UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
INSTITUTO DE INFORMÁTICA
CENTRO DE EMPREENDIMENTOS DE INFORMÁTICA

Curso de Introdução à Programação em C++

por

MARCELO DE OLIVEIRA JOHANN

monitoria

GLAUCO BORGES VALIM DOS SANTOS

Porto Alegre, agosto de 2004.

```
C++ e' como sexo na Adolescencia
=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=
```

1) Esta' na cabeca de todo mundo
2) Todo mundo fala sobre isto o tempo todo
3) Na realidade, quase ninguem esta' fazendo
4) Os poucos que estao fazendo:
    a) Fazem da maneira errada
    b) Desculpam falando que a proxima vez talvez sera' melhor
    c) Nao praticam com seguranca

Sumário

Resumo

Esta apostila é material de apoio a um curso de extensão ministrado pelo professor Marcelo de Olieira Johann na UFRGS, em agosto de 2004. O curso é eminentemente prático, com exemplos que demonstram os recursos e funcionalidades sendo oferecidos pelo professor, alterados em sala de aula, e com a proposição de exercícios para os alunos, a serem realizados durante a aula e como trabalhos extra-classe de fixação de conteúdo.

Devido à extensão e complexidade dos recursos oferecidos pela linguagem C++, o curso tem uma abordagem seletiva de conteúdos, e prioriza o desenvolvimento da habiliade de programação produtiva com um subconjunto adequadamente selecionado destes recursos. Outros recursos oferecidos pela linguagem, como sobrecarga de operadores, herança múltipla, programação de algoritmos em templates, são apresentados pela sua definição, objetivo, forma geral, mas não trabalhados em detalhe, permitindo que os alunos se concentrem em um modelo orientado a objetos e não se dispersem com a complexidade e detalhes de todas as opções da linguagem..

Palavras-chave: Programação, Linguagens, C++

1 Introdução

A linguagem C++ foi desenvolvida inicialmente por Bjarne Stroustrup na AT&T, de 1979 a 1983, à partir da linguagem C, tendo como idéia principal a de agregar o conceito de classes, de orientação à objetos, àquela linguagem. Razão porque iniciamente chamva-se de "C com classes". Bjarne procurou tanto quanto possível manter retrocompatibilidade com C, de modo que programas em C pudessem ser compilados por um compilador C++ com um mínimo de alterações. Entretanto, encarar C++ como um superconjunto de C é um erro, e C++ deve ser vista como uma "outra linguagem", por diversas razões. Em primeiro lugar, nem todo o programa escrito em C é compilado em C++ sem erros, e pode nem mesmo gerar o mesmo resultado, já que a sintaxe e a semântica de algumas construções diferem. Ao ligar-se partes de um programa em C++ com partes em C, estas devem ser bem especificadas, pois as convenções de funcionamento do código compilado também diferem. Além disso, C++ oferece um conjunto de mecanismos básicos que não estavam presentes em C, e estes devem ser usados para produzir *software* mais modular e confiável explorando-se as verificações disponíveis no compilador. Finalmente, os mecanismos de C++ devem inspirar a programação segundo o paradigma de orientação a objetos e, portanto, não se deve programar em C++ como se faz em C.

A partir da primeira versão de 1983, a linguagem foi sendo revisada e evoluindo, tornou-se disponível fora da AT&T em 1985, e após um longo processo foi padronizada pela ISO no final de 1997, pelo padrão ISO/IEC 14882. Você pode obter mais informações sobre o desenvolvimento da linguagem na página do próprio autor em [STR 2004].

STL é uma parte do padrão C++, e consiste em uma biblioteca de funções e estruturas de dados que todo compilador C++ deve oferecer, provê as implementações mais comuns em um programa, e pode-se utilizá-la com diferentes tipos de dados. Um bom ponto de partida para leitura específica sobre STL é na página da Silicon Graphics, Inc. em [STL 2004].

De forma geral, é possível obter muito material de referência da linguagem através de páginas na Internet. Tente por exemplo pesquisar com a expressão *"C++ reference"* no Google. Também é muito produtivo criar o hábito de pesquisar por problemas bem específicos. Tente, por exemplo, pesquisar *"iostream.h iostream difference"*, ou *"STL hash_map code example"*. Pode-se obter cursos [BRA 1998], livros e exemplos de código [ECK 2000]

Os compiladores do projeto GNU também podem ser obtidos livremente através da Internet. O ambiente BloodShed Dev-C++ roda sobre Windows e utiliza os compiladores gcc e g++. É possível baixá-lo de: http://www.bloodshed.net/devcpp.html. Há também inúmeros fóruns e listas de discussão sobre aspectos técnicos da linguagem. Então, use e abuse da Internet para aprendê-la e resolver seus problemas.

2 Recursos para Programação Estruturada

Este capítulo apresenta recursos básicos da linguagem C++ que servem para o modelo de programação conhecido como bloco-estruturado. São vistos os tipos de dados primitivos e as principais construções que servem para descrever operações e algoritmos, como controle de fluxo, juntamente com recursos diversos da linguagem. Em função da compatibilidade, muitos desses recursos são idênticos aos da liguagem C, e por isso essa parte é semelhante a uma breve "revisão" da linguagem C, seguida da apresentação dos mecanismos que são próprios de C++.

2.1 Compilando o primeiro programa: Hello World

O melhor meio de aprender uma linguagem é através de exemplos. Eis o primeiro programa, que dá sinal de vida apresentano uma mensagem. Identificamos nele os elementos: *#include*, função *main*, geração de saída de dados na tela, e como se colocam comentários.

chapter2/2.1-01-hello.cpp

A diretiva ***#include*** serve para fazer com que o compilador inclua como parte desse código outro arquivo de código fonte. Em geral, esse recurso é usado para incluir definições de dados e código que serão utilizados por nosso programa, mas já foram compilados e estão disponíveis em uma biblioteca. É função da etapa de ligação montar o executável final incluindo o código compilado para o nosso programa com o código compilado dessas bibliotecas.

Todo programa em C++, assim como em C, pode ser visto como um conjunto de funções. As funções serão apresentadas no capítulo 2, mas é necessário saber primeiro que todos os comandos da linguagem devem estar dentro de funções, e que, portanto, um programa deve ter no mínimo uma função. Essa função principal tem o nome de ***main***, sendo esse nome o que identifica por onde a execução inicia.

O ***namespace*** é um recurso utilizado para evitar que, quando se constróem grandes programas, nomes de variáveis, classes e funções conflitem. Esse conceito será apresentado mais adiante, mas aqui precisamos dele, pois a saída padrão, ***cout***, está definida dentro do espaço chamado ***std***, de *standard*. Dados podem ser impressos enviando-os para a saída adrão, *cout*. Da mesma forma, dados podem ser lidos através da entrada padrão, ***cin***. Também nesse caso o tipo de dado lido depende da variável para a qual se está fazendo a leitura.

chapter2/2.1-02-hello.cpp

Todas as declarações e comandos da linguagem devem ser terminados por '*;*'. Esse sinal não serve apenas como separador nas declarações, mas serve para identificar composição de seqüência entre os comandos, isto é, primeiro é executado um e depois o outro. Utilize vários comandos de impressão para gerar uma saída mais longa.

