12. SOBRECARGA DE OPERADOR

Uma forma de C++ alcançar o polimorfismo é por meio do uso de sobrecarga de função. A sobrecarga, simplificadamente, consiste na redefinição de itens já existentes. Em outras palavras, ela permite que sejam definidas duas ou mais funções com o mesmo nome, desde que suas listas de argumentos sejam suficientemente diferentes para as chamadas a serem resolvidas. Nesta situação, quando diversas declarações de funções diferentes são especificadas por um único nome no mesmo escopo, diz-se que as funções que compartilham o mesmo nome estão sobrecarregadas. Quando esse nome é utilizado, a função correta é selecionada pela comparação dos tipos dos argumentos reais com os tipos dos argumentos formais.

Um exemplo prático de sobrecarga de funções é ilustrado pelo programa a seguir. C e C++ não contêm funções de bibliotecas que pedem ao usuário para digitar e aguardar uma resposta. Entretanto, este programa cria três funções chamadas "prompt()" que efetuam essa tarefa para dados do tipo *int*, *double* e *long*:

```
#include <iostream>
using namespace std;
void prompt (char *str, int *i);
void prompt (char *str, double *d);
void prompt (char *str, long *l);
main() {
  int i;
  double d;
  long l;
  prompt("Digite um inteiro: ", &i);
  prompt("Digite um double: ", &d);
  prompt("Digite um long: ", &l);
  cout << i << " " << d << " " << l;
}
void prompt (char *str, int *i) {
  cout << str;
  cin >> *i;
}
void prompt (char *str, double *d) {
  cout << str;
  cin >> *d;
}
void prompt (char *str, long *l) {
  cout << str;
  cin >> *l;
}
```

A sobrecarga de funções é importante porque pode ajudar a gerenciar a complexidade. Para compreender como, considere o seguinte exemplo. O Borland C++ contém as funções "atoi()", "atof()" e "atol()" em sua biblioteca padrão. Coletivamente, essas funções convertem uma *string* para diferentes tipos de números equivalentes (*int*, *double* e *long*). Embora estas funções realizem ações quase idênticas, em C três nomes diferentes precisam ser usados para representar essas tarefas, o que torna a situação mais complexa do que realmente é. Em outras palavras, mesmo que o conceito subjacente de cada função seja o mesmo, o programador deve lembrar de três nomes. Entretanto, em C++ é possível usar o mesmo nome para todas as três funções. Assim, o nome representa a ação geral que está sendo realizada e o compilador escolhe a versão específica para uma determinada circunstância. Portanto, aplicando-se sobrecarga, três nomes para memorizar são reduzidos a um.

Esta seção apresenta a sintaxe e a semântica para a sobrecarga de operadores. Baseando-se nos conceitos definidos, a sobrecarga de operador consiste na redefinição de seu significado para tipos diferentes daqueles com os quais ele tinha sido previamente definido. Por meio da sobrecarga de operadores, o programador pode redefinir o significado da maior parte dos operadores de C++, quando pelo menos um operando é um objeto de classe. Isto é, em geral, pode-se sobrecarregar os operadores de C++ definindo o que eles significam em relação a uma classe específica [2, 9, 10].

Torna-se importante salientar, tendo em vista que, para qualquer tipo "T", um "T" e um "T&" aceitam o mesmo conjunto de valores de inicializadores, funções com tipos de argumentos que diferem apenas por esse aspecto não podem ter o mesmo nome. Exemplo [2]:

```
int F (int i) { ... }
int F (int& r)        // erro: tipos de funções não são
{ ... }               // suficientemente diferentes
```

Antes da descrição de sobreposição de operadores, é útil rever alguns conceitos sobre operadores. O que é um operador é algo trivial. Por exemplo, o sinal de mais (+) é um operador que adiciona dois valores, e o sinal de menos (-) é um operador que subtrai dois valores. Estes e outros operadores semelhantes são denominados operadores binários, porque operam sobre dois argumentos. Alguns outros, tais como o operador de negação (!), são unários, pois eles requerem apenas um único argumento [4].

