C++ Para Leigos

Folha de Cola

Entender e executar programação em C++, que é o padrão para linguagens orientadas a objetos, é mais fácil quando você conhece as expressões, as declarações e os operadores para efetuar cálculos.

Expressões e Declarações em Programação C++

Para efetuar um cálculo no programa C++, você precisa de uma expressão. Uma expressão é uma instrução que possui um valor e um tipo. No programa C++, uma declaração é uma instrução que define uma variável ou é o "tanque de segurança" para algum tipo de valor, como um número ou caractere.

Expressões

As expressões tomam uma das seguintes formas:

```
objName                          // para um objeto simples
operator expression              // para operadores unários
expr1 operator expr2             // para operadores binários
expr1 ? expr2 : expr3            // para o operador ternário
funcName([argument list]);       // para chamadas de função
```

Expressões Literais

Uma literal é uma forma de expressão constante. Os vários tipos de literais são definidos na seguinte tabela literal.

Exemplo	Tipo
1	int
1L	long int
1LL	long long int
1.0	double
1.0F	float
'1'	char
"a string"	char* (automaticamente terminado por um caractere nulo)
L "a string"	wchar_t*
u8 "this is a UTF-8 string with a UTF-8 character: \u2018"	char8_t*

u “this is a UTF-16 string with a UTF-16 character: \u2018”	char16_t*
U “this is a UTF-32 string with a UTF-32 character: \U00002018”	char32_t*
true, false	bool
0b101	binary (padrão C++ 2014)

Declarações

Declarações são tipos intrínsecos e definidos pelo usuário. Os tipos intrínsecos são:

```
[<signed | unsigned >]char
[<signed | unsigned >]wchar_t
[<signed | unsigned>] [<short | long | long long>] int
float
[long] double
bool
```

As declarações têm um destes formatos:

```
[<extern|static>][const] type var[=expression];            // variável
[<extern|static>][const] type array[size][={list}];        // array
[const] type object[(argument list)];                      // objeto
[const] type object [= {argument list}];                   // alternativo
[const] type * [const] ptr[=pointer expression];           // ponteiro
type& refName = object;                                    // referência
type fnName([argument list]);                              // função
```

A palavra-chave auto pode ser usada se C++ determinar o tipo de variável sozinho.

```
auto var = 1L;                                  // o tipo de var é long int
```

A palavra-chave decltype extrai o tipo de uma expressão. Esse tipo pode ser usado quando um nome de tipo for usado. Por exemplo, o seguinte exemplo usa decltype para declarar a segunda variável com o mesmo tipo de uma variável existente.

```
decltype(var1) var2; // o tipo de var2 é o mesmo de var1
```

Uma definição de função tem o seguinte formato:

```
// função simples
[<inline|constexpr>] type fnName(argument list) {...}
```

```
// função de membro definida fora da classe
[inline] type Class::func(argument list) [const] {...}
// construtores/destrutores também podem ser definidos fora da classe
Class::Class([argument list]) {...}
Class::~Class() {...}
//construtores/destrutores podem ser excluídos ou padronizados
// no lugar de definição
Class::Class([argument list]) = <delete|default>;
Class::~Class() = <delete|default>;
```

Um operador sobrecarregado se parece com uma definição de função. A maioria dos operadores sobrecarregados podem ser escritos como uma função de membro ou uma função simples. Quando escrito como função de membro, `*this` é presumidamente o primeiro argumento do operador.

```
MyClass& operator+(const MyClass& m1, const MyClass& m2); // simples
MyClass& MyClass::operator+(const MyClass& m2); // membro;
```