2.2 Declaração de variáveis e tipos de dados

Declarações de variáveis, ao contrário dos comandos, podem ser feitas tanto dentro quanto fora de funções. Se estão fora de uma função, elas são globais, e acessíveis a qualquer função do programa. As variáveis declaradas dentro de uma função são variáveis locais, e são acessíveis somente a essa função. As variáveis podem ser inicializadas, opcionalmente, na sua declaração, ou terem algum valor atribuído dentro de uma função, através da **atribuição**. Ao contrário de C, C++ não exige que todas as declarações de variáveis ocorram antes de todos os comandos em uma função. As declarações podem ser feitas em qualquer lugar.

chapter2/2.2-01-declaration.cpp

A tabela 1 apresenta os tipos de dados básicos da linguagem, exemplos de valores literais, e intervalos de valores aceitos. Cada tipo de dado possui representação de valores literais adequados, tanto para inicialização quanto para atribuições. Em geral usa-se *bool*, *char, int*, *long* e *double* mais freqüentemente, onde *long* é uma abreviação para *long int*.

bool	Valores booleanos	true false	true false
char	Caracteres simples	'a' 'A' '+' '\t' '\n'	0 a 255, isto é, 8 bits
int	Números inteiros	100 0x3F	16 bits ou mais, 32 normal
float	Números reais	1.2f .23f 1.f 1.5e-15f	
double	Reais dupla precisão	1.2 .23 1. 1.5e-15	

Há quatro modificadores que alteram a representação desses tipos básicos. Os modificadores ***signed*** e ***unsigned*** alteram o significado dos dados para representação de números negativos. Os modificadores ***short*** e ***long*** alteram a quantidade de bits com que o dado é representado. A linguagem não padroniza completamente o tamanho da representação binária de números, e esse é um aspecto que pode comprometer a portabilidade.

Assim como C, C++ é uma linguagem que permite programação em baixo nível. Assim, a linguagem não possui propriamente caracteres, mas o tipo *char* é apenas um núemro de no mínimo 8 bits, e os literais (*'a','X'*) podem ser atribuídos a qualquer variável numérica.

chapter2/2.2-02-declaration.cpp

Um tipo de dado pode ser convertido em outro tipo com um ***cast***. O *cast* deve preceder um valor e consiste no tipo para o qual se deseja converter, entre parênteses, como, por exemplo: *(int) 'a'*. Esse *cast* simples introduzido aqui é o mesmo da linguagem C, mas deve-se evitar utilizá-lo, pois em seu lugar C++ possui outros quatro tipos diferentes de *cast* que são mais seguros, a serem apresentados adiante.

chapter2/2.2-03-charcast.cpp

2.3 Operadores, expressões e precedência

Operadores são elementos fundamentais para efetuar computação. Eles atuam sobre um ou mais operandos, e produzem um resultado, que é o valor da expressão

formada por eles e seus operandos. Por exemplo *2 + 2* é uma expressão que utiliza o operador +, ordena que o computador efetue esse cálculo, e oferece como resultado o valor *4*. Assim, se você escreve *a = 3 * (2 + 2)*, a variável a deve receber o valor *12*. Elementar, por enquanto! Mas ocorre que C++ possui 42 operadores, com as mais variadas finalidades. Além disso, como em C, o conceito de expressão é extremamente abrangente, e utilizado em várias construções. Isso será mostrado passo por passo, mas iniciemos guardando bem esses conceitos: variável, literal, atribuição, operador, operando e expressão.

chapter2/2.3-01-operadores.cpp

2.4 Usando alguns vetores

Assim como declaramos uma variável de um tipo determinado, podemos declarar um vetor de qualquer tipo com a seguinte sintraxe: *tipo nome[tamanho]*. Por exemplo, o programa abaixo declara um vetor de inteiros, assinala valores a algumas posições, e as imprime. Vetores serão melhor estudados adiante, mas são aqui introduzidos para enriquecer os exemplos iniciais.

chapter2/2.4-01-vetor.cpp

2.5 Construções de controle de fluxo

A seguir são apresentadas as principais construções de controle de fluxo de execução. Elas servem para produzir seqüências de execução mais complexas que a seüência do texto do programa, por desvio ou repetição. Em uma linguagem bloco-estruturada, não se deve pensar em desvios para posições arbitrárias, mas em uma formação do programa como conjunto de repetições e execuções condicionais propriamente aninhadas.

2.5.1 A construção *if*

A construção *if* tem a seguinte sintaxe, sendo opcional a parte do *else* em diante:

if (*condição****)*** *comando_para_resultado_verdadeiro****; else*** *comando_para_falso****;***

Exercícios:

1- Escreva um programa que lê um valor da linha de comando e diz se ele é par ou ímpar.

2- Escreva um programa que recebe a nota final de um aluno e imprime seu conceito, conforme a tabela abaixo:

de 9 a 10	A
de 7,5 a 9	B
de 6 a 7,5	C
de 0 a 6	D

3- Escreva um programa que lê 3 valores na linha de comando e imprime a média, dizendo "Parabéns!!!" se a media for superior a um limiar pré-defido, ou "Que pena..." se for inferior.

2.5.2 Blocos e escopos

Se for necessário colocar mais de um comando aninhado dentro de uma construção condicional (ou de iteração) usa-se um bloco. Um bloco é uma construção que permite tratar um conjunto de comandos como se eles fossem um só. O bloco consiste nesse conjunto de comando entre chaves (´{´, ´}´). Um bloco define um escopo de variáveis. Isto significa que se uma nova variável é declarada dentro de um bloco com o mesmo nome de uma variável fora desse bloco, somente a variável interna é acessada desse ponto em diante até o final do bloco.

2.5.3 A construção *for*

A construção for serve para fazer repetições, e é bastante flexível ao contrário da maioria das linguagens. ela é composta por 4 elementos, identificados assim:

for (*expr1****;*** *expr2****;*** *expr3* ***)*** *comando****;***

A execução do for corresponde à seguinte seqüência:

expr1; expr2; {comando; expr3; expr2; } {comando; expr3; expr2;} ...

Em *expr1* e *expr3* pode-se escrever qualquer código de inicialização e de preparação para a próxima repetição, atuando sobre quaisquer variáveis. Essas expressões podem conter vários assinalamentos separados por vírgulas. A expressão *expr2* controla o laço, e também pode conter código arbitrário. Ela é executada antes da primeira repetição, de modo que o comando não é executado nenhuma vez se a expressão *expr2* é falsa inicialmente.

Exercícios:

1- Escreva um programa que lê um conjunto de 10 valoes inteiros e verifica se algum deles é negativo.

