

Capítulo 2a

Descrição de recursos de C++ não relacionados às classes

Comentários

O primeiro recurso apresentado é simplesmente uma forma alternativa de comentar o código. Em C++, os caracteres `//` iniciam um comentário que termina no fim da linha na qual estão estes caracteres. Por exemplo:

```
int main(void)
{
    return -1; // retorna o valor -1 para o sistema operacional
}
```

Declaração de variáveis

Em C as variáveis só podem ser declaradas no início de um bloco. Se for necessário usar uma variável no meio de uma função existem duas soluções: voltar ao início da função para declarar a variável ou abrir um novo bloco, simplesmente para possibilitar uma nova declaração de variável.

C++ não tem esta limitação, e as variáveis podem ser declaradas em qualquer ponto do código:

```
void main(void)
{
    int a;
    a = 1;
    printf("%d\n", a);
    // ...
    char b[] = "teste";
    printf("%s\n", b);
}
```

O objetivo deste recurso é minimizar declarações de variáveis não inicializadas. Se a variável pode ser declarada em qualquer ponto, ela pode ser sempre inicializada na própria declaração. Um exemplo comum é o de variáveis usadas apenas para controle de loops:

```
for (int i=0; i<20; i++) // ...
```

A variável *i* está sendo criada dentro do `for`, que é o ponto onde se sabe o seu valor inicial. O escopo de uma variável é desde a sua declaração até o fim do bloco corrente (fecha chaves). Na linha acima a variável *i* continua existindo depois do `for`, até o fim do bloco.

É interessante notar que esta característica é diferente de se abrir um novo bloco a cada declaração. Uma tentativa de declarar duas variáveis no mesmo bloco com o mesmo nome causa um erro, o que não acontece se um novo bloco `for` aberto.

Declaração de tipos

Em C++, as declarações abaixo são equivalentes:

```
struct a {
    // ...
};
typedef struct a {
    // ...
} a;
```

Ou seja, não é mais necessário o uso de `typedef` neste caso. A simples declaração de uma estrutura já permite que se use o nome sem a necessidade da palavra reservada `struct`, como mostrado abaixo:

```
struct a {};
```

```
void f(void)
{
```

```

    struct a a1; // em C o tipo é usado como "struct a"
    a          a2; // em C++ a palavra struct
                  // não é mais necessária
}

```

Declaração de uniões

Em C++ as uniões podem ser anônimas. Uma união anônima é uma declaração da forma:

```
union { lista dos campos };
```

Neste caso, os campos da união são usados como se fossem declarados no escopo da própria união:

```

void f(void)
{
    union { int a; char *str; };
    a = 1;
    // ...
    str = malloc( 10 * sizeof(char) );
}

```

a e *str* são campos de uma união anônima, e são usados como se tivessem sido declarados como variáveis da função. Mas na realidade as duas tem o mesmo endereço. Um uso mais comum para uniões anônimas é dentro de estruturas:

```

struct A {
    int tipo;
    union {
        int inteiro;
        float real;
        void *ponteiro;
    };
};

```

Os três campos da união podem ser acessados diretamente, da mesma maneira que o campo *tipo*.

Protótipos de funções

Em C++ uma função só pode ser usada se esta já foi declarada. Em C, o uso de uma função não declarada geralmente causava uma warning do compilador, mas não um erro. Em C++ isto é um erro.

Para usar uma função que não tenha sido definida antes da chamada — tipicamente chamada de funções entre módulos —, é necessário usar protótipos. Os protótipos de C++ incluem não só o tipo de retorno da função, mas também os tipos dos parâmetros:

```

void f(int a, float b); // protótipo da função f

void main(void)
{
    f(1, 4.5); // o protótipo possibilita a utilização de f aqui
}

void f(int a, float b)
{
    printf("%.2f\n", b+a*0.5);
}

```

Uma tentativa de utilizar uma função não declarada gera um erro de símbolo desconhecido.