Um dos grandes poderes do C/C++, o seu pequeno tamanho, muito se deve à utilização da sobrecarga de operadores. Por exemplo, o operador "*" pode significar, dependendo do contexto, multiplicação, conteúdo de ponteiro ou declaração de ponteiro. Esta sobrecarga no operador "*" é inerente à linguagem C e não pode ser modificada pelo programador, ou seja, não se pode acrescentar novas definições ao símbolo "*".

Infelizmente, a linguagem C não permite também a utilização direta do operador de adição "+" para tipos criados pelo usuário. Assim, considerando estes tipos, deve-se executar uma declaração como:

```
s = soma (a, b);
```

Contudo, a forma mais clara de se realizar a operação seria:

```
s = a + b;
```

Em C++, é possível implementar esta última declaração. Para tal, deve-se sobrecarregar o operador de adição "+" [9].

Uma restrição que deve-se conhecer, é que não se pode criar operadores. O C++ permite apenas novas definições para os operadores já existentes. Além disso, nem todos os operadores podem ser sobrecarregados. A tabela a seguir mostra os operadores passíveis de sobrecarga em C++. Operadores unários são aplicáveis a somente um operando, enquanto operadores binários envolvem dois operandos. Em C++ não existem operadores sobrecarregáveis de ordem maior que dois.

Operadores Unários			
++	- -	~	!
Operadores Binários			
+	-	*	/
%	^	&	|
=	<	>	,
+=	-=	*=	/=
%=	^=	&=	|=
<<	>>	>>=	<<=
!=	<=	>=	==
||	&&	[]	( )
->	->*	new	delete
Operadores Unários ou Binários			
&	*	+	-

Além de não se poder criar operadores e só poder sobrecarregar aqueles que a linguagem permite, há mais restrições que devem ser conhecidas. Primeira, o operador sobrecarregado não pode alterar as regras de precedência e associatividade que o C e o C++ estabeleceram para a definição original. Segunda, ao menos um dos parâmetros da função operadora deverá ser objeto de classe, o que significa que não se pode redefinir as operações para os tipos da linguagem. Terceira, não se pode combinar sobrecargas para gerar uma terceira função operadora.

Existe duas formas de se definir sobrecarga de operadores em C++: declarando-a como função membro ou *friend* da classe que a utilizará. Há uma pequena diferença entre as sintaxes de declaração e definição da sobrecarga em cada forma. Inicialmente, serão observadas as sintaxes gerais e, posteriormente, serão discutidas estas diferenças. As sintaxes são as seguintes:

- Sobrecarga como função membro

  Declaração:

```
class Nome_Classe
{
        ...
        <tipo_retorno> operator op (<lista_de_parametros>);
};
```

  Definição:

```
<tipo_retorno> Nome_Classe :: operator op (<lista_de_parametros>)
{
        // definição da função membro
}
```

- Sobrecarga como função amiga

  Declaração:

```
class Nome_Classe
{
        ...
        friend <tipo_retorno> operator op (<lista_de_parametros>);
};
```

  Definição:

```
<tipo_retorno> operator op (<lista_de_parametros>)
{
        // definição da função friend
}
```

Observando-se as declarações de sobrecarga, conclui-se que elas são muito semelhante à sintaxe de declaração de qualquer função membro. A diferença está na inclusão da palavra-chave *operator* e no nome da função ("op") que, na sobrecarga, é o operador sobrecarregado. A palavra reservada *operator* identifica uma função operadora. <*tipo_retorno*> consiste em algum tipo conhecido da linguagem, porém, geralmente, é a própria classe para a qual o operador está sendo redefinido. Deve-se ter cuidado com o retorno do operador e lembrar-se de como este é utilizado originalmente. Por exemplo, se o operador de atribuição "=" for sobrecarregado para uma classe *String*, é possível que se queira continuar fazendo atribuições encadeadas, tais como:

```
String str1, str2, str3("Exemplo");
str1 = str2 = str3;
```

Se a sobrecarga declarar o retorno como *void*, será impossível fazer esta atribuição encadeada [9].

Torna-se importante salientar que o significado de operadores aplicados a tipos que não sejam classes não pode ser redefinido. A intenção é tornar C++ extensível, mas não mutável. Isso protege o programador de C++ dos abusos mais óbvios da sobrecarga de operadores, tais como redefinir + em inteiros para indicar subtração [2].

