro:

- O ponteiro é um ponteiro nulo.
- O ponteiro especifica o endereço de um objeto após o fim de seu tempo de vida (como um objeto fora do escopo ou desalocado) 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.

O operador address-of unário (**&**) fornece o endereço de seu operando. O operando precisa ser um lvalue que designa um objeto não declarado como **registro** e não seja um campo de bits, ou o resultado de um operador unário *****, ou um operador de desreferenciação de matriz (**[]**), ou um designador de função. O resultado é do tipo *ponteiro para tipo* para um operando do tipo *tipo*.

Se o operando for o resultado de um operador unário *****, o operador não será avaliado e o resultado omitirá ambos. O resultado não é um lvalue e as restrições ainda se aplicam aos operadores. Se o operando for o resultado de um operador **[]**, o operador **&** e o operador unário ***** implícito no **[]** não serão avaliados. O resultado tem o mesmo efeito que remover o **&** operador e alterar o operador de **[]** para um **+** operador. Caso contrário, o resultado será um ponteiro para o objeto ou função designado pelo operando.

Exemplos

Os exemplos a seguir usam estas declarações comuns:

```
int *pa, x;
int a[20];
double d;
```

Essa instrução usa o operador address-of (**&**) para obter o endereço do sexto elemento da matriz `a` . O resultado é armazenado na variável de ponteiro `pa`:

```
pa = &a[5];
```

O operador de indireção (*) é usado neste exemplo para acessar o `int` valor no endereço armazenado em `pa` . O valor é atribuído à variável de inteiro `x`:

```
x = *pa;
```

Este exemplo demonstra que o resultado da aplicação do operador de indireção ao endereço de `x` é o mesmo que `x`:

```
assert( x == *&x );
```

Este exemplo mostra as formas semelhantes de declarar um ponteiro para uma função:

```
int roundup( void );     /* Function declaration */

int  *proundup  = roundup;
int  *pround  = &roundup;
assert( pround == proundup );
```

Uma vez que a função `roundup` é declarada, dois ponteiros para `roundup` são declarados e inicializados. O primeiro ponteiro, `proundup`, é inicializado usando apenas o nome da função, enquanto o segundo, `pround`, usa o operador address-of na inicialização. As inicializações são equivalentes.

Veja também

Operador de indireção: *
Operador address-of: &

Operadores aritméticos unários

13/05/2021 • 2 minutes to read

Vários operadores do C – unário de adição, de negação aritmética, de complemento e de negação lógica – são discutidos na lista a seguir:

OPERADOR	DESCRIÇÃO
+	O operador unário de adição que precede uma expressão entre parênteses força o agrupamento das operações incluídas. É usado com expressões que envolvem mais de um operador binário associativo ou comutativo. O operando deve ser do tipo aritmético. O resultado é o valor do operando. Um operando de integral passa por uma promoção de integral. O tipo do resultado é o tipo do operando promovido.
-	O operador de negação aritmética produz o negativo (complemento de dois) do respectivo operando. O operando deve ser um valor de integral ou flutuante. Este operador executa as conversões aritméticas comuns.
`~`	O operador de complemento bit a bit (ou NOT bit a bit) produz o complemento bit a bit do respectivo operando. O operando deve ser do tipo integral. Este operador executa as conversões aritméticas comuns; o resultado tem o tipo do operando após a conversão.
!	O operador de negação lógica (NOT lógico) produz o valor 0 se o operando é verdadeiro (diferente de zero) e o valor 1 se o operando é falso (0). O resultado tem `int` tipo. O operando deve ser um valor de ponteiro, integral ou flutuante.

Operações aritméticas unárias em ponteiros não são válidas.

Exemplos

Os exemplos a seguir ilustram os operadores aritméticos unários:

```
short x = 987;
    x = -x;
```

No exemplo acima, o novo valor de `x` é o negativo de 987, ou seja, -987.

```
unsigned short y = 0xAAAA;
    y = ~y;
```

Nesse exemplo, o novo valor atribuído a `y` é o complemento de um do valor sem sinal 0xAAAA, ou 0x5555.

```
if( !(x < y) )
```

Se `x` é maior ou igual a `y`, o resultado da expressão é 1 (verdadeiro). Se `x` é menor que `y`, o resultado é 0 (falso).

Veja também

Expressões com operadores unários

Operador sizeof (C)

13/05/2021 • 2 minutes to read

O `sizeof` operador fornece a quantidade de armazenamento, em bytes, necessária para armazenar um objeto do tipo do operando. Esse operador permite que você evite especificar tamanhos de dados dependentes do computador em seus programas.

Sintaxe

```
sizeof unary-expression
sizeof ( type-name )
```

Comentários

O operando é um identificador que é uma *unary-expression* ou uma expressão de conversão de tipo (ou seja, um especificador de tipo incluído entre parênteses). A *unary-expression* não pode representar um objeto de campo de bits, um tipo incompleto ou um designador de função. O resultado é uma constante integral sem sinal. O cabeçalho padrão STDDEF.H define esse tipo como **size_t**.

Quando você aplica o `sizeof` operador a um identificador de matriz, o resultado é o tamanho de toda a matriz, em vez do tamanho do ponteiro representado pelo identificador de matriz.

Quando você aplica o `sizeof` operador a um nome de tipo de estrutura ou de União, ou a um identificador de estrutura ou tipo de União, o resultado é o número de bytes na estrutura ou União, incluindo preenchimento interno e à direita. Esse tamanho pode incluir o preenchimento interno e à direita usado para alinhar os membros da estrutura ou união nos limites de memória. Assim, o resultado pode não corresponder ao tamanho calculado pela adição dos requisitos de armazenamento dos membros individuais.

Se uma matriz sem tamanho for o último elemento de uma estrutura, o `sizeof` operador retornará o tamanho da estrutura sem a matriz.

```
buffer = calloc(100, sizeof (int) );
```

Este exemplo usa o `sizeof` operador para passar o tamanho de um `int` , que varia entre computadores, como um argumento para uma função de tempo de execução chamada `calloc` . O valor retornado pela função é armazenado em `buffer` .

```
static char *strings[] = {
      "this is string one",
      "this is string two",
      "this is string three",
   };
const int string_no = ( sizeof strings ) / ( sizeof strings[0] );
```

Neste exemplo, `strings` é uma matriz de ponteiros para `char` . O número de ponteiros é o número de elementos na matriz, mas não é especificado. É fácil determinar o número de ponteiros usando o `sizeof` operador para calcular o número de elementos na matriz. O `const` valor inteiro `string_no` é inicializado para esse número. Porque é um `const` valor, `string_no` não pode ser modificado.

Veja também

Operadores de C

Operadores internos C++, precedência e associatividade

Operadores cast

13/05/2021 • 2 minutes to read

Uma conversão de tipo fornece um método para conversão explícita do tipo de um objeto em uma situação específica.

Sintaxe

cast-expression: *unary-expression*

expressão CAST (***nome-do-tipo***)

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 restringidas pelas mesmas regras que determinam os efeitos de conversões implícitas, descritos em Conversões de atribuição. As restrições adicionais de conversões podem resultar de tamanhos reais ou de representação de tipos específicos. Consulte Armazenamento de tipos básicos para obter informações sobre tamanhos reais de tipos integrais. Para obter mais informações sobre conversões de tipos, consulte Conversões Type-Cast.

Veja também

Operador cast: ()

Operadores multiplicativos C

13/05/2021 • 2 minutes to read

Os operadores multiplicativa executam operações de multiplicação (*****), divisão (**/**) e resto (**%**).

Sintaxe

multiplicativa- expressão: *Cast-Expression* *multiplicativa-expressão* ***** *Cast-* Expression
multiplicativa- Expression **/** *Cast-* expressão *multiplicativa-* Expression **%** *Cast-Expression*

Os operandos do operador resto (**%**) devem ser integral. Os operadores de multiplicação (*****) e divisão (**/**) podem ter operandos de tipo inteiro ou flutuante; os tipos de operandos podem ser diferentes.

Os operadores multiplicativos executam as conversões aritméticas comuns nos operandos. O tipo do resultado é o tipo dos operandos após conversão.

> **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.

Os operadores multiplicativos C são descritos abaixo:

OPERADOR	DESCRIÇÃO
*	O operador de multiplicação faz com que dois operandos sejam multiplicados.
/	O operador de divisão faz com que o primeiro operando seja dividido pelo segundo. Se dois operandos de inteiro forem divididos e o resultado não for um inteiro, ele será truncado de acordo com as seguintes regras: - O resultado da divisão por 0 é indefinida de acordo com o padrão ANSI C. O compilador do Microsoft C gera um erro no tempo de compilação ou no tempo de execução. - Se ambos os operandos forem positivos ou sem sinal, o resultado será truncado para 0. - Se qualquer operando for negativo, se o resultado da operação for o maior inteiro menor ou igual ao quociente algébrico ou o menor inteiro maior ou igual ao quociente algébrico, ele será definido pela implementação. (Consulte a seção específica da Microsoft abaixo.)

OPERADOR	DESCRIÇÃO
%	O resultado do operador de restante é o restante quando o primeiro operando é dividido pelo segundo. Quando a divisão não é exata, o resultado é determinado pelas seguintes regras: - Se o operando da direita for zero, o resultado será indefinido. - Se ambos os operandos forem positivos ou sem sinal, o resultado será positivo. - Se qualquer operando for negativo e o resultado não for exato, o resultado será definido para implementação. (Consulte a seção específica da Microsoft abaixo.)

Específico da Microsoft

A divisão onde qualquer operando é negativo, a direção de truncamento é sentido a 0.

Se uma ou outra operação for negativa na divisão com o operador de restante, o resultado terá o mesmo sinal do dividendo (o primeiro operando na expressão).

Exemplos

As declarações mostradas abaixo são usadas para os exemplos a seguir:

```
int i = 10, j = 3, n;
double x = 2.0, y;
```

Esta instrução usa o operador de multiplicação:

```
y = x * i;
```

Nesse caso, `x` é multiplicado por `i` para fornecer o valor 20,0. O resultado tem `double` tipo.

```
n = i / j;
```

Nesse exemplo, 10 é dividido por 3. O resultado é truncado para 0, gerando um valor inteiro 3.

```
n = i % j;
```

Essa instrução aloca a `n` ao restante inteiro, 1, quando 10 é dividido por 3.

Específico da Microsoft

O sinal do restante é o mesmo sinal do dividendo. Por exemplo:

```
50 % -6 = 2
-50 % 6 = -2
```

Em cada caso, `50` e `2` têm o mesmo sinal.

FINAL específico da Microsoft

Veja também

Operadores multiplicativa e o operador de módulo

Operadores aditivos C

13/05/2021 • 2 minutes to read

Os operadores aditivos executam adição (**+**) e subtração (**-**).

Sintaxe

additive-expression:
  multiplicative-expression
  expressão aditiva **+** *multiplicativa-expressão*
  expressão aditiva **-** *multiplicativa-expressão*

> **NOTE**
> Embora a sintaxe de *additive-expression* inclua *multiplicative-expression*, isso não significa que expressões que usam multiplicação sejam necessárias. Consulte a sintaxe em Resumo de sintaxe da linguagem C, para *multiplicative-expression*, *cast-expression* e *unary-expression*.

Os operandos podem ser valores integrais ou flutuantes. Algumas operações aditivas também podem ser executadas em valores de ponteiro, como descrito na discussão de cada operador.

Os operadores aditivos executam as conversões aritméticas comuns em operandos do tipo integral ou flutuantes. O tipo do resultado é o tipo dos operandos após conversão. Como as conversões executadas pelos operadores aditivos não fornecem condições de estouro ou de estouro negativo, as informações podem ser perdidas se o resultado de uma operação aditiva não puder ser representado no tipo dos operandos após a conversão.

Veja também

Operadores aditivos: + e-

Adição (+)

13/05/2021 • 2 minutes to read

O operador de adição (+) faz com que seus dois operandos sejam adicionados. Ambos os operandos podem ser do tipo integral ou flutuante, ou um operando pode ser um ponteiro e o outro um inteiro.

Quando um inteiro é adicionado a um ponteiro, o valor do inteiro (*i*) é convertido ao multiplicá-lo pelo tamanho do valor que o ponteiro indica. Após a conversão, o valor de inteiro representa as posições de memória *i*, no qual cada posição tem o comprimento especificado pelo tipo de ponteiro. Quando o valor do inteiro convertido é somado ao valor do ponteiro, o resultado é um novo valor de ponteiro que representa as posições do endereço *i* do endereço original. O novo valor de ponteiro indica um valor do mesmo tipo que o valor do ponteiro original e, portanto, é o mesmo que a indexação de matriz (consulte Matrizes unidimensionais e Matrizes multidimensionais). Se o ponteiro da soma apontar para fora da matriz, exceto no primeiro local após a extremidade, o resultado será indefinido. Para obter mais informações, consulte o tópico sobre aritmética de ponteiros.

Veja também

Operadores aditivos C

Subtração (-)

13/05/2021 • 2 minutes to read

O operador de subtração (-) subtrai o segundo operando do primeiro. Ambos os operandos podem ser do tipo integral ou flutuante, ou um operando pode ser um ponteiro e o outro um inteiro.

Quando dois ponteiros são subtraídos, a diferença é convertida em um valor integral assinado dividindo a diferença pelo tamanho de um valor do tipo que os ponteiros indicam. O tamanho do valor integral é definido pelo tipo **ptrdiff_t** no arquivo STDDEF.H de inclusão padrão. O resultado representa o número de posições de memória desse tipo entre os dois endereços. O resultado é garantido somente para ser significativo para dois elementos da mesma matriz, como discutido em Aritmética do ponteiro.

Quando um valor inteiro é subtraído de um valor de ponteiro, o operador de subtração converte o valor do inteiro (*i*) multiplicando-o pelo tamanho do valor que o ponteiro indica. Após a conversão, o valor de inteiro representa as posições de memória *i*, no qual cada posição tem o comprimento especificado pelo tipo de ponteiro. Quando o valor do inteiro convertido é subtraído do valor do ponteiro, como resultado, o endereço de memória *i* se posiciona antes do endereço original. O novo ponteiro aponta para um valor do tipo indicado pelo valor do ponteiro original.

Veja também

Operadores aditivos C

Usando os operadores Additive

13/05/2021 • 2 minutes to read

Os exemplos a seguir, que ilustram os operadores de adição e subtração, usam estas declarações:

```
int i = 4, j;
float x[10];
float *px;
```

Estas instruções são equivalentes:

```
px = &x[4 + i];
px = &x[4] + i;
```

O valor de `i` é multiplicado pelo comprimento de um `float` e adicionado a `&x[4]` . O valor de ponteiro resultante é o endereço de `x[8]` .

```
j = &x[i] - &x[i-2];
```

Nesse exemplo, o endereço do terceiro elemento de `x` (determinado por `x[i-2]`) é subtraído do endereço do quinto elemento de `x` (determinado por `x[i]`). A diferença é dividida pelo comprimento de a `float` ; o resultado é o valor inteiro 2.

Veja também

Operadores aditivos C

Ponteiro aritmético

13/05/2021 • 2 minutes to read

As operações de adição envolvendo um ponteiro e um inteiro fornecem resultados significativos apenas se o operando do ponteiro endereçar um membro de matriz e o valor inteiro produzir um deslocamento dentro dos limites da mesma matriz. Quando o valor inteiro é convertido em um deslocamento de endereço, o compilador pressupõe que somente as posições de memória do mesmo tamanho se encontram entre o endereço original e o endereço mais o deslocamento.

Esse pressuposto é válido para membros de matriz. Por definição, uma matriz é uma série de valores do mesmo tipo; seus elementos residem em locais de memória contíguos. Porém, o armazenamento para alguns tipos exceto elementos da matriz não têm garantia de preenchimento pelo mesmo tipo de identificadores. Ou seja, os espaços em branco podem aparecer entre as posições de memória, até mesmo posições do mesmo tipo. Consequentemente, os resultados da adição a ou da subtração de endereços de quaisquer valores que não os elementos da matriz serão indefinidos.

Da mesma forma, quando dois valores de ponteiro são subtraídos, a conversão pressupõe que somente valores do mesmo tipo, sem espaços em branco, se encontram entre os endereços dados pelos operandos.

Veja também

Operadores aditivos C

Operadores shift bit a bit

13/05/2021 • 2 minutes to read

Os operadores Shift alternam seu primeiro operando < < para a esquerda () ou direita (> >) pelo número de posições que o segundo operando especifica.

Sintaxe

expressão Shift:
  expressão aditiva
  expressão Shift < < *expressão aditiva*
  expressão Shift > > *expressão aditiva*

Ambos os operandos devem ser valores integrais. Esses operadores executam conversões aritméticas comuns; o tipo de resultado é o tipo do operando à esquerda após a conversão.

Para mudanças à esquerda, os bits vazios à direita são definidos como 0. Para a mudança à direita, os bits vazios à esquerda são preenchidos com base no tipo do primeiro operando após a conversão. Se o tipo for `unsigned`, eles serão definidos como 0. Caso contrário, eles são preenchidos com cópias do bit de sinal. Para os operadores Left Shift sem estouro, a instrução

```
expr1 << expr2
```

é equivalente à multiplicação por 2^{expr2}. Para os operadores Right Shift,

```
expr1 >> expr2
```

é equivalente à divisão por 2^{expr2} se `expr1` não estiver assinado ou tiver um valor não negativo.

O resultado de uma operação de deslocamento é indefinido se o segundo operando for negativo, ou se o operando à direita for maior ou igual à largura em bits do operando promovido à esquerda.

Como as conversões executadas pelos operadores Shift não fornecem condições de estouro ou estouro negativo, as informações poderão ser perdidas se o resultado de uma operação de deslocamento não puder ser representado no tipo do primeiro operando após a conversão.

```
unsigned int x, y, z;

x = 0x00AA;
y = 0x5500;

z = ( x << 8 ) + ( y >> 8 );
```

Neste exemplo, `x` é deslocado para a esquerda oito posições e `y` é deslocado para a direita oito posições. Os valores deslocados são adicionados, resultando em 0xAA55 e atribuídos a `z`.

O deslocamento de um valor negativo para a direita gera a metade do valor original, arredondado para baixo. Por exemplo, -253 (binário 11111111 00000011) com o deslocamento para a direita de um de bit produz -127 (11111111 10000001). Um deslocamento positivo de 253 para a direita gera +126.

Os deslocamentos para a direita preservam o bit de sinal. Quando um inteiro assinado é deslocado para a

direita, o bit mais significativo permanece definido. Quando um inteiro não assinado é deslocado para a direita, o bit mais significativo é limpo.

Veja também

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

Operadores relacionais e de igualdade C

13/05/2021 • 3 minutes to read

Os operadores relacionais binários e de igualdade comparam o primeiro operando ao segundo operando para testar a validade da relação especificada. O resultado de uma expressão relacional é 1 se a relação testada for verdadeira e 0 se for falsa. O tipo do resultado é `int` .

Sintaxe

expressão relacional:
  shift-expression
  expressão < relacional *expressão Shift*
  expressão > relacional *expressão Shift*
  expressão < = relacional *expressão Shift*
  expressão > = relacional *expressão Shift*

equality-expression:
  relational-expression
  expressão = = de igualdade *expressão relacional*
  equality-expression ! = *relational-expression*

Os operadores relacionais e de igualdade testam as seguintes relações:

OPERADOR	RELAÇÃO TESTADA
<	Primeiro operando menor que o segundo operando
>	Primeiro operando maior que o segundo operando
< =	Primeiro operando menor ou igual ao segundo operando
> =	Primeiro operando maior ou igual ao segundo operando
= =	Primeiro operando igual ao segundo operando
! =	Primeiro operando não é igual ao segundo operando

Os primeiros quatro operadores na lista acima têm alta prioridade quanto aos operadores de igualdade (`==` e `!=`). Consulte as informações de precedência na tabela Precedência e associatividade dos operadores de C.

Os operandos podem ter o tipo integral, flutuação ou ponteiro. Os tipos dos operandos podem ser diferentes. Os operadores relacionais executam as conversões aritméticas comuns em operandos do tipo integral ou flutuação. Além disso, você pode usar as seguintes combinações de tipos de operando com os operadores relacionais e de igualdade:

- Ambos os operandos de qualquer operador relacional ou de igualdade podem ser ponteiros para o mesmo tipo. Para os operadores de igualdade (`==`) e de desigualdade (`!=`), o resultado da comparação indica se os dois ponteiros endereçam o mesmo local da memória. Para os outros operadores relacionais (< , > , < =, and > =), o resultado da comparação indica a posição relativa dos dois endereços de memória dos objetos apontados. Os operadores relacionais são apenas deslocamentos.

  A comparação do ponteiro é definida apenas para partes do mesmo objeto. Se os ponteiros fizerem

referência aos membros de uma matriz, uma comparação é equivalente à comparação dos subscritos correspondentes. O endereço do primeiro elemento da matriz é "menor que" o endereço do último elemento. No caso de estruturas, os ponteiros para os membros da estrutura declarados posteriormente são ponteiros "maiores que" para os membros declarados anteriormente na estrutura. Os ponteiros para os membros da mesma união são iguais.

- Um valor de ponteiro pode ser comparado ao valor da constante 0 para igualdade (`==`) ou desigualdade (`!=`). Um ponteiro com um valor de 0 é chamado de ponteiro "nulo"; ou seja, ele não aponta para um local de memória válido.
- Os operadores de igualdade seguem as mesmas regras que os operadores relacionais, mas permitem possibilidades adicionais: um ponteiro pode ser comparado a uma expressão integral constante com valor 0 ou a um ponteiro para `void` . Se os dois ponteiros forem nulos, eles serão comparados como iguais. Os operadores de igualdade comparam o segmento e o deslocamento.

Exemplos

Os exemplos a seguir ilustram os operadores relacionais e de igualdade.

```
int x = 0, y = 0;
if ( x < y )
```

Como `x` e `y` são iguais, a expressão neste exemplo gera o valor 0.

```
char array[10];
char *p;

for ( p = array; p < &array[10]; p++ )
    *p = '\0';
```

O fragmento neste exemplo define cada elemento de `array` como uma constante de caractere nulo.

```
enum color { red, white, green } col;
   .
   .
   .
   if ( col == red )
   .
   .
   .
```

Essas instruções declaram uma variável de enumeração chamada `col` com a marca `color` . A qualquer momento, a variável pode conter um valor inteiro de 0, 1, ou 2, que representa um dos elementos do conjunto de enumeração `color` : a cor vermelha, branca ou verde, respectivamente. Se `col` contiver 0 quando a `if` instrução for executada, todas as instruções dependendo do `if` serão executadas.

Veja também

Operadores relacionais: <, > , <=, and >=
Operadores de igualdade: = = e! =

Operadores bit a bit C

13/05/2021 • 2 minutes to read

Os operadores de bits bit a passo executam operações bit-a-AND (**&**), Exclusive-or (**^**) e bit-a-inclusivo-or (**|**).

Sintaxe

Expressão *and*- Expression: *igualdade*- expressão e expressão de **&** *igualdade* de expressão

expressão *Exclusive-ou-Expression*: *e*- Expression *Exclusive-ou*- Expression **^** *e-Expression*

inclusivo-ou- expressão: *Exclusive-ou* -expressão *inclusiva ou* expressão | *Exclusive-ou-Expression*

Os operandos dos operadores bit a bit devem ter tipos integrais, mas seus tipos podem ser diferentes. Esses operadores executam conversões aritméticas comuns; o tipo do resultado é o tipo dos operandos após a conversão.

Os operadores bit a bit de C são descritos abaixo:

OPERADOR	DESCRIÇÃO
&	O operador AND bit a bit compara cada bit do primeiro operando com o bit correspondente de seu 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).
^	O operador OR exclusivo bit a bit compara cada bit do primeiro operando ao bit correspondente do seu segundo operando. Se um bit for 0 e o outro bit for 1, o bit de resultado correspondente será definido como 1. Caso contrário, o bit de resultado correspondente é definido como zero (0).
|	O operador OR inclusivo bit a bit compara cada bit do 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).

Exemplos

Essas declarações são usadas para os três exemplos a seguir:

```
short i = 0xAB00;
short j = 0xABCD;
short n;

n = i & j;
```

O resultado atribuído a `n` neste primeiro exemplo é o mesmo que `i` (0xAB00 hexadecimal).

```
n = i | j;

n = i ^ j;
```

O operador OR inclusivo bit a bit no segundo exemplo resulta no valor 0xABCD (hexadecimal), enquanto o OU exclusivo bit a bit no terceiro exemplo gerencia 0xCD (hexadecimal).

Específico da Microsoft

Os resultados de operação bit a bit em números inteiros assinados é definido pela implementação de acordo com o padrão ANSI C. Para o compilador C da Microsoft, as operações bit a bit em números inteiros assinados funcionam da mesma forma que as operações bit a bit em inteiros não assinados. Por exemplo, `-16 & 99` pode ser expresso em binário como

```
  11111111 11110000
& 00000000 01100011
  ________________
  00000000 01100000
```

O resultado do E bit a bit é decimal 96.

FINAL específico da Microsoft

Veja também

Operador AND bit a bit: &
Operador OR exclusivo de OR bit: ^
Bitwise Inclusive OR Operator: |

Operadores lógicos C

13/05/2021 • 2 minutes to read

Os operadores lógicos executam operações lógicas-AND (**&&**) e or (**||**).

Sintaxe

expressão and lógica:
  inclusivo-ou-expressão
  expressão **&&** and lógica *inclusivo-ou-expressão*

expressão or lógica:
  expressão AND lógica
  expressão or *lógica* **||** *lógica e expressão*

Comentários

Os operadores lógicos não executam as conversões aritméticas comuns. Em vez disso, eles avaliam cada operando em termos de sua equivalência a 0. O resultado de uma operação lógica é 0 ou 1. O tipo do resultado é `int` .