2- Escreva um programa que lê um conjunto de 10 números na linha de comando e imprime o número de pares e ímpares

2.5.4 As construções *while* e *do*

A construção *while (expr) comando;* é idêntica a um *for (;expr;) comando;*. Já a construção *do comando; while (expr)* difere das anteriores no sentido de que executa pelo menos uma vez o comando, mesmo quando a expressão *expr* é inicialmente falsa.

Exercícios:

1- Escreva um programa que fique indefinidamente lendo valores de entrada do usuário, dizendo se o valor é par ou ímpar, até que este digite um número especial de saída, como 99, por exemplo.

2- Escreva o programa que lê 3 valores na linha de comando e imprime a média de forma que, após realizar o primeiro cálculo de média, ele pergunte ao usuário se ele deseja fazer outro.

2.5.5 As declarações *break* e *continue*

A declaração *break* interrompe o comando de repetição mais interno dentro da qual ela se encontra. Serve também para separar casos da construção *switch*, vista adiante. Já a declaração *continue* interrompe apenas uma das repetições do laço, passando diretamente à execução da próxima repetição.

Exercícios:

1- Escreva um programa que lê dez valores e os imprime em ordem inversa após a leitura. Porém a leitura deve ser interrompida caso um valor negativo seja lido, imprimindo "VALOR INCORRETO".

2- Escreva um programa para dar o conceito de alunos em função da nota, conforme a mesma tabela de exercício anterior, mas que consulte três vetores de números inteiros, cada um com 10 posições, correspondendo aos 10 alunos. O primeiro vetor tem 0 se o aluno desistiu e cancelou o curso, e você não deve simplesmente ignorá-lo, ou 1 em caso contrário. O segundo vetor tem o número de aulas assistidas, e se for menor do que 3 (de 5 aulas) o aluno receberá conceito 'FF' (falta de freqüência). Finalmente o terceito vetor contém a nota do aluno, para classificação segundo os outros conceitos.

2.5.6 A construção *switch*

A construção *switch* serve para testar uma variável ou expressão contra diversos valores constantes. Casa valor deve aparecer separado em uma cláusula *case*. E cada cláusula *case* precisa ser interrompida por uma declaração *break*, caso contrário os comandos deste ponto em diante sontinuam sendo executados.

Exercícios:

1- Escreva um programa que leia valores entre 0 e 100 indefinidamente e calcule sempre a média atual. Deve ser impresso "Média ok", "Média insatisfatória" e "Média insuficiente" respectivamente, para valores nos intervalos [80-100], [60-79] e [0-59]. Caso o valor lido não seja um número entre 0 e 100, abortar o programa com uma mensagem de erro

2.5.7 O operador *?*

O par de operadores ? : funciona como a construção condicional if, mas sob a forma de expressão. Isto é, esse operador retorna um valor, o valor da expressão que foi avaliada. Um exemplo de uma expressão com esse operador, já usada em um assinalamento, é a seguinte:

int a = (b>10) ? 10 : b; // a recebe 10 se b maior que 10 ou b caso contrário

2.5.8 C++ tem *label* e *goto*, mas não contem para ninguém

Assim como C, C++ também possui o comando goto, que desvia a execução do programa para um *label* que identifica uma outra linha de código. Deve-se evitar ao máximo o uso desse recurso, pois ele quebra a estrutura do programa em termos de iterações, condições e blocos. Entretanto, ele pode ser bem utilizado para geração automática de código, como, por exemplo, na implementação de autômatos.

2.6 Recursos próprios de C++

A seguir apresentamos alguns recursos e construções que são próprias de C++, não aparecendo em C, ou cuja implementação é completamente diferente.

2.6.1 Namespaces

Namespaces servem para separar espaços de nomes em módulos e bibliotecas inteiras, evitando que nomes de variáveis, estruturas, funções e classes conflitem. Os *namespaces* são abertos, isto é, podem ser declarados várias vezes, cada qual acrescentando novos elementos dentro do mesmo *namespace*.

chapter2/2.6-01-namespace.cpp

2.6.2 Strings em C++ e em C

Em C uma *string* é apenas um vetor de caracteres terminado por *'\0'* (valor binário zero). Sendo um vetor, ele é caracterizado pelo seu endereço inicial. Esse endereço pode ser constante, quando foi declarado como vetor ou literal, ou variável, quando declarado como ponteiro. Nesse último caso, ele deve ser inicializado para apontar para uma área de memória com a *string*, seja ela constante, de outro vetor, ou alocada dinamicamente. Ocasionalmente será necessário usar esses vetores de caracateres em C++, mas a linguagem C++ tem objetos do tipo *string* com muitas operações convenientes e fáceis de usar. Atribuição, cópia e concatenação, que somente podem ser feitas por funções em C, são feitas apenas com = e + em C++, por exemplo.

chapter2/2.6-02-strings.cpp

2.6.3 Entrada e saída em C++ e em C

Os exemplos a seguir mostram como se faz entrada e saída de dados em C e C++. O primeiro apresenta exemplos de entrada e saída de dados formatados por console, enquanto o segundo programa apresenta leitura de arquivos em C e C++.

chapter2/2.6-03-io.cpp

chapter2/2.6-04-files.cpp

2.6.4 Tratamento de exceções

Para que serve tratamento de exceções? Em programas pequenos, seu uso pode não ser justificado, pois tudo que eles oferecem pode ser obtido com um simples teste (*if*) após a realização de uma operação. Mas em programas grandes, o local onde uma situação de erro, ou exceção ao funcionamento desejado, é identificada pode não ter

uma relação direta com o local onde essa situação deve ser tratada. Por exemplo, para uma função que procura por um valor em um vetor, não encontrá-lo é uma exceção. Mas quem a chamou pode estar simplesmente consultando o vetor justamente para saber se o elemento está lá, e nesse caso a resposta negativa é uma situação normal. Ao contrário, se a chamada é parte de uma função que colocou esse valor e precisa dele para continuar, o fato de não estar lá pode ser um erro gravíssimo, e necessitar um relato ao usuário e interrupção do programa. Essas diferenças de tratamento podem ocorrer entre funções que foram chamadas em vários níveis, inclusive com diferentes funções tendo diferentes interpretações, ações e mensagens para a mesma situação. É por isso que um mecanismo simples de testes e codificação de retorno se torna inapropriado para a tarefa.

O mecanismo de tratamento de exceções da linguagem serve justamente para isso. Ele especifica o registro de uma situação de exceção, e o desvio para um local que declara tratá-lo diretamente, cruzando as fronteiras de funções e controle de fluxo (mas fazendo as operações necessárias de desativação dessas funções, como um retorno prematuro). No disparo da exceção, é possível passar quaisquer informações ao tratamento. Aqui é utilizado um número inteiro apenas, mas será tipicamente um objeto de uma classe “exceção” em uma aplicação real.

chapter2/2.6-05-try.cpp

3 Estruturas de Dados, Funções e Ponteiros

Neste capítulo exploramos estruturas de dados básicas que fazem parte da sintaxe da linguagem, como vetores, estruturas, uniões, a organização básica em funções e passagem de parâmetros, e finalmente o conceito de endereços, ponteiros, referênciação e de-referenciação, vendo como esses conceitos relacionam-se entre si. O uso das estruturas de dados básicas, indispensável em outras linguagens, dará lugar ao uso de classes e templates da STL, introduzidas posteriormente, mas completam até aqui um conjunto de recursos completo para programação estruturada.