Os nomes das funções sobrepostas podem ser, por exemplo, "operador+", "operador-" e "operador*". Normalmente, não é permitido usar símbolos como +, - e * em identificadores, porém, para o caso especial de sobreposição de operadores, o C++ permite que o nome de função consista na palavra reservada *operator* e em um dos símbolos apresentados na tabela anterior.

Um exemplo prático de uma classe que utilize operadores sobrepostos ajudará a ilustrar os conceitos apresentados nesta seção. A listagem a seguir (strop1.cpp) mostra o início de uma classe que pode armazenar valores inteiros no formato de *strings*. Usando operadores sobrepostos, declarados como função *friend*, o programa pode avaliar expressões que adicionam *strings* sem precisar convertê-las para valores numéricos.

```
// strop1.cpp
# include <iostream>
# include <stdlib.h>
# include <string.h>
using namespace std;
class strop {
                    char value[12];
          public:
                    strop();
                    strop(const char *s);
                    friend long operator+ (strop a, strop b);
                    friend long operator- (strop a, strop b);
};
strop::strop() {
          value[0] = 0;
}
strop::strop(const char *s) {
          strncpy (value, s, 11);
          value[11] = 0;
}

void main () {
          strop a = "1234";
          strop b = "4321";
          cout << "\n a + b + 6 == " << (a + b + 6);
          cout << "\n a - b + 10 ==" << (a - b + 10);
}
long operator+ (strop a, strop b) {
          return ( atol(a.value) + atol(b.value) );
}
long operator- (strop a, strop b) {
          return ( atol(a.value) - atol(b.value) );
}
```

O programa "strop1.cpp" é apenas um exemplo simples de sobreposição de operadores e o programa demandaria um trabalho extenso para se tornar prático. Porém, ele ilustra diversas regras importantes. A classe "strop" armazena caracteres num pequeno vetor de tipo *char*, que é grande o suficiente para armazenar 11 dígitos e um delimitador NULL - espaço para o menor valor possível do tipo *long*, -2147483648, no formato de um *string*.

Os dois construtores fornecem os meios para inicializar novas instâncias do tipo "strop". O primeiro construtor cuida da definição de uma variável *default*. O segundo permite a criação de variáveis a partir de linhas como as definidas no início da *main*. As duas funções *friend* sobrepõem os operadores de adição e subtração para instâncias do tipo da classe e retornam um tipo *long*. Tipicamente, funções de operadores sobrepostos retornam o mesmo tipo que suas classes (ou uma referência a uma instância da classe). Mas isto não é obrigatório, funções de operadores sobrepostos podem retornar qualquer tipo de dado.

Em primeiro lugar o programa define duas instâncias "a" e "b" do tipo da classe, atribuindo a essas instâncias os *strings* "1234" e "4321", respectivamente. As instruções de fluxo de saída de dados usam estas instâncias nas expressões "a + b + 6" e "a - b + 10". Estas expressões somam e subtraem *strings* e valores inteiros, o que normalmente os operadores de mais e menos não podem fazer.

As funções de operadores sobrepostos foram declaradas como *friends* e não como funções membro. Por isso suas implementações são idênticas a de outras funções comuns do C++. As únicas diferenças são os nomes de função especiais "operator+" e "operator-", que permitem ao compilador avaliar expressões com os operadores de mais e de menos e instâncias da classe "strop".

Em "strop1.cpp", então, as duas funções de operadores sobrepostos, "operator+" e "operator-", são declaradas como amigas em comum. Um outro modo de alcançar o mesmo objetivo de sobrepor os operadores de mais e menos (e outros) é listar as funções como membro da classe. A próxima listagem, "strop2.cpp", é similar a anterior, porém mostra como sobrepor operadores como funções membro. Comparar os dois programas ajudará na decisão de qual dos dois métodos é apropriado para cada aplicação.

```
// strop2.cpp
# include <iostream>
# include <stdlib.h>
# include <string.h>
using namespace std;
class strop {
                    char value[12];
          public:
                    strop();
                    strop(const char *s);
                    long operator+ (strop b);
                    long operator- (strop b);
};
strop::strop() {
          value[0] = 0;
}
strop::strop(const char *s) {
          strncpy (value, s, 11);
          value[11] = 0;
}
long strop :: operator+ (strop b) {
          return ( atol(value) + atol(b.value) );
}
long strop :: operator- (strop b) {
          return ( atol(value) - atol(b.value) );
}
void main () {
          strop a = "1234";
          strop b = "4321";
          cout << "\n a + b + 6 == " << (a + b + 6);
          cout << "\n a - b + 10 ==" << (a - b + 10);
}
```