Os operadores lógicos de C são descritos abaixo:

OPERADOR	DESCRIÇÃO
&&	O operador lógico AND produz o valor 1 se os dois operandos tiverem valores diferentes de zero. Se qualquer um dos operandos for igual a 0, o resultado será 0. Se o primeiro operando de uma operação AND lógica for igual a 0, o segundo operando não será avaliado.
||	O operador OR lógico executa uma operação OR inclusiva em seus operandos. O resultado é 0 se os dois operandos tiverem valores 0. Se qualquer um dos operando tiver um valor diferente de zero, o resultado será 1. Se o primeiro operando de uma operação OR lógica tiver um valor diferente de 0, o segundo operando não será avaliado.

Os operandos de expressões AND lógica e OR lógica são avaliados da esquerda para a direita. Se o valor do primeiro operando for suficiente para determinar o resultado da operação, o segundo operando não será avaliado. Isso é chamado "avaliação pelo caminho mais curto". Há um ponto de sequência depois do primeiro operando. Consulte Pontos de sequência para obter mais informações.

Exemplos

Os exemplos a seguir ilustram os operadores lógicos:

```
int w, x, y, z;

if ( x < y && y < z )
    printf( "x is less than z\n" );
```

Neste exemplo, a função **printf** é chamada para imprimir uma mensagem se `x` for menor que `y` e `y` for menor que `z`. Se `x` for maior que `y`, o segundo operando (`y < z`) não será avaliado e nada será impresso. Observe que isso pode causar problemas nos casos em que o segundo operando tiver efeitos colaterais que sejam usados por algum outro motivo.

```
printf( "%d" , (x == w || x == y || x == z) );
```

Nesse exemplo, se `x` for igual a `w`, `y` ou `z`, o segundo argumento para a função **printf** é avaliado como verdadeiro e o valor 1 é impresso. Caso contrário, será avaliado como false, e o valor 0 será impresso. Assim que uma das condições for avaliada como true, a avaliação é encerrada.

Veja também

- Operador AND lógico: &&
- Operador OR lógico: ||

Operador de expressão condicional

13/05/2021 • 2 minutes to read

C tem um operador ternário: o operador de expressão condicional (**?:**).

Sintaxe

expressão condicional:
  logical-OR-expression
  expressão logical-OR **?** *expression* **:** *expressão condicional*

A *logical-OR-expression* deve ter um tipo integral, de flutuação ou ponteiro. Ela é avaliada em termos de sua equivalência a 0. Um ponto de sequência vem após a *logical-OR-expression*. A avaliação dos operandos acontece da seguinte maneira:

- Se a *logical-OR-expression* não for igual a 0, a *expression* será avaliada. O resultado da avaliação da expressão é determinado pela *expression* não terminal. (Isso significa que a *expression* será avaliada apenas se a *logical-OR-expression* for verdadeira.)
- Se a *logical-OR-expression* for igual a 0, a *conditional-expression* será avaliada. O resultado da expressão será o valor da *conditional-expression*. (Isso significa que a *conditional-expression* será avaliada apenas se a *logical-OR-expression* for falsa.)

Observe que apenas a *expression* ou a *conditional-expression* é avaliada, mas não ambas.

O tipo de resultado de uma operação condicional depende do tipo do operando da *expression* ou *conditional-expression*, da seguinte maneira:

- Se a *expression* ou *conditional-expression* tiver o tipo integral ou flutuante (seus tipos podem ser diferentes), o operador executa conversões aritméticas comuns. O tipo do resultado é o tipo dos operandos após conversão.
- Se a *expression* e a *conditional-expression* tiverem a mesma estrutura, união ou tipo de ponteiro, o tipo de resultado é a mesma estrutura, união ou tipo de ponteiro.
- Se ambos os operandos tiverem tipo `void` , o resultado terá tipo `void` .
- Se qualquer operando for um ponteiro para um objeto de qualquer tipo, e o outro operando for um ponteiro para `void` , o ponteiro para o objeto será convertido em um ponteiro para `void` e o resultado será um ponteiro para `void` .
- Se a *expression* ou *conditional-expression* for um ponteiro e o outro operando for uma expressão constante com o valor 0, o tipo de resultado será o tipo de ponteiro.

Na comparação de tipos para ponteiros, qualquer qualificador de tipo (`const` ou `volatile`) no tipo ao qual os pontos apontadores são insignificantes, mas o tipo de resultado herda os qualificadores de ambos os componentes do condicional.

Exemplos

Os exemplos a seguir mostram o uso do operador condicional:

```
j = ( i < 0 ) ? ( -i ) : ( i );
```

Este exemplo atribui o valor absoluto de `i` a `j`. Se `i` for menor que 0, `-i` é atribuído a `j`. Se `i` for maior ou igual a 0, `i` é atribuído a `j`.

```
void f1( void );
void f2( void );
int x;
int y;
    .
    .
    .
( x == y ) ? ( f1() ) : ( f2() );
```

Neste exemplo, duas funções, `f1` e `f2`, e duas variáveis, `x` e `y`, são declaradas. Posteriormente no programa, se as duas variáveis tiverem o mesmo valor, a função `f1` será chamada. Caso contrário, `f2` é chamada.

Veja também

Operador condicional: ? :

Operadores de atribuição C

13/05/2021 • 2 minutes to read

Uma operação de atribuição atribui o valor do operando à direita para o local de armazenamento nomeado pelo operando à esquerda. Portanto, o operando à esquerda de uma operação de atribuição deve ser um valor l modificável. Após a atribuição, uma expressão de atribuição tem o valor do operando à esquerda mas não é um valor l.

Syntax

`assignment-expression` :
`conditional-expression`
`unary-expression` `assignment-operator` `assignment-expression`

`assignment-operator` : um dos
`=` `*=` `/=` `%=` `+=` `-=` `<<=` `>>=` `&=` `^=` `|=`

Os operadores de atribuição em C podem transformar e atribuir valores em uma única operação. O C fornece os seguintes operadores de atribuição:

OPERADOR	OPERAÇÃO EXECUTADA
`=`	Atribuição simples
`*=`	Atribuição de multiplicação
`/=`	Atribuição de divisão
`%=`	Atribuição restante
`+=`	Atribuição de adição
`-=`	Atribuição de subtração
`<<=`	Atribuição de shift esquerda
`>>=`	Atribuição de shift direita
`&=`	Atribuição AND bit a bit
`^=`	Atribuição OR exclusivo bit a bit
`|=`	Atribuição OR inclusivo bit a bit

Na atribuição, o tipo do valor à direita é convertido no tipo do valor à esquerda, e o valor é armazenado no operando à esquerda depois que a atribuição ocorreu. O operando à esquerda não deve ser uma matriz, uma função ou uma constante. O caminho específico de conversão, que depende dos dois tipos, é descrito em detalhes em Conversões de tipos.

Confira também

- Operadores de atribuição

Atribuição simples (C)

13/05/2021 • 2 minutes to read

O operador de atribuição simples atribui o operando à direita ao operando à esquerda. O valor do operando à direita é convertido no tipo da expressão de atribuição e substituir o valor armazenado no objeto designado pelo operando à esquerda. As regras de conversão para a atribuição se aplicam (consulte Conversões de atribuição).

```
double x;
int y;

x = y;
```

Neste exemplo, o valor de `y` é convertido em tipo `double` e atribuído a `x` .

Veja também

Operadores de atribuição de C

Atribuição composta C

13/05/2021 • 2 minutes to read

Os operadores de atribuição composta combinam o operador de atribuição simples com outro operador binário. Os operadores de atribuição composta executam a operação especificada pelo operador adicional e, em seguida, atribuem o resultado ao operando esquerdo. Por exemplo, uma expressão de atribuição composta, como

```
expression1 += expression2
```

pode ser entendida como

```
expression1 = expression1 + expression2
```

No entanto, a expressão de atribuição composta não é equivalente à versão expandida porque a expressão de atribuição composta avalia *expression1* apenas uma vez, enquanto a versão expandida avalia *expression1* duas vezes: na operação de adição e na operação de atribuição.

Os operandos de um operador de atribuição composta devem ser do tipo integral ou flutuação. Cada operador de atribuição composta executa as conversões que o operador binário correspondente executa e restringe os tipos de seus operandos de acordo. Os operadores adição-atribuição (`+=`) e subtração-atribuição (`-=`) também podem ter um operando esquerdo de tipo de ponteiro; nesse caso, o operando à direita deve ser do tipo integral. O resultado de uma operação de atribuição composta tem o valor e o tipo do operando esquerdo.

```
#define MASK 0xff00

n &= MASK;
```

Neste exemplo, a operação AND inclusivo bit a bit é executada em `n` e em `MASK`, e o resultado é atribuído a `n`. A constante de manifesto `MASK` é definida com uma diretiva do pré-processador #define.

Veja também

Operadores de atribuição de C

Operador de avaliação sequencial

13/05/2021 • 2 minutes to read

O operador de avaliação sequencial, também chamado de "operador vírgula", avalia seus dois operandos em sequência da esquerda para a direita.

Sintaxe

expressão:
  expressão de atribuição
  expressão , expressão *de atribuição*

O operando esquerdo do operador de avaliação sequencial é avaliado como uma `void` expressão. O resultado da operação tem o mesmo valor e tipo que o operando direito. Cada operando pode ser de qualquer tipo. O operador de avaliação sequencial não executa conversões de tipos entre seus operandos e não produz um l-value. Há um ponto de sequência depois do primeiro operando, o que significa que todos os efeitos colaterais da avaliação do operando esquerdo são concluídos antes de iniciar a avaliação do operando direito. Consulte Pontos de sequência para obter mais informações.

Geralmente, o operador de avaliação sequencial é usado para avaliar duas ou mais expressões em contextos em que apenas uma expressão é permitida.

As vírgulas podem ser usadas como separadores em alguns contextos. Porém, você deve ter cuidado para não confundir o uso da vírgula como separador com seu uso como operador; os dois usos são completamente diferentes.

Exemplo

Este exemplo ilustra o operador de avaliação sequencial:

```
for ( i = j = 1; i + j < 20; i += i, j-- );
```

Neste exemplo, cada operando da `for` terceira expressão da instrução é avaliado de forma independente. O operando esquerdo, `i += i` , é avaliado primeiro; em seguida, é a vez do operando direito, `j--` .

```
func_one( x, y + 2, z );
func_two( (x--, y + 2), z );
```

Na chamada de função para `func_one` , 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` .

Veja também

Operador de vírgula:,

Conversões de tipo (C)

13/05/2021 • 2 minutes to read

As conversões de tipos dependem do operador especificado e do tipo de operando ou dos operadores. As conversões de tipo são executadas nos seguintes casos:

- Quando um valor de um tipo é atribuído a uma variável de um tipo diferente ou um operador converte o tipo do seu operando ou operandos antes de executar uma operação
- Quando um valor de um tipo é explicitamente convertido em um tipo diferente
- Quando um valor é passado como um argumento para uma função ou quando um tipo é retornado de uma função

Um caractere, inteiro curto ou campo de bit de inteiro, todos assinados ou não, ou um objeto de tipo de enumeração, podem ser usados em uma expressão sempre que um inteiro puder ser usado. Se um `int` puder representar todos os valores do tipo original, o valor será convertido em `int` ; caso contrário, será convertido em `unsigned int` . Esse processo é chamado de "promoção de integral". As promoções de integral preservam os valores. Ou seja, o valor após a promoção tem a garantia de ser o mesmo que antes de promoção. Consulte Conversões aritméticas usuais para obter mais informações.

Veja também

Expressões e atribuições

Conversões de atribuição

13/05/2021 • 2 minutes to read

Em operações de atribuição, o tipo do valor que está sendo atribuído é convertido no tipo da variável que recebe a atribuição. O C permite conversões por atribuição entre os tipos integrais e flutuantes, mesmo se as informações forem perdidas na conversão. O método de conversão usado depende dos tipos envolvidos na atribuição, conforme descrito em Conversões aritméticas comuns e nas seguintes seções:

- Conversões de tipos integrais com sinal
- Conversões de tipos integrais sem sinal
- Conversões de tipos de ponto flutuante
- Conversões de e em tipos de ponteiro
- Conversões de outros tipos

Os qualificadores de tipo não afetam a permissão da conversão, embora um `const` l-value não possa ser usado no lado esquerdo da atribuição.

Veja também

Conversões de tipo

Conversões de tipos integrais com sinal

13/05/2021 • 3 minutes to read

Quando um inteiro assinado é convertido em um número inteiro ou em um tipo de ponto flutuante, se o valor original for representável no tipo de resultado, o valor será inalterado.

Quando um inteiro assinado é convertido em um número inteiro de tamanho maior, o valor é Sign-Extended. Quando convertido para um inteiro de tamanho menor, os bits de ordem superior são truncados. O resultado é interpretado usando o tipo de resultado, conforme mostrado neste exemplo:

```
int i = -3;
unsigned short u;

u = i;
printf_s( "%hu\n", u );  // Prints 65533
```

Ao converter um inteiro assinado em um tipo de ponto flutuante, se o valor original não for reapresentável exatamente no tipo de resultado, o resultado será o próximo valor reapresentável mais alto ou mais baixo.

Para obter informações sobre os tamanhos de tipos de ponto flutuante e integral, consulte armazenamento de tipos básicos.

A tabela a seguir resume as conversões de tipos integrais com sinal. Ele pressupõe que o `char` tipo é assinado por padrão. Se você usar uma opção de tempo de compilação para alterar o padrão do `char` tipo para não assinado, as conversões fornecidas na tabela conversões de tipos de integral não assinados para o `unsigned char` tipo se aplicam, em vez das conversões nesta tabela.

Específico da Microsoft

No compilador da Microsoft, `int` e `long` são tipos distintos, mas equivalentes. A conversão de um `int` valor prossegue da mesma maneira que a conversão de um `long` .

FINAL específico da Microsoft

Tabela de conversões de tipos integrais assinados

DE	PARA	MÉTODO
`char` [uma]	`short`	Extensão de sinal
`char`	`long`	Extensão de sinal
`char`	`long long`	Extensão de sinal
`char`	`unsigned char`	Preserva o padrão; o bit de ordem superior perde a função como bit de sinal
`char`	`unsigned short`	Assinar para `short` ; Converter `short` em `unsigned short`

DE	PARA	MÉTODO
`char`	`unsigned long`	Assinar para `long` ; Converter `long` em `unsigned long`
`char`	`unsigned long long`	Assinar para `long long` ; Converter `long long` em `unsigned long long`
`char`	`float`	Assinar para `long` ; Converter `long` em `float`
`char`	`double`	Assinar para `long` ; Converter `long` em `double`
`char`	`long double`	Assinar para `long` ; Converter `long` em `double`
`short`	`char`	Preserva o byte de ordem inferior
`short`	`long`	Extensão de sinal
`short`	`long long`	Extensão de sinal
`short`	`unsigned char`	Preserva o byte de ordem inferior
`short`	`unsigned short`	Preserva o padrão de bits; o bit de ordem superior perde a função como bit de sinal
`short`	`unsigned long`	Assinar para `long` ; Converter `long` em `unsigned long`
`short`	`unsigned long long`	Assinar para `long long` ; Converter `long long` em `unsigned long long`
`short`	`float`	Assinar para `long` ; Converter `long` em `float`
`short`	`double`	Assinar para `long` ; Converter `long` em `double`
`short`	`long double`	Assinar para `long` ; Converter `long` em `double`
`long`	`char`	Preserva o byte de ordem inferior
`long`	`short`	Preserva a palavra de ordem inferior
`long`	`long long`	Extensão de sinal

DE	PARA	MÉTODO
`long`	`unsigned char`	Preserva o byte de ordem inferior
`long`	`unsigned short`	Preserva a palavra de ordem inferior
`long`	`unsigned long`	Preserva o padrão de bits; o bit de ordem superior perde a função como bit de sinal
`long`	`unsigned long long`	Assinar para `long long` ; Converter `long long` em `unsigned long long`
`long`	`float`	Representar como `float` . Se `long` não puder ser representado exatamente, alguma precisão será perdida.
`long`	`double`	Representar como `double` . Se `long` não puder ser representado exatamente como um `double` , alguma precisão será perdida.
`long`	`long double`	Representar como `double` . Se `long` não puder ser representado exatamente como um `double` , alguma precisão será perdida.
`long long`	`char`	Preserva o byte de ordem inferior
`long long`	`short`	Preserva a palavra de ordem inferior
`long long`	`long`	Preservar DWORD de ordem inferior
`long long`	`unsigned char`	Preserva o byte de ordem inferior
`long long`	`unsigned short`	Preserva a palavra de ordem inferior
`long long`	`unsigned long`	Preservar DWORD de ordem inferior
`long long`	`unsigned long long`	Preserva o padrão de bits; o bit de ordem superior perde a função como bit de sinal
`long long`	`float`	Representar como `float` . Se `long long` não puder ser representado exatamente, alguma precisão será perdida.

DE	PARA	MÉTODO
`long long`	`double`	Representar como `double` . Se `long long` não puder ser representado exatamente como um `double` , alguma precisão será perdida.
`long long`	`long double`	Representar como `double` . Se `long long` não puder ser representado exatamente como um `double` , alguma precisão será perdida.

[1] todas as `char` entradas pressupõem que o `char` tipo é assinado por padrão.

Veja também

Conversões de atribuição

Conversões de tipos integrais sem sinal

13/05/2021 • 3 minutes to read

Quando um inteiro sem sinal é convertido em um tipo inteiro ou de ponto flutuante, se o valor original for representável no tipo de resultado, o valor será inalterado.

Ao converter um inteiro não assinado em um inteiro de tamanho maior, o valor será estendido como zero. Ao converter para um inteiro de tamanho menor, os bits de ordem superior são truncados. O resultado é interpretado usando o tipo de resultado, conforme mostrado neste exemplo.

```
unsigned k = 65533;
short j;

j = k;
printf_s( "%hd\n", j );   // Prints -3
```

Ao converter um inteiro não assinado em um tipo de ponto flutuante, se o valor original não puder ser representado exatamente no tipo de resultado, o resultado será o próximo valor reapresentável mais alto ou mais baixo.

Consulte armazenamento de tipos básicos para obter informações sobre os tamanhos de tipos de ponto flutuante e integral.

Específico da Microsoft

No compilador da Microsoft, `unsigned` (ou `unsigned int`) e `unsigned long` são tipos distintos, mas equivalentes. A conversão de um `unsigned int` valor prossegue da mesma maneira que a conversão de um `unsigned long` .

FINAL específico da Microsoft

A tabela a seguir resume as conversões de tipos integrais sem sinal.

Tabela de conversões de tipos integrais não assinados

DE	PARA	MÉTODO
`unsigned char`	`char`	Preserva o padrão de bits; o bit de ordem superior torna-se o bit de sinal
`unsigned char`	`short`	Extensão de zero
`unsigned char`	`long`	Extensão de zero
`unsigned char`	`long long`	Extensão de zero
`unsigned char`	`unsigned short`	Extensão de zero
`unsigned char`	`unsigned long`	Extensão de zero
`unsigned char`	`unsigned long long`	Extensão de zero

DE	PARA	MÉTODO
`unsigned char`	`float`	Converter em `long` ; Converter `long` em `float`
`unsigned char`	`double`	Converter em `long` ; Converter `long` em `double`
`unsigned char`	`long double`	Converter em `long` ; Converter `long` em `double`
`unsigned short`	`char`	Preserva o byte de ordem inferior
`unsigned short`	`short`	Preserva o padrão de bits; o bit de ordem superior torna-se o bit de sinal
`unsigned short`	`long`	Extensão de zero
`unsigned short`	`long long`	Extensão de zero
`unsigned short`	`unsigned char`	Preserva o byte de ordem inferior
`unsigned short`	`unsigned long`	Extensão de zero
`unsigned short`	`unsigned long long`	Extensão de zero
`unsigned short`	`float`	Converter em `long` ; Converter `long` em `float`
`unsigned short`	`double`	Converter em `long` ; Converter `long` em `double`
`unsigned short`	`long double`	Converter em `long` ; Converter `long` em `double`
`unsigned long`	`char`	Preserva o byte de ordem inferior
`unsigned long`	`short`	Preserva a palavra de ordem inferior
`unsigned long`	`long`	Preserva o padrão de bits; o bit de ordem superior torna-se o bit de sinal
`unsigned long`	`long long`	Extensão de zero
`unsigned long`	`unsigned char`	Preserva o byte de ordem inferior
`unsigned long`	`unsigned short`	Preserva a palavra de ordem inferior
`unsigned long`	`unsigned long long`	Extensão de zero
`unsigned long`	`float`	Converter em `long` ; Converter `long` em `float`

DE	PARA	MÉTODO
`unsigned long`	`double`	Converter diretamente em `double`
`unsigned long`	`long double`	Converter em `long` ; Converter `long` em `double`
`unsigned long long`	`char`	Preserva o byte de ordem inferior
`unsigned long long`	`short`	Preserva a palavra de ordem inferior
`unsigned long long`	`long`	Preservar DWORD de ordem inferior
`unsigned long long`	`long long`	Preserva o padrão de bits; o bit de ordem superior torna-se o bit de sinal
`unsigned long long`	`unsigned char`	Preserva o byte de ordem inferior
`unsigned long long`	`unsigned short`	Preserva a palavra de ordem inferior
`unsigned long long`	`unsigned long`	Preservar DWORD de ordem inferior
`unsigned long long`	`float`	Converter em `long` ; Converter `long` em `float`
`unsigned long long`	`double`	Converter diretamente em `double`
`unsigned long long`	`long double`	Converter em `long` ; Converter `long` em `double`

Veja também

Conversões de atribuição

Conversões de tipos de ponto flutuante

13/05/2021 • 4 minutes to read

Um valor de ponto flutuante que é convertido em outro tipo de ponto flutuante não passa por nenhuma alteração no valor se o valor original é representável exatamente no tipo de resultado. Se o valor original for numérico, mas não for reapresentável exatamente, o resultado será o maior valor de reapresentável mais próximo ou próximo. Veja limites em constantes de ponto flutuante para o intervalo de tipos de ponto flutuante.

Um valor de ponto flutuante que é convertido em um tipo integral é truncado primeiro, descartando qualquer valor fracionário. Se esse valor truncado for representável no tipo de resultado, o resultado deverá ser esse valor. Quando não é possível representá-lo, o valor do resultado é indefinido.

Específico da Microsoft

Os compiladores da Microsoft usam a representação binary32 do IEEE-754 para `float` valores e a representação binary64 para `long double` e `double` . Como `long double` e `double` usam a mesma representação, elas têm o mesmo intervalo e precisão.

Quando o compilador converte um `double` `long double` número de ponto flutuante ou em um `float` , ele arredonda o resultado de acordo com os controles de ambiente de ponto flutuante, cujo padrão é "arredondar para mais próximo, amarra até mesmo". Se um valor numérico for muito alto ou muito baixo para ser representado como um `float` valor numérico, o resultado da conversão será um infinito positivo ou negativo, de acordo com o sinal do valor original, e uma exceção de estouro será gerada, se habilitado.

Ao converter em tipos inteiros, o resultado de uma conversão para um tipo menor do que `long` é o resultado da conversão do valor para `long` e, em seguida, a conversão para o tipo de resultado.

Para conversão em tipos inteiros, pelo menos tão grande quanto `long` , uma conversão de um valor muito alto ou muito baixo para representar no tipo de resultado pode retornar qualquer um dos seguintes valores:

- O resultado pode ser um *valor de sentinela*, que é o valor representável mais distante de zero. Para tipos assinados, é o menor valor representável (0x800... 0). Para tipos não assinados, é o valor mais alto representável (0xFF... F).
- O resultado pode ser *saturado*, onde os valores muito altos para representar são convertidos para o valor mais alto representável, e valores muito baixos para serem representados são convertidos para o valor representável mais baixo. Um desses dois valores também é usado como o valor de sentinela.
- Para a conversão para `unsigned long` ou `unsigned long long` , o resultado da conversão de um valor fora do intervalo pode ser um valor diferente do valor mais alto ou mais baixo representável. Se o resultado é um sentinela ou um valor saturado ou não depende das opções do compilador e da arquitetura de destino. Versões futuras do compilador podem retornar um valor saturado ou de sentinela.

FINAL específico da Microsoft

A tabela a seguir resume as conversões de tipos flutuantes.

Tabela de conversões de tipos de ponto flutuante

DE	PARA	MÉTODO

DE	PARA	MÉTODO
`float`	`char`	Converter em `long` ; Converter `long` em `char`
`float`	`short`	Converter em `long` ; Converter `long` em `short`
`float`	`int`	Truncar no ponto decimal. Se o resultado for muito grande para ser representado como `int` , o resultado será indefinido.
`float`	`long`	Truncar no ponto decimal. Se o resultado for muito grande para ser representado como `long` , o resultado será indefinido.
`float`	`long long`	Truncar no ponto decimal. Se o resultado for muito grande para ser representado como `long long` , o resultado será indefinido.
`float`	`unsigned char`	Converter em `long` ; Converter `long` em `unsigned char`
`float`	`unsigned short`	Converter em `long` ; Converter `long` em `unsigned short`
`float`	`unsigned`	Truncar no ponto decimal. Se o resultado for muito grande para ser representado como `unsigned` , o resultado será indefinido.
`float`	`unsigned long`	Truncar no ponto decimal. Se o resultado for muito grande para ser representado como `unsigned long` , o resultado será indefinido.
`float`	`unsigned long long`	Truncar no ponto decimal. Se o resultado for muito grande para ser representado como `unsigned long long` , o resultado será indefinido.
`float`	`double`	Representar como um `double` .
`float`	`long double`	Representar como um `long double` .
`double`	`char`	Converter em `float` ; Converter `float` em `char`
`double`	`short`	Converter em `float` ; Converter `float` em `short`

DE	PARA	MÉTODO
`double`	`int`	Truncar no ponto decimal. Se o resultado for muito grande para ser representado como `int`, o resultado será indefinido.
`double`	`long`	Truncar no ponto decimal. Se o resultado for muito grande para ser representado como `long`, o resultado será indefinido.
`double`	`unsigned char`	Converter em `long`; Converter `long` em `unsigned char`
`double`	`unsigned short`	Converter em `long`; Converter `long` em `unsigned short`
`double`	`unsigned`	Truncar no ponto decimal. Se o resultado for muito grande para ser representado como `unsigned`, o resultado será indefinido.
`double`	`unsigned long`	Truncar no ponto decimal. Se o resultado for muito grande para ser representado como `unsigned long`, o resultado será indefinido.
`double`	`unsigned long long`	Truncar no ponto decimal. Se o resultado for muito grande para ser representado como `unsigned long long`, o resultado será indefinido.
`double`	`float`	Representar como um `float`. Se `double` o valor não puder ser representado exatamente como `float`, ocorre a perda de precisão. Se o valor for muito grande para ser representado como `float`, o resultado será indefinido.
`double`	`long double`	O `long double` valor é tratado como `double`.