3.1 Vetores

Já foi vista a declaração e uso de vetores simples. Se um vetor tem inicialização, pode-se omitir seu tamanho entre os colchetes, e ele será calculado pleo número de elementos da inicialização. Se o tamanho do vetor é declarado, pode-se inicializá-lo com menos valores do que seu tamanho, e os restantes receberão 0, mas não se pode inicializá-lo com mais valores do que seu tamanho. Não é possível assinalar ou copiar vetores inteiros. Outras características de vetores são melhor compreendidas adiante com funções e ponteiros.

chapter3/3.1-00-matrix.cpp

chapter3/3.1-01-matrix1.cpp

3.2 Funções

Uma função agrupa um conjunto de comandos da linguagem, podendo esse conjunto ser chamado de várias partes de um programa. Funções servem para economizar código-fonte e organizar melhor um programa. Uma função possui um nome, um tipo de valor de retorno, e um conjunto, possivelmente vazio, de parâmetros de entrada, cada um com seu tipo específico. Esses elementos caracterizam uma assinatura da função, que, para ser completa, deve ser seguida de sua implementação, ou corpo. Um **protótipo** de uma função é uma definição com o corpo omitido, e se utiliza esse recurso para declarar previamente uma função e fazer seu nome conhecido antes de dar sua implementação.

Ao contrário de C, em C++ pode-se ter várias funções com exatamente o mesmo nome, desde que os tipos ou número de parâmetros de entrada variem. Chama-se isso de **sobrecarga** de funções. Embora se possa utilizar esse recurso arbitrariamente, ele faz sentido para dar versões diferentes de uma mesma função que estão disponíveis para diferentes parâmetros de entrada. As funções em C++ podem ter parâmetros com **valores *default***. No caso de argumentos para esses parâmetros serem omitidos na chamada os valores default são utilizados. Somente pode ser omitido o valor de um parâmetro se todos os outros à sua esquerda também tiverem sido omitidos.

O qualificador ***inline***, utilizado antes da declaração de uma função, faz com que o código dessa seja substituído no lugar da chamada, se possível, para que o programa

rode mais rapidamente. Em C isso deve ser implementado com a substituição textual das macros (*#define*).

Como exercício, experimente escrever um programa que imprime os números de *1* a *10* através de um laço, mas sem usar as construções *for* ou *while*, e sim com chamada de função recursiva. Este exercício demonstra que pode-se utilizar o paradigma de programação funcional em uma linguagem como C++, onde todo programa é expresso como um conjunto de funções aplicadas umas às outras. Programação funcional é mais do que isso, mas a semelhança existe.

3.3 Estruturas

O programador deve modelar o problema que deseja resolver ou processar utilizando-se de algoritmos e estruturas de dados. Do ponto de vista de recursos de linguagem, alguns recursos são oferecidos para facilitar a descrição dessas estruturas. De fato, um deles possui exatamente esse nome, de estrutura, mas é nada mais do que um agregado de valores.

3.3.1 struct

Em C++, uma estrutura é um tipo de dado conhecido pela linguagem. Assim, ela entende a todas as antigas formas de declaração de C, e adicionalmente o nome da estrutura pode ser diretamente utilizado sem o prefixo *struct*.

chapter3/3.3-01-struct.cpp

3.3.2 union

Uma união é apenas uma estrutura onde todos os campos utilizam a mesma posição. Ela pode ser utilizada para dar diferentes interpretações a uma área de memória, ou armazenar de forma compacta dados mutuamente excludentes.

3.3.3 enum

Uma enumeração é um novo tipo de dado que possui valores discretos específicos. Pode-se declarar uma enumeração para o tipo dia, e uma variável do tipo dessa enumeração poderá ter os valores: segunda, terca, e assim por diante. Uma enumeração é nada mais do que um número inteiro para o qual são dados nomes específicos a alguns valores. De fato, pode-se definir exatamente quais são esses valores. As enumerações servem para organização interna do código, mas quando seu valor é impresso, é interpretado apenas como número.

3.4 typedef

A declaração *typedef* define um novo tipo de dado tendo como base um tipo conhecido. Por exemplo, pode-se fazer:

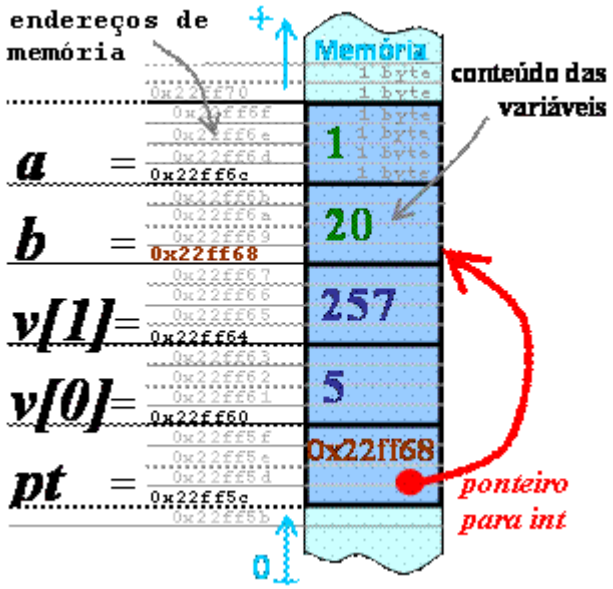
```
typedef int int32;

typedef short int16;
```

E a partir de então passar a usar somente *int16* e *int32* nas declarações de um programa. Se esse programa for portado para outra máquina e compilador onde *int* tem só 16 bits, pode-se redefinir *int32* como *long* apenas em uma posição.

3.5 Ponteiros

Todas as variáveis estão armazenadas em memória. Cada uma tem seu endereço e seu conteúdo. Quando usamos o nome de uma variável num programa, o compilador compila no código o endereço, para que, quando executado, o processador acesse o conteúdo. Isso significa que o compilador só vê o endereço e o programador só vê o conteúdo. Para ver o endereço, usa-se o operador *&* na frente da variável, que significa "*endereço de*", e é chamado operador de **referenciação**. Ao contrário, para manipular um valor armazenado em um endereço, usa-se o operador * na frente do endereço, que significa "*valor apontado por*", e é chamado operador de **de-referenciação**. Um ponteiro é apenas uma variável cujo conteúdo é um endereço de memória, provavelmente o endereço de outra variável. Esse endereço pode ser obtido por referenciação, e posteriormente o dado daquela variável apontada pode ser manipulado por de-referenciação. Para declarar um ponteiro, acrescenta-se * entre o tipo e o nome da variável (nesse caso variável apontador). Convém já notar que uma variável do tipo ponteiro, ou apontador, como todas as outras, também tem seu conteúdo (que vai ser o endereço de outra) e seu próprio endereço em memória. É por isso que aparecem os ponteiros para ponteiros.

```
#include <iostream>
using namespace std;
int main()
  {
  int a = 1;
  int b = 20;
  int v[2] = {5,257};
  int *pt = &b;
  cout << &a << " ";
  cout << a << endl;
  cout << &b << " ";
  cout << b << endl;
  cout << &v[1] << " ";
  cout << v[1] << endl;
  cout << &v[0] << " ";
  cout << v[0] << endl;
  cout << &pt << " ";
  cout << pt << endl;
  }
```