Os protótipos das funções de operadores sobrepostos da classe "strop" podem ser comparados com as funções *friend* equivalentes de "strop1.cpp". Como as funções em "strop2.cpp" são membros, elas já possuem acesso aos campos privados e protegidos da classe. Além disso, recebem um ponteiro *this* para a instância que chama as funções. Assim sendo, precisam apenas de parâmetros individuais. Para adicionar dois *strings* "values", as funções combinarão "this->value" com "b.value". Não há necessidade de passar dois parâmetros para as funções como existia em "strop1.cpp". De fato, fazê-lo é um erro, pois operadores binários tais como + e - requerem dois e apenas dois parâmetros. No "strop1.cpp" os dois parâmetros são necessários porque as funções *friend* não são membros e, portanto, não recebem ponteiros *this* para instâncias da classe.

Pode-se notar que a função *main()* de "strop2.cpp" é exatamente a mesma de "strop1.cpp". Não importando se as funções de operadores sobrepostos sejam declaradas como amigas ou como membros da classe, isso não afeta o modo desses operadores serem utilizados em expressões. Já as implementações de função sobrepostas em "strop2.cpp" diferem muito pouco daquelas encontradas em "strop1.cpp". Como as novas funções são membro, são rotuladas com o nome de classe "strop::". Além dessa mudança, as instruções agora fazem referências diretas ao campo "value" das instâncias para qual as funções foram chamadas. A referência ao segundo argumento, "b.value", é a mesma de antes.

Operadores unários, tais como o sinal de menos unário e !, operam sobre um argumento. Assim como operadores binários, pode-se declarar uma função de operador unário sobreposto como *friend* ou membro da classe. Para sobrepor um operador unário com uma função *friend*, deve-se listar apenas um parâmetro do tipo da classe. Por exemplo, pode-se acrescentar a seguinte declaração em "strop1.cpp" na declaração da classe, junto com as outras funções *friend*:

```
friend long operator- (strop a);
```

Ainda que a função *friend* declarada anteriormente na classe já faça uma sobreposição ao operador de subtração, como a nova declaração especifica apenas um parâmetro, não há conflito. Essa não é uma regra especial. É apenas uma conseqüência da sobreposição de funções que, como já citado anteriormente, permite que várias funções compartilhem o mesmo nome desde que as declarações de função difiram em pelo menos um parâmetro.

A implementação da função sobreposta retorna a negação do campo "value" do parâmetro "a". As linhas abaixo também podem ser acrescentadas em "strop1.cpp" para exemplificar o funcionamento de sobreposição de operadores unários.

```
main() {
        :
        cout << "\n -a ==" << -a;
}
long operator- (strop a) {
        return -atol (a.value);
}
```

Assim como operadores binários sobrepostos, pode-se também declarar operadores unários sobrepostos como funções membros. Para ilustrar pode-se colocar o seguinte comando na declaração da classe em "strop2.cpp":

```
long operator-(void);
```

Isto é equivalente à função unária *friend* acrescentada a "strop1.cpp" anteriormente. Mas, nesse caso, a função é declarada como membro da classe "strop". Como todas as funções membro recebem um ponteiro *this* para a instância para a qual as funções foram chamadas, a função "operator-" pode operar diretamente sobre o campo "value" de uma instância "strop". Por essa razão, a função não necessita de quaisquer parâmetros. Ela não deve ter nenhum parâmetro, ou o C++ não reconhecerá a função como sendo um operador unário sobreposto. Para implementar a função, as seguintes linhas devem ser acrescentadas em "strop2.cpp":

```
long strop :: operator- (void) {
        return -atol (value);
}
main() {    :
        cout << "\n -a ==" << -a;
}
```

Não importa se é declarada uma função de operador unário sobreposto como *friend* ou como membro da classe, pode-se usar essa função da mesma maneira. Como a classe define uma operação para o menos unário, o C++ pode avaliar a expressão "-a" para gerar a negação de um valor *long* representado como um *string* de caracteres [4].