As conversões de `long double` seguem o mesmo método que as conversões do `double`.

Veja também

Conversões de atribuição

Conversões em e de tipos de ponteiro

13/05/2021 • 2 minutes to read

Um ponteiro para um tipo de valor que pode ser convertido em um ponteiro para um tipo diferente. No entanto, o resultado pode ser indefinido devido aos requisitos de alinhamento e aos tamanhos de tipos diferentes em armazenamento. Um ponteiro para um objeto pode ser convertido em um ponteiro para um objeto cujo tipo exige menos alinhamento de armazenamento ou igualmente restrito, e reconverter sem modificação.

Um ponteiro para `void` pode ser convertido de ou para um ponteiro para qualquer tipo, sem restrição ou perda de informações. Se o resultado for reconvertido no tipo original, o ponteiro original será recuperado.

Se um ponteiro é convertido em outro ponteiro do mesmo tipo, mas com qualificadores diferentes ou adicionais, o novo ponteiro é o mesmo que o antigo, com a exceção das restrições impostas pelo novo qualificador.

Um valor de ponteiro pode ser convertido em um valor integral. O caminho de conversão depende do tamanho do ponteiro e do tamanho do tipo integral, de acordo com as seguintes regras:

- Se o tamanho do ponteiro for maior ou igual ao tamanho do tipo integral, o ponteiro se comportará como um valor sem sinal na conversão, mas não poderá ser convertido em um valor flutuante.
- Se o ponteiro for menor que o tipo integral, o ponteiro primeiro será convertido em um ponteiro do mesmo tamanho do tipo integral, e depois convertido no tipo integral.

Inversamente, um tipo integral pode ser convertido em um tipo de ponteiro de acordo com as seguintes regras:

- Se o tipo integral for do mesmo tamanho que o tipo de ponteiro, a conversão faz com que o valor integral seja tratado como um ponteiro (um número inteiro sem sinal).
- Se o tamanho do tipo integral for diferente do tamanho do tipo de ponteiro, o tipo integral é convertido primeiro no tamanho do ponteiro, usando os caminhos de conversão fornecidos nas tabelas Conversão de tipos integrais com sinal e Conversão de tipos integrais sem sinal. Depois é tratado como um valor do ponteiro.

Uma expressão constante integral com valor 0 ou tal conversão de expressão para tipo `void` * pode ser convertida por uma conversão de tipo, por atribuição ou por comparação a um ponteiro de qualquer tipo. Isso gera um ponteiro nulo que é igual a outro ponteiro nulo do mesmo tipo, mas esse ponteiro nulo não é igual a nenhum ponteiro para uma função ou a um objeto. Os números inteiros diferentes da constante 0 podem ser convertidos no tipo de ponteiro, mas o resultado não é portátil.

Veja também

Conversões de atribuição

Conversões de outros tipos

13/05/2021 • 2 minutes to read

Como um `enum` valor é um `int` valor por definição, conversões de e para um `enum` valor são as mesmas do `int` tipo. Para o compilador do Microsoft C, um inteiro é o mesmo que um `long` .

Específico da Microsoft

Nenhuma conversão entre os tipos de estrutura ou união é permitida.

Qualquer valor pode ser convertido para `void` o tipo, mas o resultado de tal conversão pode ser usado somente em um contexto em que um valor de expressão é Descartado, como em uma instrução de expressão.

O `void` tipo não tem valor, por definição. Portanto, ele não pode ser convertido em nenhum outro tipo, e outros tipos não podem ser convertidos `void` por atribuição. No entanto, você pode converter explicitamente um valor para tipo `void` , como discutido em conversões de conversão de tipo.

FINAL específico da Microsoft

Veja também

Conversões de atribuição

Conversões de conversão de tipo

13/05/2021 • 2 minutes to read

Você pode usar conversões de tipos em estilo "cast" para converter tipos explicitamente.

Sintaxe

expressão CAST:
  expressão unário
  expressão CAST (***nome-do-tipo***)

nome do tipo:
  specifier-qualifier-list abstract-declarator$_{opt}$

O *type-name* é um tipo e *cast-expression* é um valor para ser convertido para esse tipo. Uma expressão com uma conversão de tipo ("cast") não é um l- value. A *cast-expression* é convertida como se tivesse sido atribuída a uma variável do tipo *type-name*. As regras de conversão para atribuições (descritas em Conversões de atribuição) também se aplicam às conversões de tipos. A tabela a seguir mostra os tipos que podem ser convertidos em qualquer tipo indicado.

Conversões de tipos válidas

TIPOS DE DESTINO	POSSÍVEIS FONTES
Tipos integrais	Qualquer tipo de inteiro ou tipo de ponto flutuante, ou ponteiro para um objeto
Ponto flutuante	Qualquer tipo aritmético
Um ponteiro para um objeto ou (`void` *)	Qualquer tipo inteiro, (`void` *), um ponteiro para um objeto ou um ponteiro de função
Ponteiro de função	Qualquer tipo integral, um ponteiro para um objeto ou um ponteiro de função
Uma estrutura, união ou matriz	Nenhum
Tipo void	Qualquer tipo

Qualquer identificador pode ser convertido para o `void` tipo. No entanto, se o tipo especificado em uma expressão de conversão de tipo não for `void` , o identificador que está sendo convertido para esse tipo não pode ser uma `void` expressão. Qualquer expressão pode ser convertida `void` , mas uma expressão do tipo `void` não pode ser convertida em nenhum outro tipo. Por exemplo, uma função com `void` tipo de retorno não pode ter sua conversão de retorno para outro tipo.

Observe que uma `void` * expressão tem um ponteiro de tipo para `void` , não tipo `void` . Se um objeto for convertido para `void` o tipo, a expressão resultante não poderá ser atribuída a nenhum item. Do mesmo modo, um objeto de conversão de tipo não é um l-value aceitável; portanto, nenhuma atribuição pode ser feita a um objeto de conversão de tipo.

Específico da Microsoft

Uma conversão de tipo pode ser uma expressão l-value, desde que o tamanho do identificador não se altere.

Para obter informações sobre expressões l-value, consulte Expressões L-Value e R-Value.

FINAL específico da Microsoft

Você pode converter uma expressão para tipo `void` com uma conversão, mas a expressão resultante só pode ser usada quando um valor não é necessário. Um ponteiro de objeto convertido para `void` * e retornado para o tipo original retornará ao seu valor original.

Veja também

Conversões de tipo

Conversões de função de chamada

13/05/2021 • 2 minutes to read

O tipo de conversão executado nos argumentos em uma chamada de função depende da presença de um protótipo de função (declaração de encaminhamento) com tipos de argumentos declarados para a função chamada.

Se um protótipo da função estiver presente e incluir tipos de argumentos declarados, o compilador fará a verificação de tipo (consulte Funções).

Se nenhum protótipo da função estiver presente, somente as conversões aritméticas comuns serão executadas nos argumentos na chamada da função. Essas conversões são executadas independentemente em cada argumento na chamada. Isso significa que um `float` valor é convertido em a `double` ; um `char` ou `short` valor é convertido em um `int` ; e um `unsigned char` ou `unsigned short` é convertido em um `unsigned int` .

Veja também

Conversões de tipo

Instruções (C)

13/05/2021 • 2 minutes to read

As instruções de um programa de C controlam o fluxo de execução do programa. Em C, como nas outras linguagens de programação, vários tipos de instruções estão disponíveis para executar loops, selecionar a execução de outras instruções e transferir o controle. Depois de uma pequena visão geral da sintaxe da instrução, esta seção descreve as instruções de C, em ordem alfabética:

instrução break
instrução composta
instrução Continue
instrução do-while
instrução de expressão

instrução for
instruções GoTo e rotuladas
instrução If
instrução NULL
instrução de retorno

instrução switch
instrução try-Except
instrução try – finally
Instrução while

Veja também

Referência da linguagem C

Visão geral de instruções C

13/05/2021 • 2 minutes to read

As instruções C consistem em tokens, expressões e outras instruções. Uma instrução que forma um componente de outra instrução chama-se o "corpo" da instrução pertinente. Cada tipo de instrução fornecido pela sintaxe a seguir será discutido nesta seção.

Sintaxe

statement: labeled-statement

compound-statement

expression-statement

instrução de seleção

iteration-statement

jump-statement

try-Except-Statement /* específico da Microsoft */

/ instrução * try – finally Específico da Microsoft*/

Geralmente, o corpo da instrução é uma "instrução composta". Uma instrução composta consiste em outras instruções que podem incluir palavras-chave. A instrução compound é delimitada por chaves (**{ }**). Todas as outras instruções C terminam com ponto-e-vírgula (**;**). O ponto-e-vírgula é um terminador de instrução.

A instrução da expressão contém a expressão C que pode conter operadores aritméticos ou lógicos introduzidos em Expressões e atribuições. A instrução null é uma instrução vazia.

Qualquer instrução C pode começar com um rótulo de identificação que consiste em um nome e dois-pontos. Como apenas a `goto` instrução reconhece os rótulos de instrução, os rótulos de instrução são discutidos com `goto` . Consulte as instruções goto e labeled para obter mais informações.

Veja também

Instruções

Instrução break (C)

13/05/2021 • 2 minutes to read

A `break` instrução encerra a execução da instrução,,, ou delimitadora mais próxima `do` `for` `switch` `while` na qual ela aparece. O controle passa para a instrução que segue a instrução encerrada.

Sintaxe

instrução de salto:
break ;

A `break` instrução é frequentemente usada para encerrar o processamento de um caso específico dentro de `switch` uma instrução. A falta de uma instrução ou iterativa delimitadora `switch` gera um erro.

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 em outro lugar da estrutura aninhada.

Este exemplo ilustra a `break` instrução:

```
#include <stdio.h>
int main() {
   char c;
   for(;;) {
      printf_s( "\nPress any key, Q to quit: " );

      // Convert to character value
      scanf_s("%c", &c);
      if (c == 'Q')
          break;
   }
} // Loop exits only when 'Q' is pressed
```

Veja também

Instrução break

Instrução composta (C)

13/05/2021 • 2 minutes to read

Uma instrução composta (também chamada de "bloquear") normalmente aparece como o corpo de outra instrução, como a `if` instrução. Declarações e tipos descreve o formato e o significado das declarações que podem aparecer no cabeçalho de uma instrução composta.

Sintaxe

instrução composta:
  { *declaração-lista* de instruções $_{opt}$ *-lista*$_{opt}$ **}**

declaration-list:
  mesma
  declaration-list declaration

statement-list:
  privacidade
  statement-list statement

Se houver declarações, elas deverão vir antes de todas as instruções. O escopo de cada identificador declarado no início de uma instrução composta estende-se de seu ponto de declaração ao fim do bloco. É visível em todo o bloco a menos que uma declaração do mesmo identificador exista em um bloco interno.

Os identificadores em uma instrução composta são presumidos, `auto` a menos que sejam declarados explicitamente de outra forma com `register` `static` as funções,, ou, `extern` exceto, que só podem ser `extern` . Você pode deixar o `extern` especificador em declarações de função e a função ainda será `extern` .

O armazenamento não é alocado e a inicialização não é permitida se uma variável ou função é declarada em uma instrução composta com classe de armazenamento `extern` . A declaração se refere a uma variável ou função externa definida em outro lugar.

As variáveis declaradas em um bloco com a `auto` `register` palavra-chave ou são realocadas e, se necessário, são inicializadas sempre que a instrução composta é inserida. Essas variáveis não são definidas depois que a instrução composta é encerrada. Se uma variável declarada dentro de um bloco tiver o `static` atributo, a variável será inicializada quando a execução do programa for iniciada e manterá seu valor em todo o programa. Consulte classes de armazenamento para obter informações sobre o `static` .

Este exemplo ilustra uma instrução composta:

```
if ( i > 0 )
{
    line[i] = x;
    x++;
    i--;
}
```

Neste exemplo, se `i` for maior que 0, todas as instruções na instrução composta serão executadas na ordem.

Veja também

Instruções

`continue` instrução (C)

13/05/2021 • 2 minutes to read

A `continue` instrução passa o controle para a próxima iteração da instrução„ ou delimitadora mais próxima, `do` `for` `while` na qual ele aparece, ignorando as instruções restantes no `do` `for` corpo da instrução, ou `while` .

Sintaxe

`jump-statement`:
`continue ;`

A próxima iteração de `do` uma `for` instrução, ou `while` é determinada da seguinte maneira:

- Dentro de uma `do` `while` instrução ou, a próxima iteração começa reavaliando a expressão da `do` instrução or `while` .
- Uma `continue` instrução em uma `for` instrução causa a avaliação da expressão de loop da `for` instrução. Em seguida, o código reavalia a expressão condicional. Dependendo do resultado, ele termina ou itera o corpo da instrução. Para obter mais informações sobre a `for` instrução e seus não terminais, consulte a `for` instrução.

Veja um exemplo da `continue` instrução:

```
while ( i-- > 0 )
{
    x = f( i );
    if ( x == 1 )
        continue;
    y += x * x;
}
```

Nesse exemplo, o corpo da instrução é executado enquanto `i` é maior que 0. `f(i)` O primeiro é atribuído a `x` ; em seguida, se `x` for igual a 1, a `continue` instrução será executada. O restante das instruções no corpo é ignorado. A execução é retomada na parte superior do loop com a avaliação do teste do loop.

Confira também

`continue` instrução (C++)

Instrução do-while (C)

13/05/2021 • 2 minutes to read

A instrução *do-while* permite que você repita uma instrução ou instrução composta até que uma expressão especificada seja falsa.

Sintaxe

instrução *Iteration*: `do` *instrução* **while (*expressão*);**

A *expressão* em uma instrução *do-while* é avaliada depois que o corpo do loop é executado. Portanto, o corpo do loop é sempre executado ao menos uma vez.

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.

A instrução *do-while* também pode terminar quando uma `break` `goto` instrução, ou `return` for executada dentro do corpo da instrução.

Esse é um exemplo da instrução *do-while*:

```
do
{
    y = f( x );
    x--;
} while ( x > 0 );
```

Nesta instrução *do-while*, as duas instruções `y = f( x );` e `x--;` são executadas, independentemente do valor inicial de `x`. Em seguida, `x > 0` é avaliado. Se `x` for maior que 0, o corpo da instrução é executado novamente e `x > 0` é reavaliado. O corpo da instrução é executado repetidamente, enquanto `x` permanece maior que 0. A execução da instrução *do-while* termina quando `x` se torna 0 ou negativo. O corpo do loop é executado ao menos uma vez.

Veja também

Instrução do-while (C++)

Instrução de expressão (C)

13/05/2021 • 2 minutes to read

Quando uma instrução de expressão é executada, ela é avaliada de acordo com as regras descritas em Expressões e atribuições.

Sintaxe

expressão-instrução:
  expression$_{opt}$;

Todos os efeitos colaterais de avaliação da expressão são concluídos antes de a próxima instrução ser executada. Uma instrução expression vazia é chamada de instrução null. Consulte A instrução null para obter mais informações.

Estes exemplos demonstram instruções expression.

```
x = ( y + 3 );            /* x is assigned the value of y + 3  */
x++;                      /* x is incremented                  */
x = y = 0;                /* Both x and y are initialized to 0 */
proc( arg1, arg2 );       /* Function call returning void      */
y = z = ( f( x ) + 3 );   /* A function-call expression        */
```

Na última instrução, a expressão de chamada de função, o valor da expressão, que inclui os valores retornados pela função, é aumentada em 3 e atribuída às duas variáveis `y` e `z`.

Veja também

Instruções

Instrução for (C)

13/05/2021 • 2 minutes to read

A `for` instrução permite repetir uma instrução ou instrução composta um número especificado de vezes. O corpo de uma `for` instrução é executado zero ou mais vezes até que uma condição opcional se torne falsa. Você pode usar expressões opcionais dentro da `for` instrução para inicializar e alterar valores durante a `for` execução da instrução.

Sintaxe

instrução Iteration:
`for` **** (*init-expressão*$_{opt}$; *condicional-expressão*$_{opt}$; *instrução loop-expressão*$_{opt}$)

A execução de uma `for` instrução continua da seguinte maneira:

1. A *init-expression*, se houver, é avaliada. Isso especifica a inicialização do loop. Não há nenhuma restrição quanto ao tipo da *init-expression*.

2. A *cond-expression*, se houver, é avaliada. Essa expressão deve ter tipo aritmético ou ponteiro. Ela é avaliada antes de cada iteração. Três resultados são possíveis:

   - Se *condicional* for `true` (diferente de zero), a *instrução* será executada; em seguida, *a expressão de loop*, se houver, será avaliada. A *loop-expression* é avaliada após cada iteração. Não há nenhuma restrição quanto ao tipo dele. Os efeitos colaterais serão executados na ordem. Em seguida, o processo é iniciado novamente com a avaliação de *cond-expression*.

   - Se *cond-expression* está omitida, *cond-expression* é considerada true e a execução continua exatamente conforme descrita no parágrafo anterior. Uma `for` instrução sem um argumento *condicional* é encerrada somente quando uma `break` instrução ou `return` dentro do corpo da instrução é executada ou quando um `goto` (para uma instrução rotulada fora do corpo da `for` instrução) é executado.

   - Se *condicional* for `false` (0), a execução da instrução será `for` encerrada e o controle passará para a próxima instrução no programa.

Uma `for` instrução também termina quando uma `break` instrução, `goto` ou `return` dentro do corpo da instrução é executada. Uma `continue` instrução em um `for` loop faz com que a *expressão de loop* seja avaliada. Quando uma `break` instrução é executada dentro de um `for` loop, a *expressão de loop* não é avaliada ou executada. Esta instrução

```
for( ; ; )
```

é a maneira personalizada de produzir um loop infinito que só pode ser encerrado com uma `break` instrução, `goto` ou `return` .

Exemplo

Este exemplo ilustra a `for` instrução:

```
// c_for.c
int main()
{
   char* line = "H e  \tl\tlo World\0";
   int space = 0;
   int tab = 0;
   int i;
   int max = strlen(line);
   for (i = 0; i < max; i++ )
   {
      if ( line[i] == ' ' )
      {
          space++;
      }
      if ( line[i] == '\t' )
      {
          tab++;
      }
   }

   printf("Number of spaces: %i\n", space);
   printf("Number of tabs: %i\n", tab);
   return 0;
}
```

Saída

```
Number of spaces: 4
Number of tabs: 2
```

Veja também

Instruções

Instruções goto e identificadas (C)

13/05/2021 • 2 minutes to read

A `goto` instrução transfere o controle para um rótulo. O rótulo fornecido deve residir na mesma função e pode aparecer antes de apenas uma instrução na mesma função.

Sintaxe

instrução:
  instrução rotulada
  instrução de salto

instrução de salto:
  `goto` *identificador* ;

rotulado-instrução:
  identificador : *instrução*

Um rótulo de instrução é significativo apenas para uma `goto` instrução; em qualquer outro contexto, uma instrução rotulada é executada sem considerar o rótulo.

Um elemento *jump-statement* deve residir na mesma função e pode aparecer antes de apenas uma instrução na mesma função. O conjunto de nomes de *identificadores* a seguir `goto` tem seu próprio espaço de nome para que os nomes não interfiram com outros identificadores. Os rótulos não podem ser redeclarados. Consulte Namespaces para obter mais informações.

É um bom estilo de programação usar a `break` `continue` instrução, e `return` em preferência, `goto` sempre que possível. Como a `break` instrução só sai de um nível do loop, uma `goto` pode ser necessária para sair de um loop de dentro de um loop profundamente aninhado.

Este exemplo demonstra a `goto` instrução:

```
// goto.c
#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 < 3; j++ )
        {
            printf_s( " Inner loop executing. j = %d\n", j );
            if ( i == 5 )
                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 );
}
```

Neste exemplo, uma `goto` instrução transfere o controle para o ponto rotulado `stop` quando `i` é igual a 5.

Veja também

Instruções

Instrução if (C)

13/05/2021 • 2 minutes to read

A `if` instrução controla a ramificação condicional. O corpo de uma `if` instrução será executado se o valor da expressão for diferente de zero. A sintaxe da `if` instrução tem duas formas.

Sintaxe

instrução *de seleção*: instrução **If (*expression*)**

instrução de *instrução* **If (*expression*)** `else`

Em ambas as formas da `if` instrução, as expressões, que podem ter qualquer valor, exceto uma estrutura, são avaliadas, incluindo todos os efeitos colaterais.

No primeiro formato da sintaxe, se a *expressão* for verdadeira (diferente de zero), a *instrução* será executada. Se a *expressão* for falsa, a *instrução* será ignorada. Na segunda forma de sintaxe, que usa `else` , a segunda *instrução* é executada se *expression* for false. Com ambos os formulários, o controle passa da `if` instrução para a próxima instrução no programa, a menos que uma das instruções contenha um `break` , `continue` ou `goto` .

Veja a seguir exemplos da `if` instrução:

```
if ( i > 0 )
    y = x / i;
else
{
    x = i;
    y = f( x );
}
```

Neste exemplo, a instrução `y = x/i;` é executada se `i` for maior que 0. Se `i` for menor ou igual a 0, `i` é atribuído a `x` e `f( x )` é atribuído a `y` . Observe que a instrução que formam a `if` cláusula termina com um ponto e vírgula.

Ao aninhar `if` instruções e `else` cláusulas, use chaves para agrupar as instruções e cláusulas em instruções compostas que esclarecem sua intenção. Se nenhuma chave estiver presente, o compilador resolverá as ambiguidades associando cada `else` uma com o mais próximo `if` que não tem um `else` .

```
if ( i > 0 )          /* Without braces */
    if ( j > i )
        x = j;
    else
        x = i;
```

A `else` cláusula é associada à instrução interna `if` neste exemplo. Se `i` for menor ou igual a 0, nenhum valor será atribuído a `x` .

```
if ( i > 0 )
{                       /* With braces */
    if ( j > i )
        x = j;
}
else
    x = i;
```

As chaves que circundam a `if` instrução interna neste exemplo fazem a `else` parte da cláusula da `if` instrução externa. Se `i` for menor ou igual a 0, `i` é atribuído a `x`.

Veja também

Instrução if-else (C++)

Instrução nula (C)

13/05/2021 • 2 minutes to read

Uma "instrução nula" é uma instrução que contém apenas um ponto e vírgula; ela pode aparecer onde quer que uma instrução seja esperada. Nada acontece quando uma instrução nula é executada. A forma correta de codificar uma instrução nula é:

Sintaxe

```
;
```

Comentários

Instruções como `do` „ `for` `if` e `while` exigem que uma instrução executável apareça como o corpo da instrução. A instrução nula satisfaz o requisito de sintaxe nos casos que não precisam de um corpo de instrução substantivo.

Como acontece com qualquer outra instrução do C, você pode incluir um rótulo antes de uma instrução nula. Para rotular um item que não é uma instrução, como a chave de fechamento de uma instrução composta, você pode rotular uma instrução nula e inseri-la imediatamente antes do item para obter o mesmo efeito.

Este exemplo ilustra a instrução nula:

```
for ( i = 0; i < 10; line[i++] = 0 )
     ;
```

Neste exemplo, a expressão de loop da `for` instrução `line[i++] = 0` Inicializa os primeiros 10 elementos de `line` como 0. O corpo da instrução é uma instrução nula, já que nenhuma instrução adicional é necessária.

Veja também

Instruções

Instrução return (C)

13/05/2021 • 4 minutes to read

Uma `return` instrução encerra a execução de uma função e retorna o controle para a função de chamada. A execução é retomada na função de chamada no ponto imediatamente após a chamada. Uma `return` instrução pode retornar um valor para a função de chamada. Para obter mais informações, consulte tipo de retorno.

Syntax

> *instrução de salto*:
>
> `return` ***expressão*** $_{opt}$ `;`

O valor de *expression*, se existir, será retornado à função de chamada. Se o valor de *expression* for omitido, o valor retornado da função será indefinido. A expressão, se presente, será avaliada e convertida no tipo retornado pela função. Quando uma `return` instrução contém uma expressão em funções que têm um `void` tipo de retorno, o compilador gera um aviso e a expressão não é avaliada.

Se nenhuma `return` instrução aparecer em uma definição de função, o controle retornará automaticamente para a função de chamada depois que a última instrução da função chamada for executada. Nesse caso, o valor de retorno da função chamada será indefinido. Se a função tiver um tipo de retorno diferente de `void` , trata-se de um bug sério e o compilador imprime uma mensagem de diagnóstico de aviso. Se a função tiver um `void` tipo de retorno, esse comportamento será ok, mas poderá ser considerado um estilo ruim. Use uma `return` instrução simples para tornar sua intenção clara.

Como uma boa prática de engenharia, sempre especifique um tipo de retorno para suas funções. Se um valor de retorno não for necessário, declare a função para ter o `void` tipo de retorno. Se um tipo de retorno não for especificado, o compilador C assumirá um tipo de retorno padrão de `int` .

Muitos programadores usam parênteses para colocar o argumento de *expressão* da `return` instrução. No entanto, C não exige os parênteses.

O compilador pode emitir uma mensagem de diagnóstico de aviso sobre código inacessível se encontrar quaisquer instruções colocadas após a `return` instrução.