Funções que não recebem parâmetros

Em C puro, um protótipo pode especificar apenas o tipo de retorno de uma função, sem dizer nada sobre seus parâmetros. Por exemplo,

```
float f(); // em C, não diz nada sobre os parâmetros de f
```

é um protótipo incompleto da função *f*. De acordo com a seção anterior, um protótipo incompleto não faz muito sentido. Na realidade, esta é uma das diferenças entre C e C++. Um compilador de C++

interpretará a linha acima como o protótipo de uma função que retorna um float e não recebe nenhum parâmetro. Ou seja, é exatamente equivalente a uma função (void):

```
float f(); // em C++ é o mesmo que float f(void);
```

Funções inline

Funções inline são comuns em C++, tanto em funções globais como em métodos. Estas funções tem como objetivo tornar mais eficiente, em relação à velocidade, o código que chama estas funções. Elas são tratadas pelo compilador quase como uma macro: a chamada da função é substituída pelo corpo da função. Para funções pequenas, isto é extremamente eficiente, já que evita geração de código para a chamada e o retorno da função.

O corpo de funções inline pode ser tão complexo quanto uma função normal, não há limitação alguma. Na realidade o fato de uma função ser inline ou não é apenas uma otimização; a semântica de uma função e sua chamada é a mesma seja ela inline ou não.

Este tipo de otimização normalmente só é utilizado em funções pequenas, pois funções inline grandes podem aumentar muito o tamanho do código gerado. Do ponto de vista de quem chama a função, não faz nenhuma diferença se a função é inline ou não.

Nem todas as funções declaradas inline são expandidas como tal. Se o compilador julgar que a função é muito grande ou complexa, ela é tratada como uma função normal. No caso de o programador especificar uma função como inline e o compilador decidir que ela será uma função normal, será sinalizada uma warning. O critério de quando expandir ou não funções inline não é definido pela linguagem C++, e pode variar de compilador para compilador.

Para a declaração de funções inline foi criada mais uma palavra reservada, inline. Esta deve preceder a declaração de uma função inline:

```
inline double quadrado(double x)
{
    return x * x;
}
```

Neste caso a função *quadrado* foi declarada como inline. A chamada desta função é feita normalmente:

```
{
    double c = quadrado( 7 );
    double d = quadrado( c );
}
```

Assim como a chamada é feita da mesma maneira, o significado também é o mesmo com ou sem inline. A diferença fica por conta do código gerado. O trecho de código acima será compilado como se fosse:

```
{
    double c = 7 * 7;
    double d = c * c;
}
```

Além de ser mais eficiente, possibilita, por parte do compilador, otimizações extras. Por exemplo, calcular automaticamente, durante a compilação, o resultado de 7*7, gerando código para uma simples atribuição de 49 a c.

Fazendo uma comparação com as macros, que são usadas em C para produzir este efeito, nota-se que funções inline as substituem com vantagens. A primeira é que uma macro é uma simples substituição de texto, enquanto que funções inline são elementos da linguagem, e são verificadas quanto a erros. Além disso, macros não podem ser usadas exatamente como funções, como mostra o exemplo a seguir:

```
#define quadrado(x) ((x)*(x))
void main(void)
{
    double a = 4;
    double b = quadrado(a++);
}
```

Antes da compilação, a macro será expandida para:

```
double b = ((a++)*(a++));
```

A execução desta linha resultará na atribuição de $4*5 = 20$ a b , além de incrementar duas vezes a variável a . Com certeza não era este o efeito desejado.

Assim como funções, métodos também podem ser declarados como inline. No caso de métodos não é necessário usar a palavra reservada inline. A regra é a seguinte: funções com o corpo declarado dentro da própria classe são tratadas como inline; funções declaradas na classe mas definidas fora não são inline.

Supondo uma classe *Pilha*, o método *vazia* é simples o suficiente para justificar uma função inline:

```
struct Pilha {
    elemPilha* topo;
    int vazia() { return topo==NULL; }
    void push(int v);
    int pop();
};
```

Alguns detalhes devem ser levados em consideração durante a utilização de funções inline. O que o compilador faz quando se depara com uma chamada de função inline? O corpo da função deve ser expandido no lugar da chamada. Isto significa que o corpo da função deve estar disponível para o compilador antes da chamada.

Um módulo C++ é composto por dois arquivos, o arquivo com as declarações exportadas (.h) e outro com as implementações (.c, .cc ou .cpp). Tipicamente, uma função exportada por um módulo tem o seu protótipo no .h e sua implementação no .c. Isso porque um módulo não precisa conhecer a implementação de uma função para usá-la. Mas isso só é verdade para funções não inline. Por causa disso, funções inline exportadas precisam ser implementadas no .h e não mais no .c.