Os usuários também podem definir os seus próprios tipos usando a `class` ou a palavra-chave `struct`.

```
<struct | class> ClassName [ : [virtual] [public] BaseClass]
{
     <public|protected>:
     // construtor
     ClassName([arg list]) <[: member(val),...] {...} |;>
     ClassName() [= <delete|default>;]
     // destrutor
     [virtual] ~ClassName() <{...} | [=<delete|default>;>
     // membros de dados públicos
     type dataMemberName [= initialValue];
     // funções de membros públicas
     type memberFunctionName([arg list]) [{...}]
     // função de membro const
     type memberFunctionName([arg list]) const [{...}]
     // funções de membros virtuais
     virtual type memberFunctionName([arg list]) [{...}];
     // funções de membro puramente virtuais
```

```
    virtual type memberFunctionName([arg list]) = 0;
    // função que deve sobrescrever uma função de classe base
    type memberFunctionName([arg list]) override;
    // uma função que não pode ser sobrescrita em uma classe
    type memberFunctionName([arg list]) final;
};
```

Além disso, um construtor com somente um argumento pode ser marcado como `explicit`, que significa que ele não será usado em uma conversão implícita de um tipo para outro. Marcar um construtor como `default` significa "usar a definição padrão do construtor C++". Marcar um construtor como `delete` remove a definição padrão C++ do construtor.

C++ suporta dois tipos de tipos enumerados. O seguinte tipo de enumeração antiga não cria um tipo novo:

```
enum STATE {DC,                          // pega 0
            ALABAMA,                     // pega 1
            ALASKA,                      // pega 2
            ARKANSAS,                    // pega 3
            // ...e assim por diante
            };
int n = ALASKA;                          // ALASKA é do tipo int
```

Por padrão, uma entrada individual é um tipo `int` mas isso pode ser modificado no padrão C++ de 2011.

```
enum ALPHABET:char {A = 'a',        // pega 'a'
                    B,              // pega 'b'
                    C,              // pega 'c'
                                    // ...e assim por diante
                   };
char c = A;                         // A é do tipo char
```

C++ 2011 permite um segundo formato que cria um tipo novo:

```
// a seguinte enumeração define um tipo novo STATE
enum class STATE { DC,                   // pega 0
                   ALABAMA,              // pega 1
                   ALASKA,               // pega 2
                   ARKANSAS,             // pega 3
                   // ...e assim por diante
                 };
```

Folha de Cola

```
STATE s = STATE::ALASKA; // agora STATE é um tipo novo
// o seguinte usa um tipo sublinhado diferente
enum class ALPHABET:char {A = 'a', // pega 'a'
                          B,       // pega 'b'
                          C,       // pega 'c'
                                   // ...e assim por diante
                 };
ALPHABET c = ALPHABET::A;          // A é do tipo ALPHABET
```

Declarações template têm um formato um pouco diferente:

```
// o tipo T é fornecido pelo programador em uso
template <class T, {...}> type FunctionName([arg list])
template <class T, {...}> class ClassName { {...} };
```

Operadores em Programação C++

Todos os operadores em C++ executam alguma função definida. Esta tabela exibe o operador, a precedência (que determina quem vai primeiro), a cardinalidade e a associatividade no programa C++.

	Operador	Cardinalidade	Associatividade do Operador
Maior precedência	() [] -> .	unário	esquerda para direita
	! ~ + - ++ — & * (cast) sizeof	unário	esquerda para direita
	* / %	binário	esquerda para direita
	+ -	binário	esquerda para direita
	<< >>	binário	esquerda para direita
	== !=	binário	esquerda para direita
	&	binário	esquerda para direita
	^	binário	esquerda para direita
	|	binário	esquerda para direita
	&&	binário	esquerda para direita
	||	binário	esquerda para direita

	?:	ternário	direita para esquerda
	= *= /= %= += -= &= ^= |= <<= >>=	binário	direita para esquerda
Menor precedência	,	binário	esquerda para direita

Controle de Fluxo em Programação C++

As seguintes estruturas C++ direcionam o fluxo de controle pelo programa. Se você for um programador C++ experiente, a função destas estruturas serão familiares de outras linguagens.