Em uma `main` função, a `return` instrução e a expressão são opcionais. O que acontece com o valor retornado, se um for especificado, depende da implementação. **Específico da Microsoft**: a implementação do Microsoft C retorna o valor da expressão para o processo que invocou o programa, como `cmd.exe` . Se nenhuma `return` expressão for fornecida, o tempo de execução do Microsoft C retornará um valor que indica êxito (0) ou falha (um valor diferente de zero).

Exemplo

Este exemplo é um programa em várias partes. Ele demonstra a `return` instrução e como ela é usada para encerrar a execução da função e, opcionalmente, retornar um valor.

```
// C_return_statement.c
// Compile using: cl /W4 C_return_statement.c
#include <limits.h>      // for INT_MAX
#include <stdio.h>       // for printf

long long square( int value )
{
    // Cast one operand to long long to force the
    // expression to be evaluated as type long long.
    // Note that parentheses around the return expression
    // are allowed, but not required here.
    return ( value * (long long) value );
}
```

A `square` função retorna o quadrado de seu argumento, em um tipo mais amplo para evitar um erro aritmético. **Específico da Microsoft**: na implementação do Microsoft C, o `long long` tipo é grande o suficiente para conter o produto de dois `int` valores sem estouro.

Os parênteses em volta da `return` expressão no `square` são avaliados como parte da expressão e não são exigidos pela `return` instrução.

```
double ratio( int numerator, int denominator )
{
    // Cast one operand to double to force floating-point
    // division. Otherwise, integer division is used,
    // then the result is converted to the return type.
    return numerator / (double) denominator;
}
```

A `ratio` função retorna a proporção de seus dois `int` argumentos como um valor de ponto flutuante `double` . A `return` expressão é forçada a usar uma operação de ponto flutuante ao converter um dos operandos para `double` . Caso contrário, o operador de divisão de inteiro seria usado e a parte fracionária seria perdida.

```
void report_square( void )
{
    int value = INT_MAX;
    long long squared = 0LL;
    squared = square( value );
    printf( "value = %d, squared = %lld\n", value, squared );
    return; // Use an empty expression to return void.
}
```

A `report_square` função chama `square` com um valor de parâmetro de `INT_MAX` , o maior valor inteiro assinado que se ajusta a um `int` . O `long long` resultado é armazenado em `squared` e impresso. A `report_square` função tem um `void` tipo de retorno, portanto, não tem uma expressão em sua `return` instrução.

```
void report_ratio( int top, int bottom )
{
    double fraction = ratio( top, bottom );
    printf( "%d / %d = %.16f\n", top, bottom, fraction );
    // It's okay to have no return statement for functions
    // that have void return types.
}
```

A `report_ratio` função chama `ratio` com valores de parâmetro de `1` e `INT_MAX` . O `double` resultado é armazenado em `fraction` e impresso. A `report_ratio` função tem um `void` tipo de retorno, portanto, não

precisa retornar um valor explicitamente. A execução de `report_ratio` "sai da parte inferior" e não retorna nenhum valor para o chamador.

```
int main()
{
    int n = 1;
    int x = INT_MAX;

    report_square();
    report_ratio( n, x );

    return 0;
}
```

A `main` função chama duas funções: `report_square` e `report_ratio` . Como `report_square` não usa parâmetros e retorna `void` , não atribuímos seu resultado a uma variável. Da mesma forma, `report_ratio` retorna `void` , portanto, não salvamos seu valor de retorno. Depois de cada uma dessas chamadas de função, a execução continua na próxima instrução. Em seguida, `main` retorna um valor de `0` (normalmente usado para relatar êxito) para encerrar o programa.

Para compilar o exemplo, crie um arquivo de código-fonte chamado *`C_return_statement.c`* . Em seguida, copie todo o código de exemplo, na ordem mostrada. Salve o arquivo e compile-o em uma janela de prompt de comando do desenvolvedor usando o comando:

```
cl /W4 C_return_statement.c
```

Em seguida, para executar o código de exemplo, digite `C_return_statement.exe` no prompt de comando. O resultado do exemplo é semelhante a este:

```
value = 2147483647, squared = 4611686014132420609
1 / 2147483647 = 0.0000000004656613
```

Veja também

Instruções

_Static_assert palavra-chave e static_assert macro (C11)

13/05/2021 • 2 minutes to read

Testa uma asserção no momento da compilação. Se a expressão constante especificada for `false`, o compilador exibirá a mensagem especificada e a compilação falhará com o erro C2338; caso contrário, não haverá efeito. Novo no C11.

`_Static_assert` é uma palavra-chave introduzida no C11. `static_assert` é uma macro, introduzida no C11, que é mapeada para a `_Static_assert` palavra-chave.

Sintaxe

```
_Static_assert(constant-expression, string-literal);
static_assert(constant-expression, string-literal);
```

Parâmetros

expressão de constante
Uma expressão constante integral que pode ser avaliada no momento da compilação. Se a expressão for zero (false), o exibirá o parâmetro *String-literal* e a compilação falhará com um erro. Se a expressão for diferente de zero (true), não haverá nenhum efeito.

literal de cadeia de caracteres
A mensagem exibida se a *expressão constante* for avaliada como zero (false). A mensagem deve ser feita usando o conjunto de caracteres base do compilador. Os caracteres não podem ser caracteres multibyte ou largos.

Comentários

A `_Static_assert` palavra-chave e a `static_assert` macro testam uma asserção de software no momento da compilação. Eles podem ser usados em escopo global ou de função.

Em contraste, 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.

Comportamento específico da Microsoft

Em C, quando você não inclui `<assert.h>`, o compilador do Microsoft Visual C/C++ trata `static_assert` como uma palavra-chave que mapeia para `_Static_assert`. Usar `static_assert` é preferencial, pois o mesmo código funcionará em C e C++.

Exemplo de uma declaração de tempo de compilação

No exemplo a seguir, `static_assert` e `_Static_assert` são usadas para verificar quantos elementos estão em um enum e se os inteiros têm 32 bits de largura.

```
// requires /std:c11 or higher
#include <assert.h>

enum Items
{
    A,
    B,
    C,
    LENGTH
};

int main()
{
    // _Static_assert is a C11 keyword
    _Static_assert(LENGTH == 3, "Expected Items enum to have three elements");

    // Preferred: static_assert maps to _Static_Assert and is compatible with C++
    static_assert(sizeof(int) == 4, "Expecting 32 bit integers");

    return 0;
}
```

Requisitos

MACRO	CABEÇALHO NECESSÁRIO
`static_assert`	<assert.h>

Veja também

_STATIC_ASSERT macro
macro Assert e funções de _assert e _wassert /STD (especifique a versão padrão do idioma)

`switch` Instrução (C)

13/05/2021 • 4 minutes to read

As `switch` `case` instruções e ajudam a controlar operações condicionais e de ramificação complexas. A `switch` instrução transfere o controle para uma instrução dentro de seu corpo.

Sintaxe

```
selection-statement:
    switch ( expression ) statement
```

```
labeled-statement:
    case constant-expression : statement
    default : statement
```

Comentários

Uma `switch` instrução faz com que o controle seja transferido para um *`labeled-statement`* em seu corpo de instrução, dependendo do valor de *`expression`* .

Os valores de *`expression`* e cada *`constant-expression`* devem ter um tipo integral. Um *`constant-expression`* deve ter um valor integral constante não ambíguo no tempo de compilação.

O controle passa para a `case` instrução cujo *`constant-expression`* valor corresponde ao valor de *`expression`* . A `switch` instrução pode incluir qualquer número de `case` instâncias. No entanto, dois *`constant-expression`* valores dentro da mesma `switch` instrução podem ter o mesmo valor. A execução do `switch` corpo da instrução começa na primeira instrução no ou após a correspondência *`labeled-statement`* . A execução continua até o fim do corpo ou até que uma `break` instrução transfira o controle para fora do corpo.

O uso da `switch` instrução geralmente é semelhante a este:

```
switch ( expression )
{
    // declarations
    // . . .
    case constant_expression:
        // statements executed if the expression equals the
        // value of this constant_expression
        break;
    default:
        // statements executed if expression does not equal
        // any case constant_expression
}
```

Você pode usar a `break` instrução para finalizar o processamento de uma instrução rotulada em particular dentro da `switch` instrução. Ele é ramificado para o final da `switch` instrução. Sem `break` , o programa continua para a próxima instrução rotulada, executando as instruções até que um `break` ou o final da instrução seja atingido. Essa continuação pode ser desejável em algumas situações.

A `default` instrução será executada se nenhum `case` *`constant-expression`* valor for igual ao valor de *`expression`* . Se não houver nenhuma `default` instrução e nenhuma `case` correspondência for encontrada,

nenhuma das instruções no `switch` corpo será executada. Pode haver no máximo uma `default` instrução. A `default` instrução não precisa aparecer no final. Ele pode aparecer em qualquer lugar no corpo da `switch` instrução. Um `case` `default` rótulo ou só pode aparecer dentro de uma `switch` instrução.

O tipo de `switch` *`expression`* e `case` *`constant-expression`* deve ser integral. O valor de cada `case` *`constant-expression`* deve ser exclusivo dentro do corpo da instrução.

Os `case` `default` rótulos e do `switch` corpo da instrução são significativos apenas no teste inicial que determina onde a execução começa no corpo da instrução. `switch` as instruções podem ser aninhadas. Todas as variáveis estáticas são inicializadas antes de serem executadas em qualquer `switch` instrução.

> **NOTE**
> As declarações podem aparecer no início da instrução composta que formam o `switch` corpo, mas as inicializações incluídas nas declarações não são executadas. A `switch` instrução transfere o controle diretamente para uma instrução executável dentro do corpo, ignorando as linhas que contêm inicializações.

Os exemplos a seguir ilustram `switch` instruções:

```
switch( c )
{
    case 'A':
        capital_a++;
    case 'a':
        letter_a++;
    default :
        total++;
}
```

Todas as três instruções do `switch` corpo neste exemplo serão executadas se `c` for igual a `'A'` , pois nenhuma `break` instrução aparecerá antes do seguinte `case` . O controle de execução é transferido para a primeira instrução (`capital_a++;`) e continua em ordem pelo restante do corpo. Se `c` é igual a `'a'` , `letter_a` e `total` são incrementados. Só `total` é incrementado quando `c` não é igual a `'A'` ou `'a'` .

```
switch( i )
{
    case -1:
        n++;
        break;
    case 0 :
        z++;
        break;
    case 1 :
        p++;
        break;
}
```

Neste exemplo, uma `break` instrução segue cada instrução do `switch` corpo. A `break` instrução força uma saída do corpo da instrução após a execução de uma instrução. Se `i` for igual a -1, apenas `n` será incrementado. A `break` instrução a seguir `n++;` faz com que o controle de execução passe do corpo da instrução, ignorando as instruções restantes. Da mesma forma, se `i` é igual a 0, somente `z` será incrementado; se `i` é igual a 1, somente `p` será incrementado. A `break` instrução final não é estritamente necessária, já que o controle passa do corpo no final da instrução composta. Ele está incluído para fins de consistência.

Uma única instrução pode transportar vários `case` Rótulos, como mostra o exemplo a seguir:

```
switch( c )
{
    case 'a' :
    case 'b' :
    case 'c' :
    case 'd' :
    case 'e' :
    case 'f' :  convert_hex(c);
}
```

Neste exemplo, se *constant-expression* for igual a qualquer letra entre `'a'` e `'f'`, a função `convert_hex` será chamada.

Específico da Microsoft

O Microsoft C não limita o número de `case` valores em uma `switch` instrução. O número é limitado somente pela memória disponível. ANSI C requer pelo menos 257 `case` Rótulos serem permitidos em uma `switch` instrução.

O default para Microsoft C é que as extensões da Microsoft estão habilitadas. Use a opção de compilador /za para desabilitar essas extensões.

Veja também

switch Instrução (C++)

Instrução try-except (C)

13/05/2021 • 3 minutes to read

Específico da Microsoft

A `try-except` instrução é uma extensão da Microsoft para a linguagem C que permite aos aplicativos obter o controle de um programa quando os eventos que normalmente terminam a execução ocorrem. Esses eventos são denominados exceções, e o mecanismo que lida com exceções é chamado de manipulação de exceção estruturada.

As exceções podem ser baseadas em hardware ou software. Mesmo quando os aplicativos não conseguem se recuperar completamente de exceções de hardware ou software, a manipulação de exceção estruturada torna possível registrar e exibir informações de erro. É útil interceptar o estado interno do aplicativo para ajudar a diagnosticar o problema. Em particular, é útil para problemas intermitentes que não são fáceis de reproduzir.

Sintaxe

```
try-except-statement:
    __try compound-statement __except ( expression ) compound-statement
```

A instrução composta após a `__try` cláusula é a *seção protegida*. A instrução composta após a `__except` cláusula é o *manipulador de exceção*. O manipulador Especifica um conjunto de ações a serem executadas se uma exceção for gerada durante a execução da seção protegida. 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. O valor retornado determina como a exceção é tratada. Há três valores possíveis:
   - `EXCEPTION_CONTINUE_SEARCH`: 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_CONTINUE_EXECUTION`: A exceção é reconhecida, mas ignorada. Continue a execução no ponto onde ocorreu a exceção.
   - `EXCEPTION_EXECUTE_HANDLER` A exceção é reconhecida. Transfira o controle para o manipulador de exceção executando a `__except` instrução composta e continue a execução no ponto em que ocorreu a exceção.

Como a `__except` expressão é avaliada como uma expressão C, ela é limitada 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.

> **NOTE**
> A manipulação de exceção estruturada funciona com arquivos de código-fonte em C e C++. No entanto, ele não foi projetado especificamente para C++. Para programas Portable C++, a manipulação de exceção do C++ deve ser usada em vez de manipulação de exceção estruturada. Além disso, o mecanismo de tratamento de exceções de C++ é muito mais flexível, pois pode tratar exceções de qualquer tipo. Para obter mais informações, consulte tratamento de exceção na referência da *linguagem C++*.

Cada rotina em um aplicativo pode ter seu próprio manipulador de exceção. A `__except` expressão é executada no escopo do `__try` corpo. Ele tem acesso a qualquer variável local declarada lá.

A `__leave` palavra-chave é válida dentro de um `try-except` bloco de instruções. O efeito de `__leave` é saltar para o final do `try-except` bloco. A execução é retomada após o término do manipulador de exceção. Embora uma `goto` instrução possa ser usada para atingir o mesmo resultado, uma `goto` instrução causa o desenrolamento da pilha. A `__leave` instrução é mais eficiente porque não envolve o desenrolamento de pilha.

Sair de uma `try-except` instrução usando a `longjmp` função de tempo de execução é considerado encerramento anormal. Não é legal pular para uma `__try` instrução, mas é legal pular de uma. O manipulador de exceção não será chamado se um processo for eliminado no meio da execução de uma `try-except` instrução.

Exemplo

Aqui está um exemplo de um manipulador de exceção e um manipulador de terminação. Para obter mais informações sobre manipuladores de terminação, consulte `try-finally` Statement (C).

```
.
.
.
puts("hello");
__try {
   puts("in try");
   __try {
      puts("in try");
      RAISE_AN_EXCEPTION();
   } __finally {
      puts("in finally");
   }
} __except( puts("in filter"), EXCEPTION_EXECUTE_HANDLER ) {
   puts("in except");
}
puts("world");
```

Aqui está a saída do exemplo, com comentários adicionados à direita:

```
hello
in try              /* fall into try                        */
in try              /* fall into nested try                 */
in filter           /* execute filter; returns 1 so accept  */
in finally          /* unwind nested finally                */
in except           /* transfer control to selected handler */
world               /* flow out of handler                  */
```

ENCERRAR específico da Microsoft

Confira também

`try-except` instrução (C++)

Instrução try-finally (C)

13/05/2021 • 2 minutes to read

Específico da Microsoft

A instrução `try-finally` é uma extensão da Microsoft para a linguagem C que permite que aplicativos garantam a execução do código de limpeza quando a execução de um bloco de códigos 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.

```
try-finally-statement:
    __try compound-statement __finally compound-statement
```

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. Não importa se a seção protegida é encerrada por uma exceção (encerramento anormal) ou por padrão percorrem (término normal).

O controle alcança uma `__try` instrução por execução sequencial simples (que se enquadra). Quando o controle entra na `__try` instrução, seu manipulador associado se torna ativo. A execução procede da seguinte maneira:

1. A seção protegida é executada.
2. O manipulador de término é invocado.
3. Quando o manipulador de encerramento for concluído, a execução continuará após a `__finally` instrução. Não importa como a seção protegida termina (por exemplo, por meio `goto` de uma instrução fora do corpo protegido ou por meio de uma `return` instrução), o manipulador de terminação é executado antes do fluxo de controle ser movido para fora da seção protegida.

A `__leave` palavra-chave é válida dentro de um `try-finally` bloco de instruções. O efeito de `__leave` é saltar para o final do `try-finally` bloco. O manipulador de término é executado imediatamente. Embora uma `goto` instrução possa ser usada para atingir o mesmo resultado, uma `goto` instrução causa o desenrolamento da pilha. A `__leave` instrução é mais eficiente porque não envolve o desenrolamento de pilha.

Sair de uma `try-finally` instrução usando uma `return` instrução ou a `longjmp` função de tempo de execução é considerado encerramento anormal. Não é legal pular para uma `__try` instrução, mas legal para saltar de uma. Todas as `__finally` instruções que estão ativas entre o ponto de partida e o destino devem ser executadas. Ele é chamado de *desenrolamento local*.

O manipulador de encerramento não será chamado se um processo for encerrado durante a execução de uma `try-finally` instrução.

> **NOTE**
>
> A manipulação de exceção estruturada funciona com arquivos de código-fonte em C e C++. No entanto, ele não foi projetado especificamente para C++. Para programas Portable C++, a manipulação de exceção do C++ deve ser usada em vez de manipulação de exceção estruturada. Além disso, o mecanismo de tratamento de exceções de C++ é muito mais flexível, pois pode tratar exceções de qualquer tipo. Para obter mais informações, consulte tratamento de exceção na referência da *linguagem C++*.

Consulte o exemplo para a `try-except` instrução para ver como a `try-finally` instrução funciona.

ENCERRAR específico da Microsoft

Confira também

`try-finally` instrução (C++)

Instrução while (C)

13/05/2021 • 2 minutes to read

A `while` instrução permite repetir uma instrução até que uma expressão especificada se torne falsa.

Sintaxe

instrução Iteration:
instrução **while (*expressão*)**

A *expressão* deve ter o tipo aritmético ou ponteiro. A execução procede da seguinte maneira:

1. A *expressão* é avaliada.
2. Se a *expressão* for inicialmente falsa, o corpo da `while` instrução nunca será executado e o controle passará da `while` instrução para a próxima instrução no programa.

   Se a *expressão* for verdadeira (diferente de zero), o corpo da instrução será executado e o processo será repetido no início da etapa 1.

A `while` instrução também pode terminar quando um `break` , `goto` ou `return` dentro do corpo da instrução for executado. Use a `continue` instrução para encerrar uma iteração sem sair do `while` loop. A `continue` instrução passa o controle para a próxima iteração da `while` instrução.

Este é um exemplo da `while` instrução:

```
while ( i >= 0 )
{
    string1[i] = string2[i];
    i--;
}
```

Esse exemplo copia caracteres de `string2` para `string1` . Se `i` for maior ou igual a 0, `string2[i]` é atribuído a `string1[i]` e `i` é diminuído. Quando `i` atinge ou cai abaixo de 0, a execução da `while` instrução é encerrada.

Veja também

Instrução while (C++)

Funções (C)

13/05/2021 • 2 minutes to read

A função a unidade modular fundamental em C. Uma função geralmente é criada para executar uma tarefa específica e seu nome geralmente reflete essa tarefa. Uma função contém declarações e instruções. Esta seção descreve como declarar, definir e chamar funções do C. Outros tópicos discutidos são:

- Visão geral de funções
- Atributos de função
- Especificando convenções de chamada
- Funções embutidas
- Funções de exportação e importação de DLL
- Funções naked
- Classe de armazenamento
- Tipo de retorno
- Argumentos
- Parâmetros

Veja também

Referência da linguagem C

Visão geral das funções

13/05/2021 • 2 minutes to read

As funções devem ter uma definição e uma declaração, embora uma definição possa servir como uma declaração se a declaração aparecer antes que a função seja chamada. A definição de função inclui o corpo da função — o código executado quando a função é chamada.

Uma declaração de função estabelece o nome, o tipo de retorno e os atributos de uma função que é definida em outro lugar no programa. Uma declaração de função deve preceder a chamada para a função. É por isso que os arquivos de cabeçalho que contêm as declarações para as funções de tempo de execução são incluídos em seu código antes de uma chamada a uma função de tempo de execução. Se a declaração tiver informações sobre os tipos e o número de parâmetros, a declaração é um protótipo. Consulte Protótipos de função para obter mais informações.

O compilador usa o protótipo para comparar os tipos de argumentos em chamadas subsequentes à função com os parâmetros da função, e para converter os tipos dos argumentos nos tipos dos parâmetros sempre que necessário.

Uma chamada de função passa o controle de execução da função de chamada para a função chamada. Os argumentos, se houver, são passados por valor à função chamada. A execução de uma `return` instrução na função chamada retorna o controle e, possivelmente, um valor para a função de chamada.

Confira também

Funções

Formas obsoletas de declarações e definições da função

13/05/2021 • 2 minutes to read

As declarações e definições de função de estilo antigo usam regras ligeiramente diferentes para declarar parâmetros em relação à sintaxe recomendada pelo padrão ANSI C. Primeiro, as declarações de estilo antigo não têm uma lista de parâmetros. Segundo, na definição de função, os parâmetros são listados, mas seus tipos não são declarados na lista de parâmetros. As declarações de tipo precedem a instrução composta que constitui o corpo da função. A sintaxe de estilo antigo está obsoleta e não deve ser usada em novas instâncias de código. Contudo, o código que usa a sintaxe de estilo antigo ainda tem suporte. Este exemplo ilustra os formatos obsoletos de declarações e definições:

```
double old_style();           /* Obsolete function declaration */

double alt_style( a , real )  /* Obsolete function definition */
    double *real;
    int a;
{
    return ( *real + a ) ;
}
```

As funções que retornam um inteiro ou ponteiro com o mesmo tamanho que um `int` não são necessárias para ter uma declaração, embora a declaração seja recomendada.

Para manter a conformidade com o padrão ANSI C, as declarações de função de estilo antigo que usam reticências agora geram um erro ao compilar com a opção /Za e um aviso de nível 4 ao compilar com /Ze. Por exemplo:

```
void funct1( a, ... )  /* Generates a warning under /Ze or */
int a;                 /* an error when compiling with /Za */
{
}
```

Recomenda-se reescrever essa declaração como um protótipo:

```
void funct1( int a, ... )
{
}
```

As declarações de função de estilo antigo também geram avisos se você subsequentemente declara ou define a mesma função com reticências ou com um parâmetro de um tipo que não é igual ao tipo promovido.

A próxima seção, Definições da função C, mostra a sintaxe para definições da função, inclusive a sintaxe de estilo antigo. O não terminal para a lista de parâmetros na sintaxe de estilo antigo é *identifier-list*.

Veja também

Visão geral das funções

Definições de função C

13/05/2021 • 2 minutes to read

Uma definição de função especifica o nome da função, os tipos e o número de parâmetros que espera receber, além de seu tipo de retorno. Uma definição de função também inclui um corpo de função com as declarações de suas variáveis locais, e as instruções que determinam o que a função faz.

Sintaxe

translation-unit:
    declaração externa
    translation-unit external-declaration

declaração externa:/ * permitida somente em escopo externo (arquivo) */
    função-definição
    mesma

função-definição:
    declaration-specifiers$_{opt}$ *attribute-seq*$_{opt}$ *declarator declaration-list*$_{opt}$ *compound-statement*

/**Attribute-Seq* é específico da Microsoft*/

Os parâmetros de protótipo são:

declaration-specifiers:
    storage-class-specifier declaration-specifiers$_{opt}$
    type-specifier declaration-specifiers$_{opt}$
    type-qualifier declaration-specifiers$_{opt}$

declaration-list:
    mesma
    declaration-list declaration

Declarador:
    pointer$_{opt}$ *direct-declarator*

Declarador direto:/ * um Declarador de função */
    Declarador *direto***(*tipo de parâmetro-lista*)** / * Declarador de novo estilo */
    *Declarador direto***(**opção identificador-lista***)** / * Declarador de estilo obsoleto */

A lista de parâmetros em uma definição usa esta sintaxe:

tipo de parâmetro-lista:/ * a lista de parâmetros */
    lista de parâmetros
    lista de parâmetros **,...**

lista de parâmetros:
    declaração de parâmetro
    parâmetro-List **,** *declaração de parâmetro*

parameter-declaration:
    declaration-specifiers declarator
    declaration-specifiers abstract-declarator$_{opt}$

A lista de parâmetros em uma definição de função antiga usa esta sintaxe:

identificador-lista:/ * usado em definições e declarações de função de estilo obsoleto */
ID
identificador-lista , *identificador*

A sintaxe para o corpo da função é:

instrução composta:
{ *declaração-lista* de instruções $_{opt}$ *-lista*$_{opt}$ **}**

Os únicos especificadores de classe de armazenamento que podem modificar uma declaração de função são `extern` e `static` . O `extern` especificador significa que a função pode ser referenciada de outros arquivos; ou seja, o nome da função é exportado para o vinculador. O `static` especificador significa que a função não pode ser referenciada de outros arquivos; ou seja, o nome não é exportado pelo vinculador. Se nenhuma classe de armazenamento aparecer em uma definição de função, `extern` será assumida. Em qualquer caso, a função sempre é visível do ponto de definição ao final do arquivo.

Juntos, os *declaration-specifiers* opcionais e *declarator* obrigatórios especificam o nome e tipo de retorno da função. O *declarator* é uma combinação de identificador que nomeia a função e os parênteses depois do nome da função. O *attribute-seq* não terminal opcional é um recurso específico da Microsoft definido em Atributos de função.