Referências

Referência é um novo modificador que permite a criação de novos tipos derivados. Assim como pode-se criar um tipo ponteiro para um inteiro, pode-se criar uma referência para um inteiro. A declaração de uma referência é análoga à de um ponteiro, usando o caracter & no lugar de *.

Uma referência para um objeto qualquer é, internamente, um ponteiro para o objeto. Mas, diferentemente de ponteiros, uma variável que é uma referência é utilizada como se fosse o próprio objeto. Os exemplos deixarão estas idéias mais claras. Vamos analisar referências em três utilizações: como variáveis locais, como tipos de parâmetros e como tipo de retorno de funções.

Uma variável local que seja uma referência deve ser sempre inicializada; a não inicialização causa um erro de compilação. Como as referências se referenciam a objetos, a inicialização não pode ser feita com valores constantes:

```
{
    int a;          // ok, variável normal
    int& b = a;      // ok, b é uma referencia para a
    int& c;          // erro! não foi inicializada
    int& d = 12;     // erro! inicialização inválida
}
```

A variável b é utilizada como se fosse realmente um inteiro, não há diferença pelo fato de ela ser uma referência. Só que b não é um novo inteiro, e sim uma referência para o inteiro guardado em a . Qualquer alteração em a se reflete em b e vice versa. É como se b fosse um novo nome para a mesma variável a :

```
{
    int a = 10;
    int& b = a;
    printf("a=%d, b=%d\n", a, b); // produz a=10, b=10
    a = 3;
    printf("a=%d, b=%d\n", a, b); // produz a=3, b=3
    b = 7;
    printf("a=%d, b=%d\n", a, b); // produz a=7, b=7
}
```

No caso de a referência ser um tipo de algum argumento de função, o parâmetro será passado por referência, algo que não existia em C e era simulado passando-se ponteiros:

```
void f(int a1, int &a2, int *a3)
{
    a1 = 1; // altera cópia local
    a2 = 2; // altera a variável passada (b2 de main)
```

```

    *a3 = 3; // altera o conteúdo do endereço de b3
}

void main()
{
    int b1 = 10, b2 = 20, b3 = 30;
    f(b1, b2, &b3);
    printf("b1=%d, b2=%d, b3=%d\n", b1, b2, b3);
    // imprime b1=10, b2=2, b3=3
}

```

O efeito é o mesmo para *b2* e *b3*, mas repare que no caso de *b3* é passado o endereço explicitamente, e a função tem que tratar o parâmetro como tal.

Falta ainda analisar um outro uso de referências, quando esta aparece como tipo de retorno de uma função. Por exemplo:

```

int& f()
{
    static int global;
    return global; // retorna uma referência para a variável
}

void main()
{
    f() = 12; // altera a variável global
}

```

É importante notar que este exemplo é válido porque *global* é uma variável static de *f*, ou seja, é uma variável global com escopo limitado a *f*. Se *global* fosse uma variável local comum, o valor de retorno seria inválido, pois quando a função *f* terminasse a variável *global* não existiria mais, e portanto a referência seria inválida. Como um ponteiro perdido. Outras seções apresentam exemplos práticos desta utilização.

Alocação de memória

A alocação dinâmica de memória, que em C era tratada com as funções `malloc` e `free`, é diferente em C++. Programas C++ não precisam mais usar estas funções. Para o gerenciamento da memória, foram criados dois novos operadores, `new` e `delete`, que são duas palavras reservadas. O operador `new` aloca memória e é análogo ao `malloc`; `delete` desaloca memória e é análogo ao `free`. A motivação desta modificação ficará clara no estudo de classes, mais especificamente na parte de construtores e destrutores.

Por ser um operador da linguagem, não é mais necessário, como no `malloc`, calcular o tamanho da área a ser alocada. Outra preocupação não mais necessária é a conversão do ponteiro resultado para o tipo correto. O operador `new` faz isso automaticamente:

```

int * i1 = (int*)malloc(sizeof(int)); // C
int * i2 = new int; // C++

```

A alocação de um vetor também é bem simples:

```

int * i3 = (int*)malloc(sizeof(int) * 10); // C
int * i4 = new int[10]; // C++

```

A liberação da memória alocada é feita pelo operador `delete`. Este operador pode ser usado em duas formas; uma para desalocar um objeto e outra para desalocar um vetor de objetos:

```

free(i1); // alocado com malloc (C )
delete i2; // alocado com new (C++)
free(i3); // alocado com malloc (C )
delete [] i4; // alocado com new[] (C++)

```

A utilização de `delete` para desalocar um vetor, assim como a utilização de `delete[]` para desalocar um único objeto tem consequências indefinidas. A necessidade de diferenciação se explica pela existência de destrutores, apresentados mais à frente.