IF

O seguinte comando avalia `booleanExpression`. Se ele avaliar como verdade (`true`), então o controle passa para `expressions1`. Se não, o controle passa para a opcional `expressions2`.

```
if (booleanExpression)
{
   expressions1;
}
[else
{
   expressions2;
}]
```

WHILE

O seguinte comando avalia `booleanExpression`. Se ele avaliar como `true`, então o controle passa para `expressions`. No final do bloco, o controle volta para `booleanExpression` e repete o processo.

```
while (booleanExpression)
{
    expressions;
}
```

Folha de Cola

DO...WHILE

O seguinte comando executa `expressions`. Ele avalia `booleanExpression`. Se avalia como `true`, o controle retorna para o topo do loop e repete o processo.

```
do
{
    expressions;
} while(booleanExpression);
```

FOR

O seguinte comando executa `initCommand` que pode ser uma expressão ou uma declaração de variável. Se ele avaliar como `true`, o controle passa para `expressions1`. Se `boolExpression` for `false`, então o controle passa para a primeira instrução depois da chave do loop `for`. Uma vez que as expressões estão completas, o controle passa para a expressão contida em `loopExpression` antes de retornar para `boolExpression` e repetir o processo. Se `initCommand` declarar uma nova variável, ele sai do `boolExpression` e repete o processo. Se `initCommand` declara uma nova variável, ela sai do escopo assim que o controle passa fora do loop.

```
for (initCommand; boolExpression; loopExpression)
{
    expressions;
}
```

FOR (EACH)

O padrão 2011 apresenta uma segunda forma de loop for algumas vezes conhecida como "for each" por causa de sua semelhança ao `foreach` encontrado em algumas outras linguagens. Nesta forma, a variável declarada em `declaration` leva o valor do primeiro membro da `list` e executa o bloco de `expressions`. Quando completa, a variável declarada pega o segundo valor da `list` e executa `expressions` novamente. Esse processo é repetido para cada valor na `list`.

```
for (declaration: list)
{
   expressions;
}
```

SWITCH

O seguinte comando avalia `integerExpression` e compara o resultado com cada um dos casos listados. Se o valor encontrado for igual ao valor de uma das constantes integrais, `val1`, `val2`, etc., o controle passa para o conjunto de expressões correspondentes e continua até encontrar uma pausa (`break`). Se a expressão não se igualar a nenhum dos valores, o controle passa para `expressionsN` seguindo `default`.

```
switch(integerExpression)
{
   case val1:
          expressions1;
          break;
   case val2:
          expressions2;
          break;
   [default:
          expressionsN;
   ]
}
```

BREAK, CONTINUE, GOTO

Um `continue` passa o controle imediatamente para o início do loop. Isso faz o loop continuar com a próxima iteração. Por exemplo, o loop seguinte processa números entre 1 e 20:

```
for(int i = 0; i < 20; i++)
{
    // se o número não é primo...
    if (!isPrime(i))
    {
       // ...pule para o próximo valor de i
       continue;
    }
    // continue processando
}
```

Um `break` passa o controle para o primeiro argumento após a chave de qualquer comando de loop. Isso causa a execução de saída do loop imediatamente. Por exemplo, o exemplo a seguir lê caracteres até um end-of-file ser encontrado:

```
while(true)
{
   // lê uma linha do objeto input
   input >> line;
   // se uma falha ou end-of-file ocorrer...
   if (cin.eof() || cin.fail())
   {
       // ...então sai do loop
       break;
   }
   //processa a linha
}
```

Uma `goto label` passa o controle para a indicação fornecida. O exemplo `break` acima poderia ter sido escrito desta forma:

```
while(true)
{
    // lê uma linha de objeto de entrada
    input >> line;
    // se uma falha ou end-of-file ocorrer...
    if (cin.eof() || cin.fail())
    {
       // ...então sai do loop
       goto exitLabel;
    }
    // processa a linha
}
exitLabel:
    // o controle continua aqui
```