O *direct-declarator* (na sintaxe *declarator*) especifica o nome da função sendo definida e os identificadores dos seus parâmetros. Se o *direct-declarator* incluir uma *parameter-type-list*, a lista especifica os tipos de todos os parâmetros. Esse declarador também serve como um protótipo da função para chamadas posteriores para a função.

Uma *declaração* na *lista de declarações* nas definições de função não pode conter um *especificador de classe de armazenamento* diferente de `register` . O *especificador de tipo* na sintaxe de *especificadores de declaração* pode ser omitido somente se a `register` classe de armazenamento for especificada para um valor do `int` tipo.

A *compound-statement* é o corpo da função que contém declarações de variável local, referências a itens declarados externamente e instruções.

As seções Atributos de função, Classe de armazenamento, Tipo de retorno, Parâmetros e Corpo da função descrevem os componentes da definição de função detalhadamente.

Confira também

Funções

Atributos de função

13/05/2021 • 2 minutes to read

Específico da Microsoft

O não terminal *attribute-seq* opcional permite que você selecione uma convenção de chamada em uma base por função. Você também pode especificar funções como `__fastcall` ou `__inline` .

FINAL específico da Microsoft

Veja também

Definições de função C

Especificando convenções de chamada

13/05/2021 • 2 minutes to read

Específico da Microsoft

Para obter informações sobre as convenções de chamada, consulte Tópicos de Convenções de Chamada.

FINAL específico da Microsoft

Veja também

Atributos de função

Funções embutidas

13/05/2021 • 2 minutes to read

Específico da Microsoft

A `__inline` palavra-chave informa ao compilador para substituir o código na definição de função para cada instância de uma chamada de função. No entanto, a substituição ocorre apenas ao critério do compilador. Por exemplo, o compilador não uma embute uma função se seu endereço já estiver em uso ou se for muito grande para embutir.

Para que uma função seja considerada candidata para embutir, ela deve usar a definição de função de novo estilo.

Use esse formato para especificar uma função embutida:

```
__inline função opcional de tipo-definição
```

O uso de funções embutidas gera código mais rápido e às vezes pode gerar código menor do que a chamada de função equivalente gera pelos seguintes motivos:

- Ela poupa o tempo necessário para executar chamadas de função.
- As funções embutidas pequenas, talvez três linhas ou menos, criam menos código do que a chamada de função equivalente porque o compilador não gera código para tratar de argumentos e um valor de retorno.
- As funções geradas embutidas estão sujeitas às otimizações de código não disponíveis para funções normais porque o compilador não executa otimizações entre procedimentos.

As funções que usam `__inline` não devem ser confundidas com o código do assembler embutido. Consulte Assembler embutido para obter mais informações.

FINAL específico da Microsoft

Veja também

inline, __inline, _ _forceinline

Assembler embutido (C)

13/05/2021 • 2 minutes to read

Específico da Microsoft

O assembler integrado permite inserir instruções da linguagem de assembly diretamente aos seus programas de código-fonte C sem etapas adicionais de assembly e vinculação. O assembler integrado é incorporado ao compilador e, portanto, não é necessário um assembler separado, como o MASM (Microsoft Macro Assembler).

Como o assembler integrado não requer etapas de link e do assembly separado, é mais conveniente que um assembler separado. O código do assembly integrado pode usar qualquer nome de variável ou de função C que esteja no escopo. Portanto, ele é de fácil integração com código C do programa. E como o código do assembly pode ser combinado com as instruções de C, ele poderá realizar as tarefas que são incômodas ou impossíveis apenas em C.

A `__asm` palavra-chave invoca o Assembler embutido e pode aparecer sempre que uma instrução C for legal. Ela não pode aparecer sozinha. Ela deve ser seguida por uma instrução de assembly, um grupo de instruções entre chaves ou, pelo menos, um par vazio de chaves. O termo " `__asm` Bloquear" refere-se a qualquer instrução ou grupo de instruções, seja ou não entre chaves.

O código a seguir é um `__asm` bloco simples entre chaves. (O código é uma sequência personalizada de prólogos da função.)

```
__asm
{
   push ebp
   mov  ebp, esp
   sub  esp, __LOCAL_SIZE
}
```

Como alternativa, você pode colocar `__asm` na frente de cada instrução de assembly:

```
__asm push ebp
__asm mov  ebp, esp
__asm sub  esp, __LOCAL_SIZE
```

Como a `__asm` palavra-chave é um separador de instrução, você também pode colocar instruções de assembly na mesma linha:

```
__asm push ebp   __asm mov  ebp, esp   __asm sub  esp, __LOCAL_SIZE
```

FINAL específico da Microsoft

Veja também

Atributos de função

`_Noreturn` palavra-chave e `noreturn` macro (C11)

13/05/2021 • 2 minutes to read

A `_Noreturn` palavra-chave foi introduzida em C11. Ele informa ao compilador que a função a que ela é aplicada não retorna ao chamador. O compilador sabe que o código após uma chamada para uma `_Noreturn` função está inacessível. Um exemplo de uma função que não retorna é abortar. Se houver uma possibilidade para o fluxo de controle retornar ao chamador, a função não deverá ter o `_Noreturn` atributo.

A palavra-chave normalmente é usada por meio da macro de conveniência, `noreturn` fornecida em <stdnoreturn. h>, que é mapeada para a `_Noreturn` palavra-chave.

Os principais benefícios de usar `_Noreturn` (ou o equivalente `noreturn`) estão tornando a intenção da função clara no código para futuros leitores e detectando código inacessível involuntariamente.

Uma função marcada `noreturn` não deve incluir um tipo de retorno porque não retorna um valor para o chamador. Ela deverá ser `void` .

Exemplo usando `noreturn` macro e `_Noreturn` palavra-chave

O exemplo a seguir demonstra a `_Noreturn` palavra-chave e a `noreturn` macro equivalente.

O IntelliSense pode gerar um erro falso, `E0065` se você usar a macro `noreturn` que você pode ignorar. Ele não impede que você execute o exemplo.

```
// Compile with Warning Level4 (/W4) and /std:c11
#include <stdio.h>
#include <stdlib.h>
#include <stdnoreturn.h>

noreturn void fatal_error(void)
{
    exit(3);
}

_Noreturn void not_coming_back(void)
{
    puts("There's no coming back");
    fatal_error();
    return; // warning C4645 - function declared with noreturn has a return statement
}

void done(void)
{
    puts("We'll never get here");
}

int main(void)
{
    not_coming_back();
    done(); // warning c4702 - unreachable code

    return 0;
}
```

Requisitos

MACRO	CABEÇALHO NECESSÁRIO
`noreturn`	<stdnoreturn.h>

Confira também

/STD (especifique a versão padrão do idioma)
/W4 (especificar nível de aviso)
Aviso de C4702
__declspec (noreturn)

Funções de importação e exportação de DLL

13/05/2021 • 2 minutes to read

Específico da Microsoft

As informações mais completas e atualizadas sobre este tópico podem ser encontradas em dllexport, dllimport.

Os `dllimport` `dllexport` modificadores de classe de armazenamento e são extensões específicas da Microsoft para a linguagem C. Esses modificadores definem explicitamente a interface da DLL para o cliente (o arquivo executável ou outra DLL). Declarar funções como `dllexport` elimina a necessidade de um arquivo de definição de módulo (.DEF). Você também pode usar os `dllimport` `dllexport` modificadores e com dados e objetos.

Os `dllimport` `dllexport` modificadores de classe de armazenamento e devem ser usados com a palavra-chave de sintaxe de atributo estendido, `__declspec` , conforme mostrado neste exemplo:

```
#define DllImport   __declspec( dllimport )
#define DllExport   __declspec( dllexport )

DllExport void func();
DllExport int i = 10;
DllExport int j;
DllExport int n;
```

Para obter informações específicas sobre a sintaxe de modificadores de classe de armazenamento estendidos, consulte Atributos de classe de armazenamento estendidos.

FINAL específico da Microsoft

Veja também

Definições de função C

Definições e declarações (C)

13/05/2021 • 2 minutes to read

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 `dllimport` ou `dllexport` . Todas as declarações incluídas na interface DLL devem especificar o `dllimport` atributo ou `dllexport` . No entanto, a definição pode especificar apenas o atributo `dllexport` . Por exemplo, a definição de função a seguir gera um erro de compilador:

```
#define DllImport   __declspec( dllimport )
#define DllExport   __declspec( dllexport )

DllImport int func()    /* Error; dllimport prohibited in */
                        /* definition. */
{
   return 1;
}
```

Este código também gera um erro:

```
#define DllImport   __declspec( dllimport )
#define DllExport   __declspec( dllexport )

DllImport int i = 10;      /* Error; this is a definition. */
```

No entanto, esta é uma sintaxe correta:

```
#define DllImport   __declspec( dllimport )
#define DllExport   __declspec( dllexport )

DllExport int i = 10;      /* Okay: this is an export definition. */
```

O uso de `dllexport` implica uma definição, enquanto `dllimport` implica uma declaração. Você deve usar a `extern` palavra-chave WITH `dllexport` para forçar uma declaração; caso contrário, uma definição é implícita.

```
#define DllImport   __declspec( dllimport )
#define DllExport   __declspec( dllexport )

extern DllImport int k;   /* These are correct and imply */
Dllimport int j;          /* a declaration. */
```

FINAL específico da Microsoft

Veja também

Funções de importação e exportação de DLL

Definindo funções C embutidas com dllexport e dllimport

13/05/2021 • 2 minutes to read

Específico da Microsoft

Você pode definir como uma função embutida com o atributo `dllexport`. 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 à especificação da opção do compilador /Ob (embutido)), 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.

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

Veja também

Funções de importação e exportação de DLL

Regras e limitações para dllimport/dllexport

13/05/2021 • 2 minutes to read

Específico da Microsoft

- Se você declarar uma função sem o `dllimport` `dllexport` atributo ou, a função não será considerada parte da interface DLL. Consequentemente, a definição da função deverá estar presente nesse módulo ou em outro módulo do mesmo programa. Para tornar a função parte da interface da DLL, você deve declarar a definição da função no outro módulo como `dllexport`. Caso contrário, um erro de vinculador é gerado quando o cliente é compilado.

- Se um único módulo no seu programa contiver `dllimport` `dllexport` declarações e para a mesma função, o `dllexport` atributo terá precedência sobre o `dllimport` atributo. No entanto, um aviso do compilador será gerado. Por exemplo:

```
#define DllImport   __declspec( dllimport )
#define DllExport   __declspec( dllexport )

   DllImport void func1( void );
   DllExport void func1( void );   /* Warning; dllexport */
                                   /* takes precedence. */
```

- Não é possível inicializar um ponteiro de dados estáticos com o endereço de um objeto de dados declarado com o `dllimport` atributo. Por exemplo, o código a seguir gera erros:

```
#define DllImport   __declspec( dllimport )
#define DllExport   __declspec( dllexport )

   DllImport int i;
   .
   .
   .
   int *pi = &i;                           /* Error */

   void func2()
   {
      static int *pi = &i;                   /* Error */
   }
```

- Inicializar um ponteiro de função estática com o endereço de uma função declarado com `dllimport` define o ponteiro para o endereço da conversão de importação de dll (um stub de código que transfere o controle para a função) em vez do endereço da função. Essa atribuição não gera uma mensagem de erro:

```
#define DllImport   __declspec( dllimport )
#define DllExport   __declspec( dllexport )

   DllImport void func1( void
   .
   .
   .
   static void ( *pf )( void ) = &func1;   /* No Error */

   void func2()
   {
      static void ( *pf )( void ) = &func1;  /* No Error */
   }
```

- Como um programa que inclui o atributo `dllexport` na declaração de um objeto deve fornecer a definição desse objeto, você pode inicializar um ponteiro de função estático local ou global com o endereço de uma função `dllexport`. Da mesma forma, você pode inicializar um ponteiro de dados estático global ou local com o endereço de um objeto de dados `dllexport`. Por exemplo:

```
#define DllImport   __declspec( dllimport )
#define DllExport   __declspec( dllexport )

   DllImport void func1( void );
   DllImport int i;

   DllExport void func1( void );
   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 */
   }
```

FINAL específico da Microsoft

Veja também

Funções de importação e exportação de DLL

Funções Naked

13/05/2021 • 2 minutes to read

Específico da Microsoft

O atributo de classe de armazenamento `naked` é uma extensão específica da Microsoft na linguagem C. Para funções declaradas com o atributo de classe de armazenamento `naked`, o compilador gera um código sem código de prólogo e de 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.

Como o atributo `naked` é relevante apenas para a definição de uma função e não é um modificador de tipo, as funções naked devem usar a sintaxe de atributo estendido, descrita em Atributos de classe de armazenamento estendidos.

O seguinte exemplo define uma função com o atributo `naked`:

```
__declspec( naked ) int func( formal_parameters )
{
   /* Function body */
}
```

Ou, alternativamente:

```
#define Naked   __declspec( naked )

Naked int func( formal_parameters )
{
   /* Function body */
}
```

O atributo `naked` afeta somente a natureza de 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. Assim, o atributo `naked` não é considerado parte do tipo de função, e os ponteiros da função não podem ter o atributo `naked`. Além disso, o atributo `naked` não pode ser aplicado a uma definição de dados. Por exemplo, o código a seguir gera erros:

```
__declspec( naked ) int i;  /* Error--naked attribute not */
                            /* permitted on data declarations. */
```

O atributo `naked` é relevante apenas à definição da função e não pode ser especificado no protótipo da função. Esta declaração gera um erro de compilador:

```
__declspec( naked ) int func();   /* Error--naked attribute not */
                    /* permitted on function declarations.    */   \
```

FINAL específico da Microsoft

Veja também

Regras e limitações para usar funções naked

13/05/2021 • 2 minutes to read

Para obter informações sobre regras e restrições para o uso de funções naked, consulte o tópico correspondente na referência da linguagem C++: Regras e restrições para funções naked.

Veja também

Funções Naked

Considerações quando escrever código de prólogo/epílogo

13/05/2021 • 2 minutes to read

Específico da Microsoft

Antes de escrever suas próprias sequências de código prólogo e epílogo, é importante entender como o quadro de pilhas é disposto. Também é útil saber como usar o **__LOCAL_SIZE** constante predefinida.

Layout de quadro CStack

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                         ; Return from function
```

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 enviados por push de `ebp` . A área de variáveis locais começa em `ebp-2` . Para acessar variáveis locais, calcule um deslocamento de `ebp` subtraindo o valor apropriado de `ebp` .

A constante __LOCAL_SIZE

O compilador fornece uma constante, **__LOCAL_SIZE**, para o uso no bloco embutido do assembler do código do prólogo da função. Essa constante é usada para alocar espaço para as variáveis locais no quadro da pilha no código personalizado de prólogo.

O compilador determina o valor de **__LOCAL_SIZE**. O 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 usada apenas como um operando imediato; não pode ser usada em uma expressão. Você não deve alterar ou redefinir o valor dessa constante. Por exemplo:

```
mov     eax, __LOCAL_SIZE           ;Immediate operand--Okay
mov     eax, [ebp - __LOCAL_SIZE]   ;Error
```

O exemplo a seguir mostra uma função naked que contém sequências de prólogo e epílogo personalizadas e usa **__LOCAL_SIZE** na sequência de prólogo:

```
__declspec ( naked ) func()
{
   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
      }
}
```

FINAL específico da Microsoft

Veja também

Funções Naked

Classe de armazenamento

13/05/2021 • 2 minutes to read

O especificador de classe de armazenamento em uma definição de função fornece a função `extern` ou a `static` classe de armazenamento.

Sintaxe

função-definição:
declaration-specifiers$_{opt}$ *attribute-seq*$_{opt}$ *declarator declaration-list*$_{opt}$ *compound-statement*

/* *Attribute-Seq* é específico da Microsoft*/

declaration-specifiers:
storage-class-specifier declaration-specifiers$_{opt}$
type-specifier declaration-specifiers$_{opt}$
type-qualifier declaration-specifiers$_{opt}$

armazenamento-classe-especificador:/ * para definições de função */
`extern`
`static`

Se uma definição de função não incluir um *especificador de classe de armazenamento*, a classe de armazenamento será padronizada como `extern` . Você pode declarar explicitamente uma função como `extern` , mas ela não é necessária.

Se a declaração de uma função contiver o *especificador de classe de armazenamento* `extern` , o identificador terá o mesmo vínculo que qualquer declaração visível do identificador com escopo de arquivo. Se não houver declaração visível com escopo de arquivo, o identificador terá vinculação externa. Se um identificador tiver escopo de arquivo e nenhum *storage-class-specifier*, o identificador terá vinculação externa. Vinculação externa significa que cada instância do identificador denota o mesmo objeto ou função. Consulte Tempo de vida, escopo, visibilidade e vinculação para obter informações sobre vinculação e escopo de arquivo.

Declarações de função de escopo de bloco com um especificador de classe de armazenamento diferente de `extern` gerar erros.

Uma função com `static` classe de armazenamento é visível somente no arquivo de origem no qual ela está definida. Todas as outras funções, se recebem `extern` uma classe de armazenamento explícita ou implicitamente, são visíveis em todos os arquivos de origem do programa. Se a `static` classe de armazenamento for desejada, ela deverá ser declarada na primeira ocorrência de uma declaração (se houver) da função e na definição da função.

Específico da Microsoft

Quando as extensões da Microsoft são habilitadas, uma função originalmente declarada sem uma classe de armazenamento (ou com `extern` classe de armazenamento) recebe `static` a classe de armazenamento se a definição da função estiver no mesmo arquivo de origem e se a definição especificar explicitamente a `static` classe de armazenamento.

Ao compilar com a opção de compilador/ze, as funções declaradas em um bloco usando a `extern` palavra-chave têm visibilidade global. Isso não ocorre na compilação com /Za. Esse recurso não deve ser usado se a portabilidade do código-fonte estiver em consideração.

Veja também

Definições de função C

Tipo de retorno

13/05/2021 • 2 minutes to read

O tipo de retorno de uma função estabelece o tamanho e o tipo do valor retornado pela função e corresponde ao elemento type-specifier na sintaxe abaixo:

Sintaxe

função-definição:
declaration-specifiers$_{opt}$ *attribute-seq*$_{opt}$ *declarator declaration-list*$_{opt}$ *compound-statement*

/* *Attribute-Seq* é específico da Microsoft*/

declaration-specifiers:
storage-class-specifier declaration-specifiers$_{opt}$
type-specifier declaration-specifiers$_{opt}$
type-qualifier declaration-specifiers$_{opt}$

type-specifier:
`void`
`char`
`short`
`int`
`__int8` /* Específico da Microsoft */
`__int16` /* Específico da Microsoft */
`__int32` /* Específico da Microsoft */
`__int64` /* Específico da Microsoft */
`long`
`float`
`double`
`signed`
`unsigned`
especificador struct-ou-Union-
especificador de enumeração
typedef-Name

type-specifier pode especificar qualquer tipo de fundamental, de estrutura ou união. Se você não incluir o *especificador de tipo*, o tipo de retorno `int` será assumido.

O tipo de retorno especificado na definição de função deve corresponder ao tipo de retorno em declarações da função em outros lugares no programa. Uma função retorna um valor quando uma `return` instrução que contém uma expressão é executada. A expressão é avaliada, convertida no tipo de valor de retorno, se necessário, e retornada ao ponto em que a função foi chamada. Se uma função for declarada com o tipo de retorno `void` , uma instrução de retorno contendo uma expressão gerará um aviso e a expressão não será avaliada.

Os exemplos a seguir ilustram valores de retorno da função.

```
typedef struct
{
    char name[20];
    int id;
    long class;
} STUDENT;

/* Return type is STUDENT: */

STUDENT sortstu( STUDENT a, STUDENT b )
{
    return ( (a.id < b.id) ? a : b );
}
```

Este exemplo define o `STUDENT` tipo com uma `typedef` declaração e define a função `sortstu` para ter o `STUDENT` tipo de retorno. A função seleciona e retorna um de seus dois argumentos de estrutura. Em chamadas subsequentes à função, o compilador verifica se os tipos de argumento são `STUDENT`.

> **NOTE**
> A eficiência seria aprimorada passando ponteiros para a estrutura, em vez da estrutura inteira.

```
char *smallstr( char s1[], char s2[] )
{
    int i;

    i = 0;
    while ( s1[i] != '\0' && s2[i] != '\0' )
        i++;
    if ( s1[i] == '\0' )
        return ( s1 );
    else
        return ( s2 );
}
```

O exemplo define uma função que retorna um ponteiro para uma matriz de caracteres. A função usa duas matrizes de caracteres (cadeias de caracteres) como argumentos e retorna um ponteiro para a mais curta das duas cadeias de caracteres. Um ponteiro para uma matriz aponta para o primeiro dos elementos da matriz e tem seu tipo; assim, o tipo de retorno da função é um ponteiro a ser digitado `char` .

Você não precisa declarar funções com `int` tipo de retorno antes de chamá-las, embora os protótipos sejam recomendados para que a verificação de tipo correta para argumentos e valores de retorno esteja habilitada.

Veja também

Definições de função C

Parâmetros

13/05/2021 • 2 minutes to read

Os argumentos são nomes de valores transmitidos a uma função por uma chamada de função. Os parâmetros são os valores que a função espera receber. Em um protótipo de função, os parênteses posteriores ao nome da função contêm uma lista completa dos parâmetros da função e seus tipos. As declarações de parâmetro especificam os tipos, os tamanhos e os identificadores dos valores armazenados nos parâmetros.

Sintaxe

`function-definition` :
`declaration-specifiers` aceitar `attribute-seq` aceitar `declarator` `declaration-list` aceitar `compound-statement`

/* `attribute-seq` é específico da Microsoft*/

`declarator` :
`pointer` aceitar `direct-declarator`

`direct-declarator` :/ * Um Declarador de função */
`direct-declarator` `(` `parameter-type-list` `)` /* Declarador de novo estilo */
`direct-declarator` `(` `identifier-list` `)` / aceitar * Declarador de estilo obsoleto*/

`parameter-type-list` :/ * A lista de parâmetros */
`parameter-list`
`parameter-list` `, ...`

`parameter-list` :
`parameter-declaration`
`parameter-list` `,` `parameter-declaration`

`parameter-declaration` :
`declaration-specifiers` `declarator`
`declaration-specifiers` ** `abstract-declarator` aceitar

O `parameter-type-list` é uma sequência de declarações de parâmetro separadas por vírgulas. O formulário de cada parâmetro em uma lista de parâmetros tem esta aparência:

```
register aceitar type-specifier declarator aceitar
```

Parâmetros de função declarados com o `auto` atributo geram erros. Os identificadores dos parâmetros são usados no corpo da função para fazer referência aos valores transmitidos à função. Você pode nomear os parâmetros em um protótipo, mas os nomes saem do escopo no final da declaração. Isso significa que os nomes de parâmetros podem ser atribuídos da mesma maneira ou de forma diferente na definição da função. Esses identificadores não podem ser redefinidos no bloco mais externo do corpo da função, mas podem ser redefinidos em blocos internos e aninhados como se a lista de parâmetros fosse um bloco delimitador.

Cada identificador em `parameter-type-list` deve ser precedido por seu especificador de tipo apropriado, conforme mostrado neste exemplo:

```
void new( double x, double y, double z )
{
    /* Function body here */
}
```

Se pelo menos um parâmetro ocorrer na lista de parâmetros, a lista poderá terminar com uma vírgula seguida de três pontos (`, ...`). Essa construção, chamada "notação de reticências", indica um número variável de argumentos para a função. (Para obter mais informações, consulte chamadas com um número variável de argumentos.) No entanto, uma chamada para a função deve ter pelo menos tantos argumentos quantos houver parâmetros antes da última vírgula.

Se nenhum argumento for passado para a função, a lista de parâmetros será substituída pela palavra-chave `void` . Esse uso de `void` é diferente de seu uso como um especificador de tipo.

A ordem e o tipo de parâmetros, inclusive qualquer uso da notação de reticências, devem ser iguais em todas as declarações de função (se houver) e na definição da função. Os tipos de argumentos depois das conversões aritméticas comuns devem ter atribuições compatíveis com os tipos dos parâmetros correspondentes. (Consulte conversões aritméticas usuais para obter informações sobre conversões aritméticas.) Os argumentos após as reticências não são verificados. Um parâmetro pode ter qualquer tipo fundamental, de estrutura, união, ponteiro ou matriz.

O compilador executa as conversões aritméticas comuns independentemente em cada parâmetro e em cada argumento, se necessário. Após a conversão, nenhum parâmetro é menor que um `int` , e nenhum parâmetro tem `float` o tipo, a menos que o tipo de parâmetro seja explicitamente especificado como `float` no protótipo. Isso significa, por exemplo, que declarar um parâmetro como um `char` tem o mesmo efeito que declará-lo como um `int` .

Veja também

Definições de função C

Corpo da função

13/05/2021 • 2 minutes to read

Um *corpo de função* é uma instrução composta que contém as instruções que especificam o que a função faz.

Sintaxe

função-definição:
  declaration-specifiers$_{opt}$ *attribute-seq*$_{opt}$ *declarator declaration-list*$_{opt}$ *compound-statement*

/* *Attribute-Seq* é específico da Microsoft*/

instrução composta:/ * o corpo da função */
  { *declaração-lista* de instruções $_{opt}$ *-lista*$_{opt}$ **}**

Variáveis declaradas em um corpo de função, conhecidas como *variáveis locais*, têm `auto` a classe de armazenamento, a menos que especificado de outra forma. Quando a função for chamada, o armazenamento é criado para as variáveis locais e as inicializações de local são executadas. O controle de execução passa para a primeira instrução na *instrução composta* e continua até que uma `return` instrução seja executada ou o final do corpo da função seja encontrado. O controle então retorna para o ponto no qual a função foi chamada.

Uma `return` instrução que contém uma expressão deve ser executada se a função for retornar um valor. O valor de retorno de uma função será indefinido se nenhuma `return` instrução for executada ou se a `return` instrução não incluir uma expressão.

Veja também

Definições de função C

Protótipos de função

13/05/2021 • 3 minutes to read

Uma declaração de função precede a definição de função e especifica o nome, o tipo de retorno, a classe de armazenamento e os outros atributos de uma função. Para ser um protótipo, a declaração de função também deve estabelecer os tipos e identificadores dos argumentos da função.

Sintaxe

declaração:
  declaração-especificadores atributo-Seq$_{opt}$ *init-Declaration-List*$_{opt}$ **;**

/* *atributo-Seq*$_{opt}$ é específico da Microsoft*/

declaration-specifiers:
  storage-class-specifier declaration-specifiers$_{opt}$
  type-specifier declaration-specifiers$_{opt}$
  type-qualifier declaration-specifiers$_{opt}$

init-declarator-list:
  init-Declarador
  init-declarator-List **,** *init-declarator*

init-Declarador:
  Declarador
  Declarador **=** *inicializador*

Declarador:
  pointer$_{opt}$ *direct-declarator*

Declarador direto:/ * um Declarador de função */
  Declarador *direto***(*tipo de parâmetro-lista*)** / * Declarador de novo estilo */
  *Declarador direto***(**opção identificador-lista***)** / * Declarador de estilo obsoleto */

O protótipo tem o mesmo formato que a definição de função, exceto que é ele é encerrado por um ponto-e-vírgula imediatamente depois do parêntese de fechamento e, portanto, não tem corpo. Em ambos os casos, o tipo de retorno deve concordar com o tipo de retorno especificado na definição de função.

Os protótipos de função têm os seguintes usos importantes:

- Eles estabelecem o tipo de retorno para funções que retornam tipos diferentes de `int` . Embora as funções que retornam `int` valores não exijam protótipos, os protótipos são recomendados.
- Sem protótipos completos, as conversões padrão são feitas, mas nenhuma tentativa será feita para verificar o tipo ou o número de argumentos com o número de parâmetros.
- Os protótipos são usados para inicializar ponteiros para as funções antes que essas funções sejam definidas.
- A lista de parâmetros é usada para verificar a correspondência de argumentos na chamada de função com os parâmetros na definição de função.

O tipo convertido de cada parâmetro determina a interpretação dos argumentos que a chamada de função coloca na pilha. Uma incompatibilidade entre um argumento e um parâmetro pode fazer com que os

argumentos na pilha sejam interpretados incorretamente. Por exemplo, em um computador de 16 bits, se um ponteiro de 16 bits for passado como um argumento e, em seguida, declarado como um `long` parâmetro, os primeiros 32 bits na pilha serão interpretados como um `long` parâmetro. Esse erro cria problemas não apenas com o `long` parâmetro, mas com quaisquer parâmetros que o seguem. Você pode detectar erros desse tipo declarando protótipos de função completos para todas as funções.

Um protótipo estabelece os atributos de uma função de forma que as chamadas para a função que precedem sua definição (ou ocorrem em outros arquivos de origem) possam ser verificadas quanto a incompatibilidades de tipo de argumento e de retorno. Por exemplo, se você especificar o `static` especificador de classe de armazenamento em um protótipo, também deverá especificar a `static` classe de armazenamento na definição da função.

As declarações de parâmetro completas (`int a`) podem ser misturadas com declaradores abstratos (`int`) na mesma declaração. Por exemplo, a declaração a seguir é aceitável:

```
int add( int a, int );
```

O protótipo pode incluir o tipo e um identificador para cada expressão que é passada como um argumento. Porém, tais identificadores têm escopo apenas até o final da declaração. O protótipo também pode refletir o fato de que o número de argumentos é variável ou que nenhum argumento é passado. Sem essa lista, as incompatibilidades podem não ser reveladas, para que o compilador não gere mensagens de diagnóstico relacionadas a elas. Consulte Argumentos para obter mais informações sobre verificação de tipo.

O escopo de protótipo no compilador do Microsoft C agora está em conformidade com ANSI ao compilar com a opção do compilador **/Za**. Isso significa que, se você declarar `struct` uma `union` marca ou dentro de um protótipo, a marca será inserida nesse escopo, em vez de no escopo global. Por exemplo, ao compilar com **/Za** para estar em conformidade com ANSI, você nunca pode chamar essa função sem receber um erro de incompatibilidade:

```
void func1( struct S * );
```

Para corrigir seu código, defina ou declare o `struct` ou `union` no escopo global antes do protótipo de função:

```
struct S;
void func1( struct S * );
```

Em **/Ze**, a marca ainda é inserida no escopo global.

Confira também

Funções

Chamadas de função

13/05/2021 • 2 minutes to read

Uma *chamada de função* é uma expressão que passa o controle e os argumentos (se houver) para uma função e tem o formato:

expressão ($_{opt}$*de lista de expressões*)

em que *expression* é um nome de função ou é avaliado em um endereço de função e *expression-list* é uma lista de expressões (separadas por vírgulas). Os valores dessas últimas expressões são os argumentos passados para a função. Se a função não retornar um valor, você o declarará como uma função que retorna `void` .

Se uma declaração existe antes da chamada de função, mas nenhuma informação é fornecida quanto aos parâmetros, os argumentos não declarados simplesmente passam pelas conversões aritméticas comuns.

> **NOTE**
> As expressões na lista de argumentos da função podem ser avaliadas em qualquer ordem, portanto, os argumentos cujos valores podem ser alterados por efeitos colaterais de outro argumento têm valores indefinidos. O ponto de sequência definido pelo operador de chamada de função apenas garante que todos os efeitos colaterais na lista de argumentos sejam avaliados antes que o controle passe para a função chamada. (Observe que a ordem na qual os argumentos são enviados por push na pilha é uma questão separada.) Consulte pontos de sequência para obter mais informações.

O único requisito em qualquer chamada de função é que a expressão antes dos parênteses devem ser avaliadas como um endereço de função. Isso significa que uma função pode ser chamada por qualquer expressão de ponteiro de função.

Exemplo

Este exemplo ilustra chamadas de função chamadas de uma `switch` instrução:

```
int main()
{
    /* Function prototypes */

    long lift( int ), step( int ), drop( int );
    void work( int number, long (*function)(int i) );

    int select, count;
    .
    .
    .
    select = 1;
    switch( select )
    {
        case 1: work( count, lift );
                break;

        case 2: work( count, step );
                break;

        case 3: work( count, drop );
                /* Fall through to next case */
        default:
                break;
    }
}

/* Function definition */

void work( int number, long (*function)(int i) )
{
    int i;
    long j;

    for ( i = j = 0; i < number; i++ )
            j += ( *function )( i );
}
```

Nesse exemplo, a chamada de função em `main`,

```
work( count, lift );
```

passa uma variável de inteiro, `count`, e o endereço da função `lift` para a função `work`. Observe que o endereço da função é passado simplesmente fornecendo o identificador de função, pois um identificador de função é avaliado em uma expressão de ponteiro. Para usar um identificador de função dessa forma, a função deve ser declarada ou definida antes que o identificador seja usado; caso contrário, o identificador não é reconhecido. Nesse caso, um protótipo para `work` é fornecido no início da função `main`.

O parâmetro `function` em `work` é declarado como um ponteiro para uma função que assume um `int` argumento e retorna um `long` valor. Os parênteses em volta do nome do parâmetro são obrigatórios; sem eles, a declaração especificaria uma função que retornasse um ponteiro para um `long` valor.

A função `work` chama a função selecionada de dentro do `for` loop usando a seguinte chamada de função:

```
( *function )( i );
```

Um argumento, `i`, é passado para a função chamada.

Confira também

Argumentos

13/05/2021 • 3 minutes to read

Os argumentos em uma chamada de função têm este formato:

```
expressão ( opcional de expressão-lista de expressões ) /* chamada de função */
```

Em uma chamada de função, *expression-list* é uma lista de expressões (separadas por vírgula). Os valores dessas últimas expressões são os argumentos passados para a função. Se a função não usar argumentos, a *lista de expressões* deverá conter a palavra-chave `void` .

Um argumento pode ser qualquer valor com fundamental, estrutura, união ou tipo de ponteiro. Todos os argumentos são passados por valor. Isso significa que uma cópia do argumento será atribuída ao parâmetro correspondente. A função não sabe o local real da memória do argumento passado. A função usará essa cópia sem afetar a variável da qual foi derivada originalmente.

Embora você não possa passar matrizes ou funções como argumentos, é possível passar ponteiros para esses itens. Os ponteiros permitem que uma função acesse um valor por referência. Como um ponteiro para uma variável contém o endereço da variável, a função pode usar esse endereço para acessar o valor da variável. Os argumentos de ponteiro permitem que uma função acesse matrizes e funções, mesmo que as matrizes e funções não possam ser passadas como argumentos.

A ordem em que os argumentos são avaliados pode variar nos compiladores diferentes e níveis diferentes de otimização. No entanto, os argumentos e os efeitos colaterais são avaliados completamente antes de a função ser inserida. Consulte Efeitos colaterais para obter informações sobre os efeitos colaterais.

A *expression-list* em uma chamada de função é avaliada e as conversões aritméticas comuns são executadas em cada argumento da chamada de função. Se um protótipo estiver disponível, o tipo de argumento resultante será comparado ao parâmetro correspondente do protótipo. Caso isso não aconteça, uma conversão será executada ou uma mensagem de diagnóstico será emitida. Os parâmetros também passam por conversões aritméticas comuns.

O número de expressões em *expression-list* deve corresponder ao número de parâmetros, a menos que o protótipo ou a definição da função especifique explicitamente um número variável de argumentos. Nesse caso, o compilador verifica quantos argumentos houver nos nomes de tipo na lista de parâmetros e os converte, se necessário, conforme descrito acima. Consulte chamadas com um número variável de argumentos para obter mais informações.

Se a lista de parâmetros do protótipo contiver apenas a palavra-chave `void` , o compilador esperará zero argumentos na chamada de função e nenhum parâmetro na definição. Uma mensagem de diagnóstico será emitida se encontrar argumentos.

Exemplo

Este exemplo usa ponteiros como argumentos:

```
int main()
{
    /* Function prototype */

    void swap( int *num1, int *num2 );
    int x, y;
    .
    .
    .
    swap( &x, &y );  /* Function call */
}

/* Function definition */

void swap( int *num1, int *num2 )
{
    int t;

    t = *num1;
    *num1 = *num2;
    *num2 = t;
}
```

Neste exemplo, a `swap` função é declarada em `main` para ter dois argumentos, representados respectivamente por identificadores `num1` e `num2` , ambos, são ponteiros para `int` valores. Os parâmetros `num1` e `num2` na definição de estilo de protótipo também são declarados como ponteiros para `int` valores de tipo.

Na chamada de função

```
swap( &x, &y )
```

o endereço de `x` é armazenado em `num1` , e o endereço de `y` é armazenado em `num2` . Agora há dois nomes ou "aliases" para o mesmo local. As referências a `*num1` e a `*num2` em `swap` são efetivamente referências a `x` e a `y` em `main` . As atribuições de `swap` trocam realmente o conteúdo de `x` e `y` . Portanto, nenhuma `return` instrução é necessária.

O compilador executa a verificação de tipo nos argumentos para `swap` porque o protótipo de `swap` inclui tipos de argumento para cada parâmetro. Os identificadores entre parênteses do protótipo e a definição podem ser iguais ou diferentes. O importante aqui é que os tipos dos argumentos correspondam aos das listas de parâmetros no protótipo e na definição.

Veja também

Chamadas de função

Chamadas com um número variável de argumentos

13/05/2021 • 2 minutes to read

Uma lista de parâmetros parcial pode ser finalizada pela notação de reticências, uma vírgula seguida de três pontos (, ...), para indicar que pode haver mais argumentos transmitidos à função, mas não são fornecidas mais informações sobre eles. A verificação de tipo não é executada nesses argumentos. Pelo menos um parâmetro deve preceder a notação de reticências, e a notação de reticências deve ser o token mais recente na lista de parâmetros. Sem a notação de reticências, o comportamento de uma função é indefinido se receber parâmetros além daqueles declarados na lista de parâmetros.

Para chamar uma função com um número variável de argumentos, basta especificar qualquer número de argumentos na chamada de função. Um exemplo é a função `printf` da biblioteca em tempo de execução C. A chamada de função deve incluir um argumento para cada nome do tipo declarado na lista de parâmetros ou na lista de tipos de argumento.

Todos os argumentos especificados na chamada de função são colocados na pilha, a menos que a `__fastcall` Convenção de chamada seja especificada. O número de parâmetros declarados para a função determina quantos dos argumentos são obtidos da pilha e atribuídos aos parâmetros. Você é responsável por recuperar todos os argumentos adicionais da pilha e por determinar quantos argumentos estão presentes. O arquivo STDARG.H contém macros estilo ANSI para acessar argumentos das funções que possuem um número variável de argumentos. Além disso, macros de estilo XENIX em VARARGS.H têm suporte.

Esse exemplo de declaração é para uma função que chama um número variável de argumentos:

```
int average( int first, ...);
```

Veja também

Chamadas de função

Funções Recursivas

13/05/2021 • 2 minutes to read

Qualquer função em um programa C pode ser chamada recursivamente, ou seja, pode chamar a si mesma. O número de chamadas recursivas está limitado ao tamanho da pilha. Consulte a opção de vinculador `/STACK` (alocações de pilha) para obter informações sobre opções de vinculador que definem o tamanho da pilha. Cada vez que a função é chamada, o novo armazenamento é alocado para os parâmetros e para as `auto` variáveis e para `register` que seus valores nas chamadas anteriores, não concluídas, não sejam substituídas. Os parâmetros só são diretamente acessíveis para a instância da função na qual são criados. Os parâmetros anteriores não são diretamente acessíveis para instâncias resultantes da função.

Observe que as variáveis declaradas com `static` armazenamento não exigem um novo armazenamento com cada chamada recursiva. O armazenamento existe para todo o tempo de vida do programa. Cada referência a essa variável acessa a mesma área de armazenamento.

Exemplo

Este exemplo ilustra chamadas recursivas:

```
int factorial( int num );      /* Function prototype */

int main()
{
    int result, number;
    .
    .
    .
    result = factorial( number );
}

int factorial( int num )      /* Function definition */
{
    .
    .
    .
    if ( ( num > 0 ) || ( num <= 10 ) )
        return( num * factorial( num - 1 ) );
}
```

Veja também

Chamadas de função

Resumo da sintaxe da linguagem C

13/05/2021 • 2 minutes to read

Esta seção fornece a descrição completa dos recursos da linguagem C e da linguagem C específica da Microsoft. Você pode usar a notação de sintaxe nesta seção para determinar a sintaxe exata de qualquer componente da linguagem. A explicação da sintaxe aparece na seção deste manual em que um tópico é discutido.

> **NOTE**
> Este resumo da sintaxe não faz parte do padrão ANSI C, mas é incluído apenas para fins informativos. A sintaxe específica da Microsoft é indicada em comentários depois da sintaxe.

Veja também

Referência da linguagem C

Definições e convenções

13/05/2021 • 2 minutes to read

Os terminais são pontos de extremidade em uma definição de sintaxe. Nenhuma outra resolução é possível. Os terminais incluem o conjunto de palavras reservadas e identificadores definidos pelo usuário.

Os não terminais são espaços reservados na sintaxe e são definidos em outra parte deste resumo da sintaxe. As definições podem ser recursivas.

Um componente opcional é indicado pela $_{\text{recusa}}$de subscrito. Por exemplo:

```
{ expressão_opt }
```

indica uma expressão opcional entre chaves.

As convenções de sintaxe usam atributos diferentes de fonte para componentes diferentes de sintaxe. Os símbolos e as fontes são os seguintes:

ATRIBUTO	DESCRIÇÃO
Não terminal	O tipo em itálico indica não terminais.
`const`	Os terminais de tipo em negrito são palavras reservadas a literais e símbolos que devem ser inseridos como mostrado. Os caracteres nesse contexto sempre diferenciam maiúsculas de minúsculas.
`opt`	Os não-terminais seguidos por $_{\text{opt}}$ são sempre opcionais.
default typeface	Os caracteres no conjunto listados ou descritos nesta face de tipos podem ser usados como terminais em instruções de C.

Um sinal de dois pontos (:) depois de um não terminal introduz sua definição. Definições alternativas estão listadas em linhas separadas, exceto quando prefaciadas com as palavras "um de".

Veja também

Resumo da sintaxe da linguagem C

Gramática Lexical C

13/05/2021 • 3 minutes to read

Tokens

token :

- *keyword*
- *identifier*
- *constant*
- *string-literal*
- *punctuator*

preprocessing-token :

- *header-name*
- *identifier*
- *pp-number*
- *character-constant*
- *string-literal*
- *punctuator*

cada caractere diferente de espaço em branco que não pode ser um dos acima

Palavras-chave

keyword : um de

`auto` `break` `case` `char` `const` `continue`
`default` `do` `double` `else` `enum` `extern`
`float` `for` `goto` `if` `inline` `int` `long`
`register` `restrict` `return` `short` `signed`
`sizeof` `static` `struct` `switch` `typedef` `union`
`unsigned` `void` `volatile` `while` `_Alignas`
`_Alignof` `_Atomic` `_Bool` `_Complex` `_Generic`
`_Imaginary` `_Noreturn` `_Static_assert`
`_Thread_local`

Para obter uma lista de palavras-chave adicionais específicas da Microsoft, consulte as palavras-chave C.

Identificadores

identifier :

- *identifier-nondigit*
- *identifier* *identifier-nondigit*
- *identifier* *digit*

identifier-nondigit :

- *nondigit*
- *universal-character-name*

outros caracteres definidos pela implementação

nondigit : um de

`_` `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`

digit : um de
`0` `1` `2` `3` `4` `5` `6` `7` `8` `9`

universal-character-name :
`\u` *hex-quad*
`\U` *hex-quad* *hex-quad*

hex-quad :
hexadecimal-digit *hexadecimal-digit* *hexadecimal-digit* *hexadecimal-digit*

:
:

constant :
integer-constant
floating-constant
enumeration-constant
character-constant

integer-constant :
decimal-constant ** *integer-suffix* aceitar
binary-constant [1] *integer-suffix* aceitar
octal-constant ** *integer-suffix* aceitar
hexadecimal-constant ** *integer-suffix* aceitar

decimal-constant :
nonzero-digit
decimal-constant *digit*

binary-constant : [1]
binary-prefix *binary-digit*
binary-constant *binary-digit*

binary-prefix [1]: um de
`0b` `0B`

binary-digit [1]: um de
`0` `1`

octal-constant :
`0`
octal-constant *octal-digit*

hexadecimal-constant :
hexadecimal-prefix *hexadecimal-digit*
hexadecimal-constant *hexadecimal-digit*

hexadecimal-prefix : um de
`0x` `0X`

nonzero-digit : um de
`1` `2` `3` `4` `5` `6` `7` `8` `9`

octal-digit : um de

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

hexadecimal-digit : um de

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

`a` `b` `c` `d` `e` `f`

`A` `B` `C` `D` `E` `F`

integer-suffix :

unsigned-suffix ** *long-suffix* aceitar

unsigned-suffix ** *long-long-suffix* aceitar

long-suffix ** *unsigned-suffix* aceitar

long-long-suffix ** *unsigned-suffix* aceitar

unsigned-suffix : um de

`u` `U`

long-suffix : um de

`l` `L`

long-long-suffix : um de

`ll` `LL`

floating-constant :

decimal-floating-constant

hexadecimal-floating-constant

decimal-floating-constant :

fractional-constant optar por *exponent-part* aceitar *floating-suffix* opt

digit-sequence ** *exponent-part* *floating-suffix* aceitar

hexadecimal-floating-constant :

hexadecimal-prefix optar por *hexadecimal-fractional-constant* *binary-exponent-part* aceitar *floating-suffix* opt

hexadecimal-prefix ** *hexadecimal-digit-sequence* *binary-exponent-part* *floating-suffix* aceitar

fractional-constant :

digit-sequence aceitar `.` *digit-sequence*

digit-sequence `.`

exponent-part :

`e` *sign* aceitar *digit-sequence*

`E` *sign* aceitar *digit-sequence*

sign : um de

`+` `-`

digit-sequence :

digit

digit-sequence *digit*

hexadecimal-fractional-constant :

hexadecimal-digit-sequence aceitar `.` *hexadecimal-digit-sequence*

hexadecimal-digit-sequence `.`

binary-exponent-part :

`p` *sign* aceitar *digit-sequence*
`P` *sign* aceitar *digit-sequence*

hexadecimal-digit-sequence :
hexadecimal-digit
hexadecimal-digit-sequence *hexadecimal-digit*

floating-suffix : um de
`f` `l` `F` `L`

enumeration-constant :
identifier

character-constant :
`'` *c-char-sequence* `'`
`L'` *c-char-sequence* `'`

c-char-sequence :
c-char
c-char-sequence *c-char*

c-char :
Qualquer membro do conjunto de caracteres de origem, exceto o sinal de aspas simples (`'`), barra invertida (`\`) ou caractere de nova linha
escape-sequence

escape-sequence :
simple-escape-sequence
octal-escape-sequence
hexadecimal-escape-sequence
universal-character-name

simple-escape-sequence : um de
`\a` `\b` `\f` `\n` `\r` `\t` `\v`
`\'` `\"` `\\` `\?`

octal-escape-sequence :
`\` *octal-digit*
`\` *octal-digit* *octal-digit*
`\` *octal-digit* *octal-digit* *octal-digit*

hexadecimal-escape-sequence :
`\x` *hexadecimal-digit*
hexadecimal-escape-sequence *hexadecimal-digit*

Literais de cadeia de caracteres

string-literal :
encoding-prefix ** *s-char-sequence* aceitar `"`

encoding-prefix :
`u8`
`u`
`U`
`L`

`s-char-sequence` :

`s-char`

`s-char-sequence` `s-char`

`s-char` :

qualquer membro do conjunto de caracteres de origem, exceto aspas duplas (`"`), barra invertida (`\`) ou caractere de nova linha

`escape-sequence`

Pontuadores

`punctuator` : um de

`[` `]` `(` `)` `{` `}` `.` `->`

`++` `--` `&` `*` `+` `-` `~` `!`

`/` `%` `<<` `>>` `<` `>` `<=` `>=` `==`

`!=` `^` `|` `&&` `||` `?` `:` `;` `...`

`=` `*=` `/=` `%=` `+=` `-=` `<<=` `>>=`

`&=` `^=` `|=` `,` `#` `##`

`<:` `:>` `<%` `%>` `%:` `%:%:`

Nomes do Cabeçalho

`header-name` :

`<` `h-char-sequence` `>`

`"` `q-char-sequence` `"`

`h-char-sequence` :

`h-char`

`h-char-sequence` `h-char`

`h-char` :

qualquer membro do conjunto de caracteres de origem, exceto o caractere de nova linha e `>`

`q-char-sequence` :

`q-char`

`q-char-sequence` `q-char`

`q-char` :

qualquer membro do conjunto de caracteres de origem, exceto o caractere de nova linha e `"`

Pré-processando números

`pp-number` :

`digit`

`.` `digit`

`pp-number` `digit`

`pp-number` `identifier-nondigit`

`pp-number` `e` `sign`

`pp-number` `E` `sign`

`pp-number` `p` `sign`

`pp-number` `P` `sign`

`pp-number` `.`

[1] `binary-constant` , `binary-prefix` e `binary-digit` são extensões específicas da Microsoft.

Confira também

Resumo da sintaxe da linguagem C

Gramática de estrutura da frase

13/05/2021 • 2 minutes to read

- Expressões
- Declarações
- Instruções
- Definições externas

Veja também

Resumo da sintaxe da linguagem C

Resumo de expressões

13/05/2021 • 2 minutes to read

primary-expression:
- *identifier*
- *constant*
- *string-literal*
- (*expression*)
- *generic-selection*

generic-selection:
- _Generic (*assignment-expression* , *generic-assoc-list*)

generic-assoc-list:
- *generic-association*
- *generic-assoc-list* , *generic-association*

generic-association:
- *type-name* : *assignment-expression*
- default : *assignment-expression*

postfix-expression:
- *primary-expression*
- *postfix-expression* [*expression*]
- *postfix-expression* (*argument-expression-list* aceitar)
- *postfix-expression* . *identifier*
- *postfix-expression* -> *identifier*
- *postfix-expression* ++
- *postfix-expression* --
- (*type-name*) { *initializer-list* }
- (*type-name*) { *initializer-list* , }

argument-expression-list:
- *assignment-expression*
- *argument-expression-list* , *assignment-expression*

unary-expression:
- *postfix-expression*
- ++ *unary-expression*
- -- *unary-expression*
- *unary-operator* *cast-expression*
- sizeof *unary-expression*
- sizeof (*type-name*) _Alignof (*type-name*)

unary-operator: um de
- & * + - ~ !

cast-expression:
- *unary-expression*
- (*type-name*) *cast-expression*

`multiplicative-expression`:

- `cast-expression`
- `multiplicative-expression` `*` `cast-expression`
- `multiplicative-expression` `/` `cast-expression`
- `multiplicative-expression` `%` `cast-expression`

`additive-expression`:

- `multiplicative-expression`
- `additive-expression` `+` `multiplicative-expression`
- `additive-expression` `-` `multiplicative-expression`

`shift-expression`:

- `additive-expression`
- `shift-expression` `<<` `additive-expression`
- `shift-expression` `>>` `additive-expression`

`relational-expression`:

- `shift-expression`
- `relational-expression` `<` `shift-expression`
- `relational-expression` `>` `shift-expression`
- `relational-expression` `<=` `shift-expression`
- `relational-expression` `>=` `shift-expression`

`equality-expression`:

- `relational-expression`
- `equality-expression` `==` `relational-expression`
- `equality-expression` `!=` `relational-expression`

`AND-expression`:

- `equality-expression`
- `AND-expression` `&` `equality-expression`

`exclusive-OR-expression`:

- `AND-expression`
- `exclusive-OR-expression` `^` `AND-expression`

`inclusive-OR-expression`:

- `exclusive-OR-expression`
- `inclusive-OR-expression` `|` `exclusive-OR-expression`

`logical-AND-expression`:

- `inclusive-OR-expression`
- `logical-AND-expression` `&&` `inclusive-OR-expression`

`logical-OR-expression`:

- `logical-AND-expression`
- `logical-OR-expression` `||` `logical-AND-expression`

`conditional-expression`:

- `logical-OR-expression`
- `logical-OR-expression` `?` `expression` `:` `conditional-expression`

`assignment-expression`:

- `conditional-expression`
- `unary-expression` `assignment-operator` `assignment-expression`

assignment-operator: um de

`=` `*=` `/=` `%=` `+=` `-=` `<<=` `>>=` `&=` `^=` `|=`

expression:

assignment-expression

expression `,` *assignment-expression*

constant-expression:

conditional-expression

Confira também

- Gramática da estrutura da frase

Resumo de declarações

13/05/2021 • 2 minutes to read

declaration :
- *declaration-specifiers* ** *attribute-seq* aceitar[1] *init-declarator-list* opt `;`
- *static_assert-declaration*

declaration-specifiers :
- *storage-class-specifier* ** *declaration-specifiers* aceitar
- *type-specifier* ** *declaration-specifiers* aceitar
- *type-qualifier* ** *declaration-specifiers* aceitar
- *function-specifier* ** *declaration-specifiers* aceitar
- *alignment-specifier* ** *declaration-specifiers* aceitar

attribute-seq[1]:
- *attribute*[1] *attribute-seq* aceitar[1]

attribute[1, 2]: uma das
- `__asm` `__based` `__cdecl` `__clrcall` `__fastcall` `__inline` `__stdcall` `__thiscall` `__vectorcall`

init-declarator-list :
- *init-declarator*
- *init-declarator-list* `,` *init-declarator*

init-declarator :
- *declarator*
- *declarator* `=` *initializer*

storage-class-specifier :
- `auto`
- `extern`
- `register`
- `static`
- `_Thread_local`
- `typedef`
- `__declspec` **** `(` *extended-decl-modifier-seq* `)` [1]

extended-decl-modifier-seq[1]:
- *extended-decl-modifier* opt
- *extended-decl-modifier-seq* *extended-decl-modifier*

extended-decl-modifier[1]:
- `thread`
- `naked`
- `dllimport`
- `dllexport`

type-specifier :
- `void`
- `char`
- `short`

- `int`
- `__int8` uma
- `__int16` uma
- `__int32` uma
- `__int64` uma
- `long`
- `float`
- `double`
- `signed`
- `unsigned`
- `_Bool`
- `_Complex`
- *`atomic-type-specifier`*
- *`struct-or-union-specifier`*
- *`enum-specifier`*
- *`typedef-name`*

`struct-or-union-specifier` :
- *`struct-or-union`* ** *`identifier`* opt `{` aceitar *`struct-declaration-list`* `}`
- *`struct-or-union`* *`identifier`*

`struct-or-union` :
- `struct`
- `union`

`struct-declaration-list` :
- *`struct-declaration`*
- *`struct-declaration-list`* *`struct-declaration`*

`struct-declaration` :
- *`specifier-qualifier-list`* ** *`struct-declarator-list`* aceitar `;`
- *`static_assert-declaration`*

`specifier-qualifier-list` :
- *`type-specifier`* ** *`specifier-qualifier-list`* aceitar
- *`type-qualifier`* ** *`specifier-qualifier-list`* aceitar
- *`alignment-specifier`* ** *`specifier-qualifier-list`* aceitar

`struct-declarator-list` :
- *`struct-declarator`*
- *`struct-declarator-list`* `,` *`struct-declarator`*

`struct-declarator` :
- *`declarator`*
- *`declarator`* aceitar `:` *`constant-expression`*

`enum-specifier` :
- `enum` *`identifier`* opt `{` aceitar *`enumerator-list`* `}`
- `enum` *`identifier`* opt `{` aceitar *`enumerator-list`* `,` **** `}`
- `enum` *`identifier`*

`enumerator-list` :
- *`enumerator`*
- *`enumerator-list`* `,` *`enumerator`*

`enumerator` :

`enumeration-constant`

`enumeration-constant` `=` `constant-expression`

`atomic-type-specifier` :

`_Atomic` `(` `type-name` `)`

`type-qualifier` :

`const`

`restrict`

`volatile`

`_Atomic`

`function-specifier` :

`inline`

`_Noreturn`

`alignment-specifier` :

`_Alignas` `(` `type-name` `)`

`_Alignas` `(` `constant-expression` `)`

`declarator` :

`pointer` aceitar `direct-declarator`

`direct-declarator` :

`identifier`

`(` `declarator` `)`

`direct-declarator` optar por `[` `type-qualifier-list` aceitar `assignment-expression` opt `]`

`direct-declarator` `[` `static` `type-qualifier-list` aceitar `assignment-expression` `]`

`direct-declarator` `[` `type-qualifier-list` `static` `assignment-expression` `]`

`direct-declarator` `[` `type-qualifier-list` opt aceitar `*` **** `]`

`direct-declarator` `(` `parameter-type-list` `)`

`direct-declarator` `(` `identifier-list` aceitar `)` [3]

`pointer` :

`*` `type-qualifier-list` aceitar

`*` `type-qualifier-list` aceitar `pointer`

`type-qualifier-list` :

`type-qualifier`

`type-qualifier-list` `type-qualifier`

`parameter-type-list` :

`parameter-list`

`parameter-list` `,` `...`

`parameter-list` :

`parameter-declaration`

`parameter-list` `,` `parameter-declaration`

`parameter-declaration` :

`declaration-specifiers` `declarator`

`declaration-specifiers` ** `abstract-declarator` aceitar

`identifier-list` :/ * Para Declarador de estilo antigo */

`identifier`
`identifier-list` `,` `identifier`

`type-name` :
`specifier-qualifier-list` ** `abstract-declarator` aceitar

`abstract-declarator` :
`pointer`
`pointer` aceitar `direct-abstract-declarator`

`direct-abstract-declarator` :
`(` `abstract-declarator` `)`
`direct-abstract-declarator` optar por `[` `type-qualifier-list` aceitar `assignment-expression` opt `]`
`direct-abstract-declarator` `[` `static` `type-qualifier-list` aceitar `assignment-expression` `]`
`direct-abstract-declarator` `[` `type-qualifier-list` `static` `assignment-expression` `]`
`direct-abstract-declarator` `[` `type-qualifier-list` opt aceitar `*` **** `]`
`direct-abstract-declarator` aceitar `(` `parameter-type-list` aceitar `)`

`typedef-name` :
`identifier`

`initializer` :
`assignment-expression`
`{` `initializer-list` `}`
`{` `initializer-list` `,` `}`

`initializer-list` :
`designation` aceitar `initializer`
`initializer-list` `,` `designation` aceitar `initializer`

`designation` :
`designator-list` `=`

`designator-list` :
`designator`
`designator-list` `designator`

`designator` :
`[` `constant-expression` `]`
`.` `identifier`

`static_assert-declaration` :
`_Static_assert` `(` `constant-expression` `,` `string-literal` `)` `;`

[1] este elemento de gramática é específico da Microsoft.
[2] para obter mais informações sobre esses elementos, consulte,,,,,, `__asm` `__clrcall` `__stdcall` `__based` `__fastcall` `__thiscall` `__cdecl` , `__inline` e `__vectorcall` . [3] este estilo é obsoleto.

Confira também

Convenções de chamada
Gramática da estrutura da frase
Convenções de chamada obsoletas

Resumo de instruções C

13/05/2021 • 2 minutes to read

statement :
- *labeled-statement*
- *compound-statement*
- *expression-statement*
- *selection-statement*
- *iteration-statement*
- *jump-statement*
- *try-except-statement* /* Específico da Microsoft */
- *try-finally-statement* /* Específico da Microsoft */

jump-statement :
- goto *identifier* ;
- continue ;
- break ;
- return *expression* aceitar ;
- __leave ; /*Específico da Microsoft[1]*/

compound-statement :
- { optar por *declaration-list* aceitar *statement-list* opt }

declaration-list :
- *declaration*
- *declaration-list* *declaration*

statement-list :
- *statement*
- *statement-list* *statement*

expression-statement :
- *expression* aceitar ;

iteration-statement :
- while (*expression*) *statement*
- do *statement* while (*expression*) ;
- for (*expression* opt ; aceitar *expression* aceitar ; *expression* opt aceitar) *statement*

selection-statement :
- if (*expression*) *statement*
- if (*expression*) *statement* else *statement*
- switch (*expression*) *statement*

labeled-statement :
- *identifier* : *statement*
- case *constant-expression* : *statement*
- default : *statement*

try-except-statement :/ * Específico da Microsoft */
- __try *compound-statement* __except (*expression*) *compound-statement*

try-finally-statement :/ * Específico da Microsoft */
    __try *compound-statement* __finally *compound-statement*

1 a __leave palavra-chave só é válida dentro do __try bloco de um *try-except-statement* ou de um *try-finally-statement* .

Confira também

Gramática de estrutura de frase

Definições externas

13/05/2021 • 2 minutes to read

translation-unit:
  declaração externa
  translation-unit external-declaration

declaração externa:/ * permitida somente em escopo externo (arquivo) */
  função-definição
  mesma

Function-Definition:/ * declarator aqui é o Declarador de função */
  declaration-specifiers$_{opt}$ *declarator declaration-list*$_{opt}$ *compound-statement*

Veja também

Gramática da estrutura da frase

Comportamento definido pela implementação

13/05/2021 • 2 minutes to read

ANSI x 3.159-1989, *American National Standard para linguagem de programação de sistemas de informações -- C*, contém uma seção chamada "problemas de portabilidade". A seção da ANSI lista as áreas da linguagem C que a ANSI deixa em aberto para cada implementação específica. Esta seção descreve como o Microsoft C trata essas áreas definidas pela implementação da linguagem C.

Esta seção segue a mesma ordem que a seção de ANSI. Cada item coberto inclui referências à ANSI que explica o comportamento definido pela implementação.

> **NOTE**
> Esta seção descreve apenas a versão em inglês dos EUA do compilador do C. As implementações do Microsoft C para outros idiomas podem ser ligeiramente diferente.

Veja também

Referência da linguagem C

Translação: Diagnóstico

13/05/2021 • 2 minutes to read

ANSI 2.1.1.3 Como um diagnóstico é identificado

O Microsoft C gera mensagens de erro no formato:

> *filename* **(** *número de linha* **):** *mensagem* de *número* de *diagnóstico* **C**

em que *filename* é o nome do arquivo de origem no qual o erro foi encontrado; *line-number* é o número da linha na qual o compilador detectou o erro; *diagnostic* é "erro" ou "aviso"; *number* é um número único de quatro dígitos (precedido por um **C**, conforme observado na sintaxe) que identifica o erro ou o aviso; *message* é uma mensagem explicativa.

Veja também

Comportamento definido pela implementação

Ambiente

13/05/2021 • 2 minutes to read

- Argumentos para main
- Dispositivos interativos

Veja também

Comportamento definido pela implementação

Argumentos para main

13/05/2021 • 2 minutes to read

ANSI 2.1.2.2.1 A semântica dos argumentos para main

No Microsoft C, a função chamada na inicialização do programa é chamada **main**. Não há nenhum protótipo declarado para **main** e ele pode ser definido com zero, dois ou três parâmetros:

```
int main( void )
int main( int argc, char *argv[] )
int main( int argc, char *argv[], char *envp[] )
```

A terceira linha acima, em que **main** aceita três parâmetros, é uma extensão da Microsoft para o padrão ANSI C. O terceiro parâmetro, **envp**, é uma matriz dos ponteiros para variáveis de ambiente. A matriz **envp** é terminada por um ponteiro nulo. Consulte A função main e a execução do programa para obter mais informações sobre **main** e **envp**.

A variável **argc** nunca contém um valor negativo.

A matriz de cadeias de caracteres termina com **argv[argc]**, que contém um ponteiro nulo.

Todos os elementos da matriz **argv** são ponteiros para cadeias de caracteres.

Um programa invocado sem argumentos de linha de comando receberá um valor de um para **argc**, já que o nome do arquivo executável é colocado em **argv[0]**. (Em versões do MS-DOS anteriores a 3.0, o nome do arquivo executável não está disponível. A letra "C" é colocada em **argv [0]**.) As cadeias de caracteres apontadas por **argv [1]** a **argv [argc-1]** representam os parâmetros do programa.

Os parâmetros **argc** e **argv** são modificáveis e retêm seus últimos valores armazenados entre a inicialização do programa e o término do programa.

Veja também

Ambiente

Dispositivos interativos

13/05/2021 • 2 minutes to read

ANSI 2.1.2.3 O que constitui um dispositivo interativo

O Microsoft C define o teclado e o monitor como dispositivos interativos.

Veja também

Ambiente

Comportamento de identificadores

13/05/2021 • 2 minutes to read

- Caracteres significativos sem vínculo externo
- Caracteres significativos com vínculo externo
- Letras maiúsculas e minúsculas

Veja também

Usando extern para especificar a ligação

Caracteres significativos sem vinculação externa

13/05/2021 • 2 minutes to read

ANSI 3.1.2 O número de caracteres significativos sem vinculação externa

Os identificadores são significativos até 247 caracteres. O compilador não restringe o número de caracteres que você pode usar em um identificador; somente ignora os caracteres além do limite.

Veja também

Usando extern para especificar a ligação

Caracteres significativos com vinculação externa

13/05/2021 • 2 minutes to read

ANSI 3.1.2 O número de caracteres significativos com vinculação externa

Identificadores declarados `extern` em programas compilados com o Microsoft C são significativos a 247 caracteres. Você pode modificar esse padrão para um número menor usando a opção /H (que restringe o comprimento dos nomes externos).

Veja também

Usando extern para especificar a ligação

Maiúsculas e minúsculas

13/05/2021 • 2 minutes to read

ANSI 3.1.2 Se as distinções de maiúscula e minúscula são significativas

O Microsoft C trata os identificadores em uma unidade de compilação com diferenciação de maiúsculas e minúsculas.

O vinculador da Microsoft faz diferenciação de maiúsculas e minúsculas. Você deve especificar todos os identificadores consistentemente de acordo com maiúsculas e minúsculas.

Veja também

Comportamento de identificadores

Caracteres

13/05/2021 • 2 minutes to read

- O conjunto de caracteres ASCII
- Caracteres multibyte
- Bits por caractere
- Conjuntos de caracteres
- Constantes de caractere não representadas
- Caracteres largos
- Convertendo caracteres multibyte
- Intervalo de valores de caracteres

Veja também

Comportamento definido pela implementação

Conjunto de caracteres ASCII

13/05/2021 • 2 minutes to read

ANSI 2.2.1 Membros de conjuntos de caracteres de origem e de execução

O conjunto de caracteres de origem é o conjunto de caracteres válidos que podem aparecer nos arquivos de origem. Para o Microsoft C, o conjunto de caracteres de origem é o conjunto de caracteres ASCII padrão.

> **NOTE**
>
> **Aviso** Como os drivers de teclado e de console podem remapear o conjunto de caracteres, os programas destinados à distribuição internacional devem verificar o código do país/região.

Veja também

Caracteres

Caracteres multibyte

13/05/2021 • 2 minutes to read

ANSI 2.2.1.2 Estados de deslocamento para caracteres multibyte

Os caracteres multibyte são usados por algumas implementações, inclusive do Microsoft C, para representar caracteres de língua estrangeira não representados no conjunto de caracteres de base. No entanto, o Microsoft C não oferece suporte a codificações dependentes de estado. Portanto, não há estados de deslocamento. Consulte Caracteres multibyte e largos para obter mais informações.

Veja também

Caracteres

Bits por caractere

13/05/2021 • 2 minutes to read

ANSI 2.2.4.2.1 Número de bits em um caractere

O número de bits em um caractere é representado pela constante de manifesto **CHAR_BIT**. O arquivo LIMITS.H define **CHAR_BIT** como 8.

Veja também

Caracteres

Conjuntos de caracteres

13/05/2021 • 2 minutes to read

ANSI 3.1.3.4 O mapeamento dos membros do conjunto de caracteres de origem

O conjunto de caracteres de origem e o conjunto de caracteres de execução incluem os caracteres ASCII listados na tabela a seguir. As sequências de escape também são mostradas na tabela.

Sequências de escape

SEQUÊNCIA DE ESCAPE	CARACTERE	VALOR DE ASCII
\a	Alerta/sino	7
\b	Backspace	8
\f	Avanço de formulário	12
\n	Nova linha	10
\r	Retorno de carro	13
\t	Guia horizontal	9
\v	Guia vertical	11
\"	Aspas duplas	34
\'	Aspas simples	39
\\	Barra invertida	92

Veja também

Caracteres

Constantes de caracteres sem representação

13/05/2021 • 2 minutes to read

ANSI 3.1.3.4 O valor de uma constante de caractere inteiro que contém um caractere ou uma sequência de escape não representada no conjunto de caracteres de execução básico ou no conjunto de caracteres estendido para uma constante de caractere largo

Todas as constantes de caractere ou sequências de escape podem ser representadas no conjunto de caracteres estendido.

Veja também

Caracteres

Caracteres largos

13/05/2021 • 2 minutes to read

ANSI 3.1.3.4 O valor de uma constante de caractere de inteiro que contém mais de um caractere ou uma constante de caractere largo que contém mais de um caractere multibyte

A constante de caractere comum, 'ab' tem o valor inteiro (int)0x6162. Quando houver mais de um byte, os bytes lidos anteriormente são deslocados para a esquerda pelo valor de **CHAR_BIT** e o próximo byte é comparado usando o operador bitwise-OR com baixos bits **CHAR_BIT**. O número de bytes na constante de caracteres multibyte não pode exceder sizeof(int), que é 4 para o código de 32 bits de destino.

A constante de caracteres multibyte é lida como acima e é convertida em uma constante de caractere largo usando a função de tempo de execução `mbtowc`. Se o resultado não for uma constante de caractere largo válida, um erro será emitido. Em todo caso, o número de bytes examinados pela função `mbtowc` é limitado ao valor de `MB_CUR_MAX`.

Veja também

Caracteres

Convertendo caracteres multibyte

13/05/2021 • 2 minutes to read

ANSI 3.1.3.4 A localidade atual usada para converter caracteres multibyte nos caracteres largos (código) correspondentes para uma constante de caracteres largos

A localidade atual for a localidade de "C" por padrão. Ela pode ser alterada com o #pragma setlocale.

Veja também

Caracteres

Intervalo de valores char

13/05/2021 • 2 minutes to read

3.2.1.1 ANSI Se um "Plain" `char` tem o mesmo intervalo de valores que um `signed char` ou um `unsigned char`

Todos os valores de caracteres com sinal variam de -128 a 127. Todos os valores de caracteres sem sinal variam de 0 a 255.

A `/J` opção do compilador altera o tipo padrão para `char` de `signed char` para `unsigned char` .

Veja também

Caracteres

Inteiros

13/05/2021 • 2 minutes to read

- Intervalo de valores inteiros
- Rebaixamento de inteiros
- Operações bits assinadas
- Restantes
- Deslocamentos à direita

Veja também

Comportamento definido pela implementação

Intervalo de valores inteiros

13/05/2021 • 2 minutes to read

ANSI 3.1.2.5 As representações e os conjuntos de valores de vários tipos de inteiros

Inteiros contêm 32 bits (quatro bytes). Os inteiros com sinal são representados no formato de dois complementos. O bit mais significativo contém o sinal: 1 para o negativo, 0 para o sinal positivo e zero. Os valores são listados abaixo:

TIPO	MÍNIMO E MÁXIMO
`unsigned short`	0 a 65535
`signed short`	-32768 a 32767
`unsigned long`	0 a 4294967295
`signed long`	-2147483648 a 2147483647

Veja também

Inteiros

Rebaixamentos de inteiros

13/05/2021 • 2 minutes to read

ANSI 3.2.1.2 O resultado da conversão de um inteiro para um número inteiro com sinal mais curto ou o resultado da conversão de um número inteiro sem sinal para um número inteiro com sinal de mesmo comprimento, se o valor não puder ser representado

Quando um `long` inteiro é convertido em um `short` , ou um `short` é convertido em um `char` , os bytes menos significativos são mantidos.

Por exemplo, esta linha

```
short x = (short)0x12345678L;
```

atribui o valor 0x5678 a `x` , e esta linha

```
char y = (char)0x1234;
```

atribui o valor 0x34 a `y` .

Quando `signed` as variáveis são convertidas `unsigned` e vice-versa, os padrões de bit permanecem os mesmos. Por exemplo, a conversão-2 (0xFE) em um `unsigned` valor produz 254 (também 0xFE).

Veja também

Inteiros

Operações bit a bit com sinal

13/05/2021 • 2 minutes to read

ANSI 3.3 Os resultados de operações bit a bit em inteiros com sinal

As operações bit a bit em inteiros com sinal funcionam da mesma forma que as operações bit a bit em inteiros sem sinal. Por exemplo, `-16 & 99` pode ser expresso em binário como

```
  11111111 11110000
& 00000000 01100011
  ------------------
  00000000 01100000
```

O resultado do AND bit a bit é 96.

Veja também

Inteiros

Restantes

13/05/2021 • 2 minutes to read

ANSI 3.3.5 O sinal do resto na divisão de um inteiro

O sinal do restante é o mesmo sinal do dividendo. Por exemplo:

```
50 / -6 == -8
50 % -6 ==  2
-50 /  6 == -8
-50 %  6 == -2
```

Veja também

Inteiros

Deslocamentos para a direita

13/05/2021 • 2 minutes to read

O resultado de um deslocamento à direita de um valor negativo tipo integral com sinal

O deslocamento de um valor negativo para a direita gera metade do valor absoluto, arredondado para baixo. Por exemplo, um `signed short` valor de-253 (Hex 0xFF03, binary 11111111 00000011) foi deslocado para a direita um bit produz-127 (Hex 0xFF81, binary 11111111 10000001). Um 253 positivo deslocado à direita gera + 126.

Deslocamentos para a direita preservam o bit de sinal de tipos integrais com sinal. Quando um inteiro assinado é deslocado para a direita, o bit mais significativo permanece definido. Por exemplo, se 0xF0000000 for assinado `int` , um deslocamento à direita produzirá 0xF8000000. A mudança de um `int` à direita negativo de 32 vezes produz 0xFFFFFFFF.

Quando um inteiro não assinado é deslocado para a direita, o bit mais significativo é limpo. Por exemplo, se 0xF000 não tiver sinal, o resultado será 0x7800. A mudança de um `unsigned` or positivo para a `int` direita 32 vezes produz 0x00000000.

Veja também

Inteiros

Matemática de ponto flutuante

13/05/2021 • 2 minutes to read

- Valores
- Conversão de inteiros em valores de Floating-Point
- Truncamento de valores de Floating-Point

Veja também

Comportamento definido pela implementação

Valores

13/05/2021 • 2 minutes to read

ANSI 3.1.2.5 As representações e os conjuntos de valores de vários tipos de números de ponto flutuante

O `float` tipo contém 32 bits: 1 para o sinal, 8 para o expoente e 23 para o mantissa. O intervalo é +/- 3.4E38 com pelo menos 7 dígitos de precisão.

O `double` tipo contém 64 bits: 1 para o sinal, 11 para o expoente e 52 para o mantissa. O intervalo é +/- 1.7E308 com pelo menos 15 dígitos de precisão.

O `long double` tipo é distinto, mas tem a mesma representação que o tipo `double` no compilador do Microsoft C.

Veja também

Matemática de ponto flutuante

Convertendo inteiros em valores de ponto flutuante

13/05/2021 • 2 minutes to read

ANSI 3.2.1.3 A direção de truncamento quando um número integral é convertido em um número de ponto flutuante que não pode representar exatamente o valor original

Quando um número integral é convertido em um valor de ponto flutuante que não pode representar o valor com exatidão, o valor é arredondado (para cima ou para baixo) para o valor apropriado mais próximo.

Por exemplo, a conversão de um `unsigned long` (com 32 bits de precisão) em um `float` (cujo mantissa tem 23 bits de precisão) arredonda o número para o múltiplo mais próximo de 256. Os `long` valores de 4.294.966.913 a 4.294.967.167 são todos arredondados para o `float` valor 4.294.967.040.

Veja também

Matemática de ponto flutuante

Truncamento de valores de ponto flutuante

13/05/2021 • 2 minutes to read

ANSI 3.2.1.4 A direção de truncamento ou arredondamento quando um número de ponto flutuante é convertido em um número de ponto flutuante mais restrito

Quando ocorre um estouro negativo, o valor de uma variável de ponto flutuante é arredondado para zero. Um estouro pode causar um erro de tempo de execução ou pode gerar um valor imprevisível, dependendo das otimizações especificadas.

Veja também

Matemática de ponto flutuante

Matrizes e ponteiros

13/05/2021 • 2 minutes to read

- Maior tamanho de matriz
- Subtração de ponteiro

Veja também

Comportamento definido pela implementação

Maior tamanho da matriz

13/05/2021 • 2 minutes to read

ANSI 3.3.3.4, 4.1.1 O tipo de inteiro necessário para manter o tamanho máximo de uma matriz – ou seja, o tamanho de **size_t**

O `size_t` typedef é um `unsigned int` na plataforma x86 de 32 bits. Em plataformas de 64 bits, `size_t` typedef é um `unsigned __int64` .

Veja também

Matrizes e ponteiros

Subtração do ponteiro

13/05/2021 • 2 minutes to read

ANSI 3.3.6, 4.1.1 O tipo de inteiro necessário para manter a diferença entre dois ponteiros em relação aos elementos da mesma matriz, **ptrdiff_t**

O `ptrdiff_t` typedef é um `int` na plataforma x86 de 32 bits. Em plataformas de 64 bits, `ptrdiff_t` typedef é um `__int64` .

Veja também

Matrizes e ponteiros

Registros: disponibilidade de registros

13/05/2021 • 2 minutes to read

ANSI 3.5.1 A extensão até a qual os objetos podem ser realmente colocados nos registros por meio de um especificador de classe de armazenamento do registro

O compilador não honra as solicitações do usuário de variáveis de registro. Em vez disso, ele faz suas próprias escolhas na otimização.

Veja também

Comportamento definido pela implementação

Estruturas, uniões, enumerações e campos de bit

13/05/2021 • 2 minutes to read

- Acesso impróprio a uma União
- Preenchimento e alinhamento dos membros da estrutura
- Campos de sinal de bits
- Armazenamento de campos de bits
- O tipo enum

Veja também

Comportamento definido pela implementação

Acesso inadequado a uma união

13/05/2021 • 2 minutes to read

ANSI 3.3.2.3 Um membro de um objeto de união é acessado usando um membro de um tipo diferente

Se uma união de dois tipos é declarada e um valor é armazenado, mas a união é acessada com o outro tipo, os resultados são não confiáveis.

Por exemplo, uma União de `float` e `int` é declarada. Um `float` valor é armazenado, mas o programa mais tarde acessa o valor como um `int` . Nessa situação, o valor dependeria do armazenamento interno de `float` valores. O valor inteiro não seria confiável.

Veja também

Estruturas, uniões, enumerações e campos de bits

Preenchimento e alinhamento de membros da estrutura

13/05/2021 • 2 minutes to read

ANSI 3.5.2.1 O preenchimento e alinhamento dos membros das estruturas e se um campo de bit pode transpor um limite de unidade de armazenamento

Os membros da estrutura são armazenados em sequência na ordem em que são declarados: o primeiro membro tem o endereço de memória mais baixo e o último membro, o mais alto.

Cada objeto de dados tem um requisito alignment-requirement. O requisito de alinhamento para todos os dados, exceto estruturas, uniões e matrizes, é o tamanho do objeto ou o tamanho do pacote atual (especificado com /Zp ou o pragma `pack`, o que for menor.) Para estruturas, uniões e matrizes, o requisito de alinhamento é o maior requisito de alinhamento de seus membros. Um offset (deslocamento) é alocado para cada objeto de modo que

deslocamento **%** *alinhamento-requisito* **= = 0**

Os campos de bits adjacentes serão empacotados na mesma unidade de alocação de 1, 2, ou 4 bytes se os tipos integrais forem do mesmo tamanho e se o campo de bit seguinte se encaixar na unidade de alocação atual sem cruzar o limite imposto pelos requisitos comuns de alinhamento dos campos de bits.

Veja também

Estruturas, uniões, enumerações e campos de bits

Sinal de campos de bits

13/05/2021 • 2 minutes to read

3.5.2.1 ANSI Se um campo "simples" `int` é tratado como um `signed int` campo de bits ou como um campo de bit int não assinado

Os campos de bits podem ser assinados ou não assinados. Os campos de bits simples são tratados como assinados.

Veja também

Estruturas, uniões, enumerações e campos de bits

Armazenamento de campos de bits

13/05/2021 • 2 minutes to read

ANSI 3.5.2.1 A ordem de alocação de campos de bits dentro de um int

Os campos de bit são alocados em um inteiro do bit menos significativo ao bit mais significativo. No código a seguir