Valores default para parâmetros de funções

Em C++ existe a possibilidade de definir valores default para parâmetros de uma função. Por exemplo, uma função que imprime uma string na tela em alguma posição especificada. Se nenhuma posição for especificada, a string deve ser impressa na posição do cursor:

```
void impr( char* str, int x = -1, int y = -1)
{
    if (x == -1) x = wherex();
    if (y == -1) y = wherey();
    gotoxy( x, y );
    cputs( str );
}
```

Esta definição indica que a função *impr* tem três parâmetros, sendo que os dois últimos tem valores default. Quando uma função usa valores default, os parâmetros com default devem ser os últimos. A função acima pode ser utilizada das seguintes maneiras:

```
impr( "especificando a posição", 10, 10 ); // x=10, y=10
impr( "só x", 20 );                        // x=20, y=-1
impr( "nem x nem y" );                    // x=-1, y=-1
```

O uso deste recurso também envolve alguns detalhes. A declaração do valor default só pode aparecer uma vez. Isto deve ser levado em consideração ao definir funções com protótipos. A maneira correta especifica o valor default apenas no protótipo:

```
void impr( char* str, int x = -1, int y = -1 );
...
void impr( char* str, int x, int y )
{
    ...
}
```

Mas a regra diz apenas que os valores só aparecem em um lugar. Nada impede a seguinte construção:

```
void impr( char* str, int x, int y );
...
void impr( char* str, int x = -1, int y = -1 )
{
    ...
}
```

Nesse caso, antes da definição do corpo de *impr* ela será tratada como uma função sem valores default.

Sobrecarga de nomes de funções

Este novo recurso permite que um nome de função possa ter mais de um significado, dependendo do contexto. Ter mais de um significado quer dizer que um nome de função pode estar associado a várias implementações diferentes. Apesar disso soar estranho para pessoas familiarizadas com C, no dia a dia aparecem situações idênticas no uso da nossa linguagem. Consideremos o verbo tocar; podemos usá-lo em diversas situações. Podemos tocar violão, tocar um disco etc. Em C++, diz-se que o verbo tocar está sobrecarregado. Cada contexto está associado a um significado diferente, mas todos estão conceitualmente relacionados.

Um exemplo simples é uma função que imprime um argumento na tela. Seria interessante se pudéssemos usar esta função para vários tipos de argumentos, mas cada tipo exige uma implementação diferente. Com sobrecarga isto é possível; o compilador escolhe que implementação usar dependendo do contexto. No caso de funções o contexto é o tipo dos parâmetros (só os parâmetros, não os resultados):

```
display( "string" );
display( 123 );
display( 3.14159 );
```

Na realidade, a maioria das linguagens usa sobrecarga internamente, mas não deixa este recurso disponível para o programador. C é uma destas linguagens. Por exemplo:

```
a = a + 125;
b = 3.14159 + 1.17;
```

O operador + está sobrecarregado. Na primeira linha, ele se refere a uma soma de inteiros; na segunda a uma soma de reais. Estas somas exigem implementações diferentes, e é como se existissem duas funções, a primeira sendo

```
int +(int, int);
```

e a outra como

```
float +(float, float);
```

Voltando à função *display*, é preciso definir as várias implementações requeridas:

```
void display( char *v ) { printf("%s", v); }
void display( int   v ) { printf("%d", v); }
void display( float v ) { printf("%f", v); }
```

A simples declaração destas funções já tem todas as informações suficientes para o compilador fazer a escolha correta. Isto significa que as funções não são mais distinguidas apenas pelo seu nome, mas pelo nome e pelo tipo dos parâmetros.

A tentativa de declarar funções com mesmo nome que não possam ser diferenciadas pelo tipo dos parâmetros causa um erro:

```
void f(int a);
int  f(int b); // erro! redeclaração de f!!!
```

O uso misturado de sobrecarga e valores default para parâmetros também pode causar erros:

```
void f();
void f(int a = 0);

void main()
{
    f( 12 ); // ok, chamando f(int)
    f();     // erro!! chamada ambígua: f() ou f(int = 0)???
}
```

Repare que nesse caso o erro ocorre no uso da função, e não na declaração.