chapter3/3.5-01-pointer.cpp

chapter3/3.5-02-pointer.cpp

Uma boa sugestão para aprendizagem é declarar outras variáveis e vetores e imprimir seus valores e conteúdos. Utilize os programas abaixo para fazer essas experiências, incluindo fazer atribuições às variáveis que são ponteiros e às variáveis apontadas por elas. Sempre considere que as variáveis locais aparecem em ordem inversa na memória, pois são criadas na pilha do sistema, e que existem regras de

alinhamento, e, portanto, variáveis declaradas consecutivamente podem aparecer separadas por espaços para satisfazer tais regras.

3.6 Constantes

A palavra *const* é um qualificador, assim como *signed* ou *short*, e serve em primeiro lugar para dar nome a valores constantes posteriormente usados no programa. Já que a variável não pode ser posteriormente atribuída, toda declaração de const deve ter inicialização.

Mas o significado de *const* é um pouco mais sutil. Ele define que determinado valor **não vai ser alterado**, e não precisa necessariamente ser aplicado a um valor de fato constante. Pode-se passar uma variável para uma função que recebe uma constante, significando que essa função não vai alterar esse valor. Ao passar argumentos por valor, isso nem faz muita diferença, mas no caso de ponteiros sim. Se uma função recebe *const char**, pode-se passar qualquer ponteiro para caracteres, mesmo não constante. Nesse caso, *const* serve para especificar (e enforçar) que a função não vai e nem pode alterar o caracter apontado pelo ponteiro que ela recebeu.

Em uma declaração, o qualificador *const* pode ser usado em diversas posições, com diferentes significados. Pode-se ler da esquerda para direita (em inglês) para compreender as interpretações. A construção **const* é uma construção especial que significa ponteiro constante.

*char *const cp;*	// ponteiro constante para um caracter
*char const * p;*	// ponteiro para um caracter constante
*const char * p2;*	// outro ponteiro para um caracter constante
*const char *const cp2;*	// ponteiro constante para um caracter constante

chapter3/3.6-01-const.cpp

3.7 Os quatro tipos de cast

Deve-se evitar o uso do cast em estilo C: (tipo). C++ introduz quatro novos tipos de cast com sintaxe e funcionalidade própria. Assim, é mais fácil identificar cada um desses tipos de conversão no código e corrigi-las se necessário. Os quatro tipos de cast são:

static_cast<T> e Conversão de um valor entre tipos de dados conhecidos pela linguagem, produzindo a representação correta do valor de *e* no novo tipo *T*. Também é utilizada para conversão de tipos de ponteiros;

reinterpret_cast<T> e Informa ao compilador para passar o valor de *e* para um destino do tipo *T* simplesmente pela cópia da representação binária. É utilizado para conversão entre tipos não relacionados, para passar informações em baixo nível de um formato para outro, em implementações dependentes de máquina e menos portáveis;

const_cast<T> e Elimina o qualificador *const* de uma expressão *e*, produzindo um valor do tipo *T* em lugar de *const T*;

dynamic_cast<T> e É uma conversão de ponteiros inteligente, onde os tipos de dado de origem e destino são identificados em tempo de execução para verificar se um ponteiro do tipo T pode ser utilizado para apontar para e;

3.8 Referências

Uma referência é um nome alternativo para uma variável. Ela pode ser usada em passagem de parêmetros e retorno de resultados, para fazer com que se opere na variável original, implementando passagem por referência sem usar ponteiros. Nesse sentido ela faz algo bem semelhante a um ponteiro. Entretanto, uma referência não é um ponteiro, não ocupa espaço em memória, e não se podem efetuar operações sobre ela (somente sobre a variável que ela referencia).

chapter3/3.8-01-reference1.cpp

chapter3/3.8-02-reference2.cpp

chapter3/3.8-03-reference3.cpp

3.9 Ponteiros para estruturas

Um uso muito comum de ponteiros é para que apontem para estruturas. Uma estrutura pode ter um campo que aponta para outra estrutura ou para outra instância da mesma estrutura. Para acessar o valor de um campo de uma estrutura tendo um ponteiro para ela, usa-se: "*(*pt).campo*". A sintaxe *pt->campo* substitui essa construção, simplificando a navegação através de ponteiros, principalmente porque esse *campo* pode ser outro ponteiro. Lembre-se, entretanto, que é muito comum haver ponteiros nulos e estes não devem ser acessados, de forma que qualquer navegação deve ser precedida de teste para saber se o novo ponteiro não é nulo.

chapter3/3.9-01-arrow.cpp

3.10 Alocação de Memória

Para alocar memória adicional, ou de tamanho variável, em C, usa-se a função *calloc*, e para retornar a memória ao sistema usa-se a função *free*. Em C++, essa funções não serão usadas, e em seu lugar há dois operadores, *new* e *delete*, que fazem as funções respectivas, mas compreendem os tipos de dados e operações de criação envolvidas. Os operadores *new* e *delete* funcionam apenas para os tipos escalares. Para alocar e retornar vetores usa-se os operadores *new[]* e *delete[]*.

chapter3/3.10-01-new.cpp

4 Recursos para Programação Orientada a Objetos

As estruturas simples da linguagem C são substituídas pelas classes de C++. Uma classe é uma estrutura que tem, além de dados, funções membro, chamadas métodos. As classes agregam outros conceitos de orientação a objetos, como controles de acesso e polimorfismo. Nesse capítulo são vistos os principais recursos que C++ oferece para a programação O-O. Procura-se caracterizar a variedade de recursos da linguagem, incluindo herança múltipla e sobrecarga de operadores, mas é dada maior ênfase nos mecanismos "puros" mais fundamentais, deixando recursos mais intrincados para estudos futuros.

4.1 Orientação a Objetos

Orientação a Objetos é uma forma de programação. A idéia é agrupar os dados ao seu processamento. Ou seja, em vez de haver dados declarados e funções que os alteram arbitrariamente, cada pequeno conjunto de dados estará acompanhado por um conjunto de funções que operam sobre eles fazendo todas as operações desejáveis. Nenhum outro trecho de código do programa acessará esses dados diretamente, mas sempre chamando as funções corretas. Dessa forma, é mais fácil manter consistência dos dados, encontrar e corrigir erros, fazer manutenção, implementar novas funcionalidades ou trocar representações de dados. Então, quando se programa módulos em uma linguagem como C pela definição de estruturas e conjunto completo de funções que sobre elas operam, pode-se já estar usando uma boa dose de programações orientada a objetos.