```
struct mybitfields
{
   unsigned a : 4;
   unsigned b : 5;
   unsigned c : 7;
} test;

int main( void )
{
   test.a = 2;
   test.b = 31;
   test.c = 0;
}
```

os bits seriam organizados como segue:

```
00000001 11110010
cccccccb bbbbaaaa
```

Como os processadores 80x86 armazenam o byte inferior de valores inteiros antes do byte superior, o inteiro 0x01F2 acima seria armazenado na memória física como 0xF2 seguido por 0x01.

Veja também

Estruturas, uniões, enumerações e campos de bits

Tipo enum

13/05/2021 • 2 minutes to read

ANSI 3.5.2.2 O tipo de inteiro escolhido para representar os valores de um tipo de enumeração

Uma variável declarada como `enum` é um `int` .

Veja também

Estruturas, uniões, enumerações e campos de bits

Qualificadores: acesso a objetos voláteis

13/05/2021 • 2 minutes to read

ANSI 3.5.5.3 O que constitui um acesso a um objeto do tipo qualificado como volátil

Qualquer referência a um tipo qualificado como volátil é um acesso.

Veja também

Comportamento definido pela implementação

Declaradores: número máximo

13/05/2021 • 2 minutes to read

ANSI 3.5.4 O número máximo de declaradores que podem modificar uma aritmética, uma estrutura ou um tipo de união

O Microsoft C não limita o número de declaradores. O número é limitado somente pela memória disponível.

Veja também

Comportamento definido pela implementação

Instruções: limites em instruções switch

13/05/2021 • 2 minutes to read

3.6.4.2 ANSI O número máximo de `case` valores em uma `switch` instrução

O Microsoft C não limita o número de `case` valores em uma `switch` instrução. O número é limitado somente pela memória disponível.

Veja também

Comportamento definido pela implementação

Pré-processando diretivas

13/05/2021 • 2 minutes to read

- Constantes de caractere e inclusão condicional
- Incluindo nomes de filecolchetes
- Incluindo nomes de fileentre aspas
- Sequências de caracteres
- Pragmas
- Data e hora padrão

Veja também

Comportamento definido pela implementação

Constantes de caractere e inclusão condicional

13/05/2021 • 2 minutes to read

ANSI 3.8.1 Se o valor de uma constante de caractere único em uma expressão constante que controla a inclusão condicional corresponde ao valor da mesma constante de caractere no conjunto de caracteres de execução. Se essa constante de caractere pode ter um valor negativo

O conjunto de caracteres usado em instruções do pré-processador é igual ao conjunto de caracteres de execução. O pré-processador reconhece valores negativos de caracteres.

Veja também

Diretivas de pré-processamento

Incluindo nomes de arquivo entre colchetes

13/05/2021 • 2 minutes to read

ANSI 3.8.2 O método para localizar arquivos de origem incluíveis

Para as especificações de arquivo colocadas entre colchetes angulares, o pré-processador não pesquisa diretórios dos arquivos pai. Um arquivo "pai" é o arquivo que contém a diretiva #include. Em vez disso, ele começa procurando o arquivo nos diretórios especificados na linha de comando do compilador depois de /I. Se a opção /I não estiver presente ou falhar, o pré-processador usará a variável de ambiente INCLUDE para localizar os arquivos de inclusão entre colchetes angulares. A variável de ambiente INCLUDE pode conter vários caminhos separados por ponto e vírgula (;). Se mais de um diretório aparecer como parte da opção /I ou dentro da variável de ambiente INCLUDE, o pré-processador procurará por eles na ordem em que aparecem.

Veja também

Diretivas de pré-processamento

Incluindo nomes de arquivo entre aspas

13/05/2021 • 2 minutes to read

ANSI 3.8.2 O suporte para nomes entre aspas para arquivos de origem incluíveis

Se você indicar uma especificação de caminho completa e inequívoca para o arquivo de inclusão entre aspas duplas (" "), o pré-processador pesquisará apenas essa especificação de caminho e ignorará os diretórios padrão.

Para arquivos de inclusão especificados como #include "path-spec", a pesquisa de diretórios começa com os diretórios do arquivo pai e continua pelos diretórios dos arquivos avô, se houver. Assim, a pesquisa começa em relação ao diretório que contém o arquivo de origem que está sendo processado. Se não houver nenhum arquivo avô e o arquivo não tiver sido encontrado, a pesquisa continuará como se o nome do arquivo estivesse entre colchetes angulares.

Veja também

Diretivas de pré-processamento

Sequências de caracteres

13/05/2021 • 2 minutes to read

ANSI 3.8.2 O mapeamento de sequências de caracteres do arquivo de origem

As instruções do pré-processador usam o mesmo conjunto de caracteres das instruções de arquivo de origem, com exceção das sequências de escape, que não têm suporte.

Assim, para especificar um caminho para um arquivo de inclusão, use somente uma barra invertida:

```
#include "path1\path2\myfile"
```

No código-fonte, duas barras invertidas são necessárias:

```
fil = fopen( "path1\\path2\\myfile", "rt" );
```

Veja também

Diretivas de pré-processamento

Pragmas

13/05/2021 • 2 minutes to read

ANSI 3.8.6 O comportamento em cada política #pragma reconhecida.

As seguintes Pragmas C são definidas para o compilador Microsoft C:

alloc_text
auto_inline
check_stack
code_seg
mente

data_seg
funcionamento
hdrstop
include_alias

inline_depth
inline_recursion
controlos
message

formato
pacotes
setlocale
warning

Veja também

Diretivas de pré-processamento

Data e hora padrão

13/05/2021 • 2 minutes to read

ANSI 3.8.8 As definições para `__DATE__` e `__TIME__` quando, respectivamente, a data e a hora da tradução não estão disponíveis

Quando o sistema operacional não fornece a data e a hora da tradução, os valores padrão para `__DATE__` e `__TIME__` são `May 03 1957` e `17:00:00`.

Veja também

Diretivas de pré-processamento

Funções de biblioteca

13/05/2021 • 2 minutes to read

- Macro nula
- Diagnóstico impresso pela função Assert
- Teste de caracteres
- Erros de domínio
- Estouro negativo de valores de Floating-Point
- A função fmod
- A função signal
- Sinais padrão
- Terminando caracteres de nova linha
- Linhas em branco
- Caracteres nulos
- Posição do arquivo no modo de acréscimo
- Truncamento de arquivos de texto
- Buffer de arquivo
- Arquivos de comprimento zero
- Nomes de arquivos
- Limites de acesso a arquivos
- Excluindo arquivos abertos
- Renomeando com um nome que existe
- Lendo valores de ponteiro
- Lendo intervalos
- Erros de posição de arquivo
- Mensagens geradas pela função perror
- Alocando memória zero
- A função abort
- A função atexit
- Nomes de ambiente
- A função system
- A função strerror

- O fuso horário
- A função clock

Veja também

Comportamento definido pela implementação

Macro NULL

13/05/2021 • 2 minutes to read

ANSI 4.1.5 A constante de ponteiro nulo para a qual a macro NULL se expande

Vários arquivos de inclusão definem a macro NULL como `((void *)0)`.

Veja também

Funções de biblioteca

Diagnóstico impresso pela função assert

13/05/2021 • 2 minutes to read

ANSI 4.2 O diagnóstico impresso pela função **assert** e seu comportamento de término

A função **assert** imprimirá uma mensagem de diagnóstico e chamará a rotina **abort** se a expressão for false (0). A mensagem de diagnóstico tem o formato

> **Assertion failed**: *expressão*, **arquivo** *nome_de_arquivo*, **linha** *número_de_linha*

em que *nome_de_arquivo* é o nome do arquivo de origem e *número_de_linha* é o número de linha da asserção que falhou no arquivo de origem. Nenhuma ação será executada se a *expressão* for true (diferente de zero).

Veja também

Funções de biblioteca

Testes de caractere

13/05/2021 • 2 minutes to read

4.3.1 do ANSI Os conjuntos de caracteres testados pelas `isalnum` funções, `isalpha` , `iscntrl` , `islower` , `isprint` e `isupper`

A lista a seguir descreve essas funções como são implementadas pelo compilador Microsoft C.

FUNÇÃO	TESTES PARA
`isalnum`	Caracteres 0 – 9, A–Z, a–z ASCII 48– 57, 65– 90, 97– 122
`isalpha`	Caracteres A–Z, a–z ASCII 65–90, 97–122
`iscntrl`	ASCII 0 – 31, 127
`islower`	Caracteres A–Z ASCII 97–122
`isprint`	Caracteres A–Z, a–z, 0 – 9, pontuação, espaço ASCII 32–126
`isupper`	Caracteres A–Z ASCII 65–90

Veja também

Funções de biblioteca

Erros de domínio

13/05/2021 • 2 minutes to read

ANSI 4.5.1 Os valores retornados pelas funções matemáticas em erros de domínio

O arquivo ERRNO. H define a constante de erro de domínio `EDOM` como 33. Consulte o tópico da Ajuda para a função específica que causou o erro para obter informações sobre o valor retornado.

Veja também

Funções de biblioteca

Estouro negativo de valores de ponto flutuante

13/05/2021 • 2 minutes to read

ANSI 4.5.1 Se as funções matemáticas definem a expressão de inteiro `errno` como o valor da macro `ERANGE` em erros de alcance de estouro negativo

Um estouro negativo de ponto flutuante não define a expressão `errno` como `ERANGE`. Quando um valor se aproxima de zero e acaba sofrendo um estouro negativo, o valor é definido como zero.

Veja também

Funções de biblioteca

Função fmod

13/05/2021 • 2 minutes to read

ANSI 4.5.6.4 Se um erro de domínio ocorrer ou se zero for retornado quando a função `fmod` tiver um segundo argumento de zero

Quando a função `fmod` tem um segundo argumento de zero, a função retorna zero.

Veja também

Funções de biblioteca

Função signal (C)

13/05/2021 • 2 minutes to read

ANSI 4.7.1.1 O conjunto de sinais para a função **signal**

O primeiro argumento transmitido para **signal** deve ser uma das constantes simbólicas descritas na *Referência da biblioteca em tempo de execução* da função **signal**. As informações na *Referência da biblioteca em tempo de execução* também listam o suporte ao modo de operação para cada sinal. As constantes também são definidas em SIGNAL.H.

Veja também

Funções de biblioteca

Sinais padrão

13/05/2021 • 2 minutes to read

ANSI 4.7.1.1 Se o equivalente a `signal(sig, SIG_DFL)` não for executado antes de uma chamada para o manipulador de sinal, o bloqueio de sinal será executado

No início da execução do programa, os sinais são definidos em seus status padrão.

Veja também

Funções de biblioteca

Encerrando caracteres de nova linha

13/05/2021 • 2 minutes to read

ANSI 4.9.2 Se a última linha de um fluxo de texto requer um caractere de nova linha de término

As funções de fluxo reconhecem a nova linha ou o fim de arquivo como o caractere de término de uma linha.

Veja também

Funções de biblioteca

Linhas em branco

13/05/2021 • 2 minutes to read

ANSI 4.9.2 Se os caracteres de espaço que são escritos em um fluxo de texto imediatamente antes de um caractere de nova linha aparecem quando lidos

Os caracteres de espaço são preservados.

Veja também

Funções de biblioteca

Caracteres nulos

13/05/2021 • 2 minutes to read

ANSI 4.9.2 O número de caracteres nulos que podem ser anexados aos dados gravados em um fluxo binário

Qualquer número de caracteres nulos pode ser anexado a um fluxo binário.

Veja também

Funções de biblioteca

A posição do arquivo no modo de acréscimo

13/05/2021 • 2 minutes to read

ANSI 4.9.3 Se o indicador da posição do arquivo de um fluxo de modo de acréscimo foi posicionado inicialmente no início ou no final do arquivo

Quando um arquivo é aberto no modo de anexação, o indicador de posição do arquivo aponta inicialmente para o final do arquivo.

Veja também

Funções de biblioteca

Truncamento de arquivos de texto

13/05/2021 • 2 minutes to read

ANSI 4.9.3 Se uma gravação em um fluxo de texto faz com que o arquivo associado seja truncado além desse ponto

Gravar em um fluxo de texto não trunca o arquivo além desse ponto.

Veja também

Funções de biblioteca

Armazenar arquivo em buffer

13/05/2021 • 2 minutes to read

ANSI 4.9.3 As características do buffer de arquivos

Os arquivos em disco acessados por meio de funções padrão de E/S são totalmente armazenados em buffer. Por padrão, o buffer contém 512 bytes.

Veja também

Funções de biblioteca

Arquivos de tamanho igual a zero

13/05/2021 • 2 minutes to read

ANSI 4.9.3 Se um arquivo com comprimento de zero realmente existe

Os arquivos com um comprimento de zero são permitidos.

Veja também

Funções de biblioteca

Nomes de arquivos

13/05/2021 • 2 minutes to read

ANSI 4.9.3 As regras para a criação de nomes de arquivo válidos

Uma especificação de arquivo pode incluir uma letra de unidade opcional (sempre seguida por dois pontos), uma série de nomes de diretórios opcionais (separados por barras invertidas) e um nome de arquivo.

Para obter mais informações, consulte Nomear um arquivo.

Veja também

Funções de biblioteca

Limites de acesso a arquivos

13/05/2021 • 2 minutes to read

ANSI 4.9.3 Se o mesmo arquivo pode ser aberto várias vezes

Não é permitido abrir um arquivo que já está aberto.

Veja também

Funções de biblioteca

Excluindo arquivos abertos

13/05/2021 • 2 minutes to read

ANSI 4.9.4.1 O efeito da função remove em um arquivo aberto

A função remove exclui um arquivo. Se o arquivo estiver aberto, essa função falhará e retornará -1.

Veja também

Funções de biblioteca

Renomear com um nome que existe

13/05/2021 • 2 minutes to read

ANSI 4.9.4.2 O efeito se um arquivo com o novo nome existir antes de uma chamada à função de **rename**

Se você tentar renomear um arquivo usando um nome que existe, a função **rename** falhará e retornará um código de erro.

Veja também

Funções de biblioteca

Valores de ponteiro de leitura

13/05/2021 • 2 minutes to read

ANSI 4.9.6.2 A entrada para a conversão de **%p** na função `fscanf`

Quando o caractere de formato **%p** é especificado, a função `fscanf` converte ponteiros de valores ASCII hexadecimais no endereço correto.

Veja também

Funções de biblioteca

Intervalos de leitura

13/05/2021 • 2 minutes to read

ANSI 4.9.6.2 A interpretação de um caractere de sublinhado (–) que não é o primeiro nem o último caractere da scanlist para % [conversão na função `fscanf`

A linha a seguir

```
fscanf( fileptr, "%[A-Z]", strptr);
```

lê qualquer número de caracteres de A a Z para a cadeia de caracteres para a qual `strptr` aponta.

Veja também

Funções de biblioteca

Erros de posição do arquivo

13/05/2021 • 2 minutes to read

ANSI 4.9.9.1, 4.9.9.4 O valor para o qual a macro `errno` é definida pela função `fgetpos` ou `ftell` em caso de falha

Quando `fgetpos` ou `ftell` falha, `errno` é definido como a constante de manifesto `EINVAL` caso a posição seja inválida ou `EBADF`, caso o número do arquivo esteja incorreto. As constantes também são definidas em ERRNO.H.

Veja também

Funções de biblioteca

Mensagens geradas pela função perror

13/05/2021 • 2 minutes to read

ANSI 4.9.10.4 As mensagens geradas pela função `perror`

A função `perror` gera essas mensagens:

```
0  Error 0
1
2  No such file or directory
3
4
5
6
7  Arg list too long
8  Exec format error
9  Bad file number
10
11
12 Not enough core
13 Permission denied
14
15
16
17 File exists
18 Cross-device link
19
20
21
22 Invalid argument
23
24 Too many open files
25
26
27
28 No space left on device
29
30
31
32
33 Math argument
34 Result too large
35
36 Resource deadlock would occur
```

Veja também

Funções de biblioteca

Alocando memória igual a zero

13/05/2021 • 2 minutes to read

4.10.3 ANSI O comportamento da `calloc` função, `malloc` ou `realloc` se o tamanho solicitado for zero

As funções `calloc`, `malloc` e `realloc` aceitam zero como argumento. Nenhuma memória real é alocada, mas um ponteiro válido é retornado e o bloco de memória pode ser modificado mais tarde por realloc.

Veja também

Funções de biblioteca

Função abort (C)

13/05/2021 • 2 minutes to read

ANSI 4.10.4.1 O comportamento da função **abort** em relação aos arquivos abertos e temporários

A função **abort** não fecha os arquivos que estão abertos ou são temporários. Ela não libera os buffers de fluxo. Para obter mais informações, confira abort.

Veja também

Funções de biblioteca

Função atexit (C)

13/05/2021 • 2 minutes to read

ANSI 4.10.4.3 O status retornado pela função `atexit` se o valor do argumento for diferente de zero, EXIT_SUCCESS ou EXIT_FAILURE

A função `atexit` retorna zero se tiver êxito ou um valor diferente de zero, se não for bem-sucedida.

Veja também

Funções de biblioteca

Nomes de ambientes

13/05/2021 • 2 minutes to read

ANSI 4.10.4.4 O conjunto de nomes de ambiente e o método para alterar a lista de ambientes usada pela função getenv

O conjunto de nomes de ambientes é ilimitado.

Para alterar as variáveis de ambiente de um programa C, chame a função _putenv. Para alterar as variáveis de ambiente de linha de comando no Windows, use o comando SET (por exemplo, SET LIB = D:\ LIBS).

As variáveis de ambiente definidas dentro de um programa C só existirão enquanto sua cópia do host de shell de comando do sistema operacional estiver em execução (CMD.EXE ou COMMAND.COM). Por exemplo, a linha

```
system( SET LIB = D:\LIBS );
```

executará uma cópia do shell de comando (CMD.EXE), definirá a variável de ambiente LIB e retornará ao programa C, saindo da cópia secundária de CMD.EXE. Sair dessa cópia de CMD.EXE remove a variável de ambiente temporária LIB.

Da mesma forma, as alterações feitas pela função `_putenv` durará apenas até o encerramento do programa.

Veja também

Funções de biblioteca
_putenv, _wputenv
getenv, _wgetenv

Função system

13/05/2021 • 2 minutes to read

ANSI 4.10.4.5 O conteúdo e o modo de execução de cadeia de caracteres pela função **system**

A função **system** executa um comando interno do sistema operacional ou um arquivo .EXE, .COM (.CMD no Windows NT) ou .BAT em um programa C em vez da linha de comando.

A função system localiza o interpretador de comandos, que normalmente é CMD.EXE no sistema operacional Windows NT ou em COMMAND.COM no Windows. A função system passará a cadeia de caracteres do argumento ao interpretador de comandos.

Para obter mais informações, consulte system, _wsystem.

Veja também

Funções de biblioteca

Função strerror

13/05/2021 • 2 minutes to read

ANSI 4.11.6.2 O conteúdo das cadeias de caracteres da mensagem de erro retornada pela `strerror` função

A função `strerror` gera essas mensagens:

```
0   Error 0
1
2   No such file or directory
3
4
5
6
7   Arg list too long
8   Exec format error
9   Bad file number
10
11
12  Not enough core
13  Permission denied
14
15
16
17  File exists
18  Cross-device link
19
20
21
22  Invalid argument
23
24  Too many open files
25
26
27
28  No space left on device
29
30
31
32
33  Math argument
34  Result too large
35
36  Resource deadlock would occur
```

Veja também

Funções de biblioteca

Fuso horário

13/05/2021 • 2 minutes to read

ANSI 4.12.1 O fuso horário local e o horário de verão

O fuso horário local é a Hora Oficial do Pacífico. O Microsoft C oferece suporte ao horário de verão.

Veja também

Funções de biblioteca

Função clock (C)

13/05/2021 • 2 minutes to read

ANSI 4.12.2.1 A era da função `clock`

A era da função `clock` começa (com um valor de 0) quando o programa C começa a ser executado. Ela retorna os tempos medidos em 1/**CLOCKS_PER_SEC** (que equivale a 1/1000 no Microsoft C).

Veja também

Funções de biblioteca

Referência de pré-processador C/C++

13/05/2021 • 2 minutes to read

A *referência do pré-processador do C/c++* 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 /Zc: pré-processador fornece um pré-processamento totalmente compatível e um pré-processador C17. Esse é o padrão quando você usa o sinalizador do compilador `/std:c11` ou `/std:c17` .

Nesta seção

Pré-processador
Fornece uma visão geral dos pré-processadores de conformidade tradicionais e novos.

Diretivas de pré-processador
Descreve as políticas, normalmente usadas para tornar os programas de origem fáceis de alterar e de compilar em ambientes de execução diferentes.

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 pelos padrões C e C++ e pelo 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 da linguagem C++
Fornece o material de referência para a implementação da Microsoft da linguagem C++.

Referência da linguagem C
Fornece o material de referência para a implementação da Microsoft da linguagem C.

Referência de Build do C/C++
Fornece links para tópicos que discutem opções de compilador e de vinculador.

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 de tempo de execução da Microsoft C (CRT)

13/05/2021 • 2 minutes to read

A biblioteca de tempo de execução da Microsoft fornece rotinas para a programação do sistema operacional Microsoft Windows. Essas rotinas automatizam muitas tarefas comuns de programação que não são fornecidas pelas linguagens C e C++.

Programas de exemplo estão incluídos nos tópicos de referência individuais para a maioria das rotinas na biblioteca.

Nesta seção

Rotinas de tempo de execução C universal por categoria
Fornece links para a biblioteca de tempo de execução por categoria.

Variáveis globais e tipos padrão
Fornece links para as variáveis globais e tipos padrão fornecidos pela biblioteca de tempo de execução.

Constantes globais
Fornece links para as constantes globais definidas pela biblioteca de tempo de execução.

Estado global
Descreve o escopo do estado global na biblioteca de tempo de execução C.

Mapeamentos de texto genérico
Fornece links para os mapeamentos de texto genérico definidos em Tchar.h.

Referência de função alfabética
Fornece links para as funções da biblioteca de tempo de execução C, organizadas em ordem alfabética.

Visões gerais da família de funções
Fornece links para as funções da biblioteca de tempo de execução C, organizadas por família de funções.

Cadeias de caracteres de idioma e país/região
Descreve como usar a `setlocale` função para definir o idioma e cadeias de caracteres de país/região.

Arquivos de biblioteca de tempo de execução C (CRT) e C++ Standard (STL) `.lib`
Lista de `.lib` arquivos que compõem as bibliotecas de tempo de execução C e suas opções de compilador e diretivas de pré-processador associadas.

Seções relacionadas

Rotinas de depuração
Fornece links para as versões de depuração das rotinas da biblioteca de tempo de execução.

Verificação de erros em tempo de execução
Fornece links para funções que dão suporte a verificações de erro em tempo de execução.

Comportamento da biblioteca de tempo de execução de DLLs e Visual C++
Discute o código de inicialização e o ponto de entrada usado para uma DLL.

Pura

Fornece links para usar o depurador do Visual Studio para corrigir erros de lógica em seu aplicativo ou procedimentos armazenados.