Parâmetros de funções não utilizados

A maioria dos compiladores gera uma warning quando um dos parâmetros de uma função não é utilizado no seu corpo. Por exemplo:

```
void f(int a)
{
    return 1;
}
```

Este é um caso típico. O aviso do compilador faz sentido, já que a não utilização do parâmetro *a* pode ser uma erro na lógica da função. Mas e se for exatamente esta a implementação da função *f*? À primeira vista pode parecer estranho uma função que não utilize algum de seus parâmetros, mas em programação com callbacks (isto ficará mais claro a partir da introdução de programação orientada a eventos) é comum as funções ignorarem alguns parâmetros. Se for este o caso, não é necessário conviver com as warnings do compilador. Basta omitir o nome do parâmetro que não será usado:

```
void f(int)
{
    return 1;
}
```

Esta é uma função que recebe um inteiro como parâmetro, mas este inteiro não é utilizado na implementação da função.

Operador de escopo

C++ possui um novo operador que permite o acesso a nomes declarados em escopos que não sejam o corrente. Por exemplo, considerando o seguinte programa C:

```
char *a;

void main(void)
{
    int a;
    a = 23;
    /* como acessar a variável global a??? */
}
```

A declaração da variável local *a* esconde a global. Apesar de a variável *a* global existir, não há como referenciar o seu nome, pois no escopo da função *main*, o nome *a* está associado à local *a*.

O operador de escopo possibilita o uso de nomes que não estão no escopo corrente, o que pode ser usado neste caso:

```
char *a;

void main(void)
{
    int a;
    a = 23;
    ::a = "abc";
}
```

A sintaxe deste operador é a seguinte:

```
escopo::nome,
```

onde *escopo* é o nome da classe onde está declarado no nome *nome*. Se *escopo* não for especificado, como no exemplo acima, o nome é procurado no espaço global.

Outros usos deste operador são apresentados nas seções sobre classes aninhadas, campos de estruturas static e métodos static.

Incompatibilidades entre C e C++

Teoricamente, um programa escrito em C é compatível com um compilador C++. Na realidade esta compatibilidade não é de 100%, pois só o fato de C++ ter mais palavras reservadas já inviabiliza esta compatibilidade total. Outros detalhes contribuem para que um programa C nem sempre seja também um programa C++ válido.

Serão discutidos os pontos principais destas incompatibilidades; tanto as diferenças na linguagem como a utilização de códigos C e C++ juntos em um só projeto. No entanto a discussão não será extensiva, pois algumas construções suspeitas podem levar a várias pequenas incompatibilidades, como por exemplo:

```
a = b /* comentário
      etc.          */ 2;
```

Extraídos os comentários, este código em C fica sendo:

```
a = b / 2;
```

Enquanto que em C++, o resultado é diferente:

```
a = b
etc. */ 2;
```

Casos como estes são incomuns e não vale a pena analisá-los.

Palavras reservadas

Talvez a maior incompatibilidade seja causada pelo simples fato que C++ tem várias outras palavras reservadas. Isto significa que qualquer programa C que declare um identificador usando uma destas palavras não é um programa C++ correto. Aqui está uma lista com as novas palavras reservadas[†]:

catch	new	template
class	operator	this
delete	private	throw
friend	protected	try
inline	public	virtual

A única solução para traduzir programas que fazem uso destas palavras é trocar os nomes dos identificadores, o que pode ser feito usando diretivas #define.

Exigência de protótipos

Um dos problemas que pode aparecer durante a compilação de um programa C está relacionado ao fato de que protótipos não são obrigatórios em C. Como C++ exige protótipos, o que era uma warning C pode se transformar em um erro C++.

Estes erros simplesmente forçam o programador a fazer uma coisa que já devia ter sido feita mesmo com o compilador C: usar protótipos para as funções.

```
void main(void)
{
    printf("teste"); // C:   warning! printf undeclared
                   // C++: error!  printf undeclared
}
```

Funções que não recebem parâmetros

Programas que utilizam protótipos incompletos, ou seja, protótipos que só declaram o tipo de retorno de uma função, podem causar erros em C++. Este tipo de protótipo é considerado obsoleto em C, mas ainda válido. Em C++, estes protótipos são interpretados como declarações de funções que não recebem parâmetros, o que certamente causará erros se a função receber algum:

```
float square(); // C:   protótipo incompleto, não se sabe nada
                //     sobre os parâmetros
                // C++: protótipo completo,
                //     não recebe parâmetros

void main(void)
{
    float a = square(3); // C:   ok
                       // C++: error! too many arguments
}
```