4.1.1 Teoria

Na teoria do paradigma de orientação a objetos (O-O), o programa é formado por um conjunto de **classes** que são modelos para criação de **objetos**. As classes e objetos possuem **membros** que são dados privados, e **métodos** de acesso, que são as funções. Quando um trecho de código quer fazer uma operação sobre um objeto ele emite uma **mensagem** para esse objeto requisitando a operação. Na prática, pode-se implementar isso como uma simples chamada de função (chamada de método), que é o que ocorre em C++. A principal característica de O-O é o **encapsulamento**, que é justamente o fato de os dados não estarem acessíveis diretamente, mas apenas através dos métodos permitidos. Para completar o conceito de O-O, o mecanismo de **herança** faz a definição de classes mais especializadas a partir de classes básicas. Nesse caso, se pode reimplementar métodos de uma classe básica na classe mais especializada e posteriormente tratar diversos objetos diferentemente especializados através dos mesmos métodos, o que chama-se de **polimorfismo**. A especificação de classes cuja única função é definir conjuntos de métodos dá origem ao conceito de **classes abstratas** e as diferencia dos **tipos concretos** que são aqueles que realmente implementam os métodos.

4.2 Controle de Acesso

O controle de acesso é um recurso que proteje os dados, apenas em tempo de compilação, para enforçar a idéia de encapsulamento. Uma boa regra é sempre declarar todas as variáveis em uma classe como privadas, e oferecer métodos para inspecionar e alteras aquelas que devem ser permitidas. É natural que muitas classes

então tenham dados privados, um conjunto de funções *get* para retornar cada dado, e um conjunto de funções *set* para setar cada dado. Sempre faça assim, mesmo que paraça inútil à primeira vista. Desta forma é garantido que: 1- somente os dados para os quais você oferece métodos serão acessados diretamente; 2- todo acesso é feito através desses métodos, e portanto é possível deixá-los consistentes um a um e entre eles; 3- qualquer alteração de formato, representação, origem nos dados poderá ser feita em apenas um lugar.

chapter4/4.1-00-acesso.cpp

chapter4/4.1-01-firstclass.cpp

chapter4/4.1-02-secondclass.cpp

4.3 Construtores e destrutores

Cada classe pode implementar funções construtoras e destrutoras de objetos, que têm, respectivamente o nome da classe e o nome da classe precedido por '~'. As funções construtoras servem para fazer inicialização do objeto, cuja memória já está alocada, e para criação de sub-objetos, vetores ou quaisquer outras estruturas que este necessite para estar completo. Se na inicialização de um objeto foram criados outros objetos por alocação (com os operadores *new* e *new[]*), então na sua destruição estes devem ser apropriadamente retornados para o sistema (com os operadores *delete* e *delete[]*). Esta é a finalidade das funções destrutoras.

chapter4/4.3-01-construtor.cpp

chapter4/4.3-02-conversao.cpp

chapter4/4.3-03-destrutor.cpp

chapter4/4.3-04-destrutor2.cpp

A existência das funções destrutoras traz um problema à tona. Objetos podem ser copiados e assinalados livremente. Mas quando um objeto contém um ponteiro para algo que ele criou (outro objeto, vetor, etc...) esse ponteiro será copiado pelo seu valor binário, e não o conteúdo de um sub-objeto para o outro. No exemplo acima, se dois cursos são criados e você assinala um ao outro, não somente os dois ficam com o ponteirointerno apontando para a mesma lista de alunos, como uma das listas de alunos fica solta sem ninguém a referenciando na memória. Quando os objetos sairem do escopo (neste caso, ou forem deletados com *delete*), as funções destrutoras irão ser chamadas e irão deletar duas vezes a mesma lista de alunos, causando mal funcionamento do programa. Para resolver isso, implementa-se uma função especial para fazer o assinalamento. Isso é possível através da sobrecarga de operadores. Define-se uma nova função para o operador de assinalamento '=', que deverá atuar entre um objeto dessa classe à sua esquerda e um outro objeto da mesma classe à sua direita. A função impplementada faz a cópia correta dos valores de objetos apontados pelo nosso objeto principal sendo assinalado, e é a primeira aplicação prática e muito útil de sobrecarga de operadores.

chapter4/4.3-05-assignment.cpp

4.4 Sobrecarga de operadores

Sobrecarregar o operador de assinalamento como no exemplo anterior foi uma forma muito elegante de resolver um problema comum. Em C++, dos 42 operadores existentes, 38 podem ser sobrecarregados. Mas deve-se evitar fazer coisas muito complexas. Aqui são apresentados mais dois exemplos de sobrecarga simples e que produz construções mais elegantes.

chapter4/4.4-01-elegantoverload.cpp

chapter4/4.4-02-safevector.cpp

chapter4/4.4-03-safevector2.cpp

4.5 Membros estáticos e constantes

Membros estáticos são aqueles que pertencem não a uma instância em particular mas ao escopo global. Em uma classe, se pode ter variáveis e métodos estáticos. Variáveis estáticas são conhecidas como variáveis de classe, pois só haverá uma instância delas, e não uma por objeto, como nas variáveis de objeto. Na linguagem C++, além de serem declaradas dentro da classe, elas devem ser declaradas e inicializadas fora de todas as classes, representando que são como variáveis globais.

Já para métodos a diferença não está na existência de múltiplas instâncias, mas apenas na dependência dos objetos. Métodos estáticos são como funções globais quaisquer, e não necessitam de um objeto da classe para serem chamados.

chapter4/4.5-01-static.cpp

As variáveis e métodos também podem ser cconstantes ou não. Variáveis constantes são aquelas que não serão alteradas. Elas precisam ser inicializadas na sua declaração,já que não podem sofrer atribuições. Já os métodos constantes, que se caracterizam pela palavra *const* logo após a lista de parâmetros entre parênteses, são aqueles que apenas fazem inspeção do objeto, mas não alteram seu valor. Esses são os únicos métodos que podem acessar objetos declarados como *const* no código, pois garantem que não alterarão seu valor. O correto uso do qualificador *const* confere maior clareza ao código.

chapter4/4.5-04-constfunction.cpp

4.6 Herança de classes

A herança de classes permite especializar classes genéricas. A classe mais genérica é chamda de classe base e a classe que herda suas características é camada de classe derivada. A herança pode ser pública ou privada. Quando é privada, *default*, as características públicas da classe base se tornam privadas na classe derivada.