Estruturas aninhadas

Outra incompatibilidade pode aparecer em programas C que declaram estruturas aninhadas. Em C, só há um escopo para os tipos: o global. Por esse motivo, mesmo que uma estrutura tenha sido declarada dentro de outra, ou seja, aninhada, ela pode ser usada como se fosse global. Por exemplo:

```
struct S1 {
    struct S2 {
        int a;
    } b;
};

void main(void)
{
    struct S1 var1;
```

[†] Esta lista não é definitiva, já que constantemente novos recursos vão sendo adicionados ao padrão.

```
    struct S2 var2; // ok em C, errado em C++
}
```

Este é um programa correto em C. Em C++, uma estrutura só é válida no escopo em que foi declarada. No exemplo acima, por ter sido declarada dentro de *S1*, a estrutura *S2* só pode ser usada dentro de *S1*. Nesse caso, a tentativa de declarar a variável *var2* causaria um erro, pois *S2* não é um nome global.

Em C++, *S2* pode ser referenciada utilizando-se o operador de escopo:

```
struct S1::S2 var2; // solução em C++
```

Uso de bibliotecas C em programas C++

Esta seção discute um problema comum ao se tentar utilizar bibliotecas C em programas C++. A origem do problema é o fato de C++ permitir sobrecarga de nomes de funções. Vamos analisar o trecho de código abaixo:

```
void f(int);

void main(void)
{
    f(12);
}
```

O programa acima declara uma função *f* que será chamada com o argumento 12. O código final gerado para a chamada da função terá uma linha assim:

```
call nome-da-função-escolhida
```

Este nome deve bater com o nome gerado durante a compilação da função *f*, seja qual for o módulo a que ela pertence. Caso isto não aconteça, a linkedição do programa sinalizará erros de símbolos indefinidos. Isto significa que existem regras para a geração dos nomes dos identificadores no código gerado. Em C, esta regra é simples: os símbolos têm o mesmo nome do código fonte com o prefixo `_`. No caso acima, o código gerado seria:

```
call _f
```

Entretanto, C++ não pode usar esta mesma regra simples, porque um mesmo nome pode se referir a várias funções diferentes pelo mecanismo de sobrecarga. Basta imaginar que poderíamos ter:

```
void f(int);
void f(char*);
```

Nesse caso a regra de C não é suficiente para diferenciar as duas funções. O resultado é que C++ tem uma regra complexa para geração dos nomes, que envolve codificação dos tipos também. O compilador Borland C++ 4.0 codifica as funções acima, respectivamente, como:

```
@f$qi
@f$qpc
```

Isto deve ser levado em consideração na compilação da chamada da função. A chamada de *f* com o argumento 12 pode gerar dois códigos diferentes. Se a função *f* for na realidade uma função compilada em C, o seu nome será `_f`, enquanto que se esta for uma função C++, o seu nome será `@f$qi`. Qual deve ser então o código gerado? Qual nome deve ser chamado?

A única maneira de resolver este problema é indicar, no protótipo da função, se esta é uma função C ou C++. No exemplo acima, o compilador assume que a função é C++. Se *f* for uma função C, o seu protótipo deve ser:

```
extern "C" void f(int);
```

A construção `extern "C"` pode ser usada de outra maneira, para se aplicar a várias funções de uma vez:

```
extern "C"{
    void f(int);
    void f(char);
    // ...
}
```

Dentro de um bloco extern pode aparecer qualquer tipo de declaração, não apenas funções. A solução então é alterar os header files das bibliotecas C, explicitando que as funções não são C++. Para isso, basta envolver todas as declarações em um bloco destes.

Como uma declaração extern “C” só faz sentido em C++, ela causa um erro de sintaxe em C. Para que os mesmos header files sejam usados em programas C e C++, basta usar a macro pré-definida `__cplusplus`, definida em todo compilador C++. Com estas alterações, um header file de uma biblioteca C terá a seguinte estrutura:

```
#ifdef __cplusplus
extern "C" {

// todas as declarações
// ...

#ifdef __cplusplus
}
#endif
```

Exercício 2 - Calculadora RPN com novos recursos de C++

Analisar a calculadora alterando-a de acordo com os novos recursos de C++