4.6.1 Classes base e derivada

Um ponteiro de uma classe base pode apontar para umobjeto de classe dela derivada, mas não vice-versa. Métodos podem ser redefinidos em classes derivadas,

mas o método chamado depende do ponteiro através do qual é chamado.

chapter4/4.6-00-derived0.cpp

chapter4/4.6-01-derived1.cpp

4.6.2 Membros *protected*

Pode-se verificar que quando uma classe deriva de uma outra classe base, mesmo sendo ela um tipo especializado e sendo também do tipo da classe base, ela não tem acesso aos membros provados da classe base. Para permitir esse acesso, a classe base deve declarar membros *protected*. Membros *protected* são privados desta classe na visão de todo o programa, mas acessíveis a classes derivadas.

chapter4/4.6-06-derived6.cpp

4.6.3 Construtores

O construtor de uma classe derivada pode (e deve) chamar construtores de classes base das quais deriva, passando os parâmetros corretos, para que todas os objetos sejam construidos adequadamente.

chapter4/4.6-xx-sample.cpp

4.6.4 Funções virtuais

Embora um ponteiro para uma classe base possa apontar para um objeto de uma classe derivada, ele não pode chamar os métodos da classe derivada, mais especializada, pois é um ponteiro genérico. Para isso existem as funções virtuais. Se uma função é qualificada como virtual, quando as classes derivadas reimplementam essa mesma função, é a versão mais especializada delas que será chamada, mesmo que o ponteiro seja somente para a classe base.

chapter4/4.6-03-derived3.cpp

4.6.5 Funções virtuais puras e classes abstratas

Uma função virtual pura é aquela para a qual nenhuma implementação é dada na classe base. Ela deve ser obrigatoriamente implementada pelas classes derivadas. Isso significa que a classe base com funções virtuais não pode ser utilzada para criar objetos, pois é uma especificação incompleta, a que chamamos de tipo abstrato, diferindo dos tipos concretos, que são so completos.

chapter4/4.6-04-derived4.cpp

chapter4/4.6-05-derived5.cpp

4.6.6 Herança múltipla

A linguagem C++ permite herança múltipla. Herança múltipla permite flexibilidade mais traz problemas adicionais, como múltiplas ocorrências de algum membro comum às classes base. Assim, não é uma opção muito interessante em arquitetura de *software*. É preferível especificar interfaces abstratas, que são classes

com funções virtuais que apenas especificam que tipos de operações podem ser executadas sobre um determinado tipo de objeto, do que realmente implementar herança de múltiplos tipos concretos.

5 Modelagem e Estilos de Codificação

Com os recursos vistos até aqui, que já foram selecionados, é possível escrever programas com diversos tipos de organização e estilo de codificação, mesmo porque é em geral possível tanto modelar os dados quanto implementar algoritmos de diferentes formas em qualquer liguagem. Este capítulo dá algumas referências e exemplos sobre modelagem O-O e estilos de codificação, mostrando vantagens e desvantagens de cada um.

5.1 Regras e recomendações

Cada organização procura disciplinar o desenvolvimento de código fonte de alguma forma, para que seu desenolvimento seja seguro e a manutenção mais fácil. É usual existirem regras para a codificação e formatação de código. Aqui apenas apresentamos algumas pequenas regras gerais que você deve seguir, na opinião do autor, mas muitas outras são necessárias para um estilo de programação mais perfeito.

5.1.1 Algumas regras de programação

- sempre inicialize as variáveis na sua declaração;
- sempre teste a validade dos parâmetros recebidos em uma função: se forem números, seus limites, e se for ponteiro, para que não seja nulo;
- declare todas as variáveis de objeto como privadas, sem exceções;
- jamais utilize números no código: declare constantes que digam seu significado e sejam inicializadas com seu valor;
- sempre desconfie quando você escrever três vezes o mesmo código: deve haver um meio genérico de fazer o mesmo com repetições, vetores, indireção ou templates;
- não use variáveis globais, variáveis estáticas, *goto*, *defines*;
- prefira construções simples e claras à aglomerados de expressões que fazem o mesmo processamento;
- use o qualificador const sempre que possível;
- evite o uso de ponteiros: use referências e as estruturas de STL, guardando objetos em vetores, listas e principalmente maps, onde eles podem ser localizados, recuperados e alterados através do uso de identificadores e iteradores;

5.1.2 Algumas regras básicas de formatação

- sempre use edentação correta, preferencialmente com as chaves alinhadas;
- comente no inicio do arquivo o que ele faz, seu autor, data de criação, e ao menos data da última atualização e qual foi: se possível, mantenha a lista das atualizações, datas e quem as executou;
- use comentários antes das implementações das funções dizendo o que elas fazem, a estratégia, o algorimo;
- use comentários separando partes do código e anunciando o que será feito;
- use comentários no fim da linha para explicar detalhes daquela linha;
- declare apenas uma variável por linha;
- use nomes de variáveis que expliquem seu uso;

- faça tabulações na declaração de variáveis (em classes e métodos) de forma que fiquem alinhados verticalmente: tipos de retorno, nomes, parâmetros, comentários;

5.2 Programação em módulos

Programas grandes devem ser divididos em módulos, e estarem implementados em arquivos diferentes, para melhor organização e compilação por partes.

O primeiro exempo é um programa completo em C, onde podem-se identificar diversos tipos de declarações que formam o programa: *includes*, *defines*, estruturas, protótipos de função, variáveis globais função principal e implementação das funções utilitárias. Todas as definições quenão geram código devem estar em um arquivo de cabeçalho, para que outros módulos possam conhecer suas interfaces. O programa principal também deve estar separado de todos os módulo, para que esses possam ser aproveitados. O programa foi então dividido nos três arquivos que seguem.

chapter5/5.2-00-program0.c

chapter5/5.2-00-module.h

chapter5/5.2-00-module.c

chapter5/5.2-00-main1.c

De forma análoga, um programa em C++ também é composto por um conjunto de definições e depois as suas implementações. As definições devem ir para um arquivo de cabeçalho (.h) e as implementações para arquivo do módulo (.cpp).

chapter5/5.2-01-program1.cpp

chapter5/5.2-01-module1.h

chapter5/5.2-01-module1.cpp

chapter5/5.2-01-main1.cpp

5.3 Listas encadeadas por herança e com ponteiros

O exemplo abaixo mostra como evitar a reimplementação de listas construindo classes base genéricas para lista e depois especializando-as para os tipos de dados que se deseja armazenar. Mas também é possível fazer uma lista genérica sem herança, com nodos que contenham ponteiros *void*. Esses ponteiros podem apontar para qualquer tipo de objeto, e então é possível armazenar quaisquer objetos dentro dela, desde que as conversões de ponteiros sejam indicadas com *cast*. Sugere-se implementar uma lista desse tipo como exercício.

chapter5/5.3-01-listoptions.cpp

chapter5/5.3-02-listoptions2.cpp

5.4 Relações do tipo “é um” e “tem um”

As duas alternativas vistas para implementação da lista encadeada evocam a questão de quando se deve usar herança ou quando se devem usar apontadores (ou mesmo cópias) para outros objetos. Em alguns casos tanto uma como outra alternativapoderão funcionar, apresentando vantagens e desvantagens. Mas existe uma regra básica que é a seguinte. Quando há dois elementos (classes) relacionados sendo especificados, pergunta-se se a relação dentre os dois é do tipo **"é um"** ou **"tem um"**. Por exemplo, um funcionário **tem um** emprego, e **não é um** emprego. Ao contrário, um aluno **é uma** pessoa, e **não tem** uma pessoa. Na maioria dos casos, essa resposta é bem clara. Em alguns outros, será possível identificar as alternativas. Por exemplo, um botão é um objeto gráfico, mas ele também pode ter uma funcionalidade própria e um objeto gráfico associado.

6 Templates e STL

Um template é uma evolução das macros de C, mas que inclui todos os recursos sofisticados e verificações de alto nível da linguagem C++. É uma maneira de especificar estruturas e algoritmos configuráveis. STL é uma biblioteca padronizada contendo um conjunto de templates que implementa as estruturas de dados e os algoritmos mais utilizados na construção de um programa. Templates devem ser usados para prover maior produtividade, evitando que se tenha que re-inventar a roda a cada novo programa, ou que se dependa de bibliotecas produzidas por terceiros e os problemas associados de portabilidade. Por essa razão, aqui será tratado apenas o conceito e o praticado o uso de templates da STL, evitando-se toda complexidade associada à definição de novos templates de classes e métodos. Mesmo com a exclusão desse conteúdo, apresenta-se brevemente no final o conceito de programação genérica, cujo potencial é bastante atraente.

chapter6/6.1-01-hello.cpp

chapter6/map.cpp

chapter6/hash_map.cpp

chapter6/generic.cpp

Anexo 1 Operadores e palavras reservadas

É essencial para a programação conhecer os operadores da linguagem C++. Listas de operadores, descrições e exemplos podem ser encontrados em muitas fontes, como em "http://www.cplusplus.com/doc/tutorial/tut4-2.html" [CPP 2004], por exemplo. A seguir está uma lista de todos os operadores da linguagem , com sua associatividade e finalidade, retirada de "http://www.cs.stmarys.ca/~porter/csc/ref/cpp_operators.html" [SCO 2004]. A tabela inicia com os operadores de mais alta precedência, e cada nível de menos precedência está separado pelas linhas pontilhadas. A precedência define qual operação será efetuada primeiro quando aparece ao lado de outras com operadores de mesma precedência. Já a associatividade diz qual operação será edetuada primeiro quando os operadores têm mesmo nível de precedência. A associatividade pode ser à esquerda (LR) ou à direita (RL).

```
Operators          Assoc    Description
-------------------------------------------------------------------
-
(expression)              parentheses used for grouping
::                   RL   (unary) global scope resolution operator
::                   LR   class scope resolution operator
-------------------------------------------------------------------
-
()                   LR   parentheses used for a function call
()                   LR   value construction, as in type(expression)
.                    LR   member selection via struct or class object
->                   LR   member selection via pointer
[]                   LR   array element access
const_cast           LR   specialized type cast
dynamic_cast         LR   specialized type cast
reinterpret_cast     LR   specialized type cast
static_cast          LR   specialized type cast
typeid               LR   type identification
++   --              LR   postfix versions of increment/decrement
-------------------------------------------------------------------
-
All the operators in this section are unary (one argument) operators.
++   --              RL   prefix  versions of increment/decrement
+ -                  RL   unary versions
!                    RL   logical NOT
~                    RL   bitwise complement ("ones complement")
&                    RL   address of
*                    RL   dereference
new                  RL   allocates memory to dynamic object
delete               RL   de-allocates memory allocated to dynamic
object
new []               RL   allocates memory to dynamic array
delete []            RL   de-allocates memory allocated to dynamic
array
sizeof               RL   for computing storage size of data
(type)               RL   cast (C-style type conversion)
-------------------------------------------------------------------
-
.*                   LR   struct/union/object pointer (member
dereference)
->*                  LR   pointer to struct/union/object pointer
```

```
                          (indirect member dereference)
-------------------------------------------------------------------
-
*    /    %          LR  multiplication and division
-------------------------------------------------------------------
-
+    -               LR  addition and subtraction
-------------------------------------------------------------------
-
>>   <<              LR  input and output stream operators
-------------------------------------------------------------------
-
<    <=   >   >=     LR  inequality relational
-------------------------------------------------------------------
-
==   !=              LR  equality relational
-------------------------------------------------------------------
-
&                    LR  bitwise AND
-------------------------------------------------------------------
-
^                    LR  bitwise XOR
-------------------------------------------------------------------
-
|                    LR  bitwise OR
-------------------------------------------------------------------
-
&&                   LR  logical AND
-------------------------------------------------------------------
-
||                   LR  logical OR
-------------------------------------------------------------------
-
?:                   RL  conditional
-------------------------------------------------------------------
-
=                    RL  assignment
*=                   RL  multiplication and assignment
/=                   RL  division and assignment
%=                   RL  modulus (remainder) and assignment
+=                   RL  addition and assignment
-=                   RL  subtraction and  assignment
-------------------------------------------------------------------
-
throw                LR  throw exception
-------------------------------------------------------------------
-
,                    LP  the operator, not the separator
                         (combines two expressions into one)
-------------------------------------------------------------------
-
```

As palavras reservadas da linguagem C++ são as da tabela abaixo. Elas não podem ser utilizadas como nomes de identificadores, e isso pode conflitar com programas escritos em C e que sejam portados para C++.

asm auto	do double	if inline	public register	this throw

break	else	int	return	try
case	enum	long	short	typedef
catch	explicit	mutable	signed	union
char	extern	namespace	sizeof	unsigned
class	float	new	static	using
const	for	operator	struct	virtual
continue	friend	private	switch	void
default	goto	protected	template	volatile
delete				while

Bibliografia

[STR 97] Stroustrup, Bjarne. **The C++ Programming Language – Third edition**. Addison-Wesley, 1997.

[STR 2004] Stroustrup, Bjarne. **The C++ Programming Language**. 2004. Disponível em: "http://www.research.att.com/~bs/C++.html"

[HAM 98] Scott Hamilton. **The father of C++ explains why Standard C++ isn't just an object-oriented language**. New York, IEEE, 1998. Disponível em: http://www.research.att.com/~bs/ieee_interview.html.

[STE 91] Al Stevens. **Aprenda você mesmo C++**. Rio de Janeiro, LTC, 1991.

[SGI 2004] SGI. **Standard Template Library Programmer's Guide**. 2004. Disponível em: http://www.sgi.com/tech/stl/

[BRA 1998] Eric Brasseur . **C++ tutorial for C users**. 1998. Disponível em: http://www.4p8.com/eric.brasseur/cppcen.html#l24

[CPP 2004] Cplusplus Recources. Overloading operators. Disponível em : "http://www.cplusplus.com/doc/tutorial/tut4-2.html".

[SCO 2004] Porter Scobey. **C++ Operators, with Precedence and Associativity** 2004 Disponível em : "http://www.cs.stmarys.ca/~porter/csc/ref/cpp_operators.html".

[ECK 2000] Bruce Eckel. **Thinking in C++ 2nd Edition.** 2004 Disponível em : "http://mindview.net/Books/TICPP/ThinkingInCPP2e.html". (Free Electronic Book)