

Sumário

I	Básico	6
1	Sintaxe básica	7
1.1	O que é sintaxe	7
1.2	Os blocos de código e o ponto-e-vírgula	7
1.3	A sensibilidade de caixa e os caracteres ASCII	7
1.4	Comentários	8
1.5	Um pequeno exemplo de sintaxe	8
2	Tipos de dados	10
2.1	Valores lógicos	10
2.1.1	Booleanos	10
2.2	Números	10
2.2.1	Números inteiros	10
2.2.2	Números flutuante	10
2.3	Caracteres	11
2.3.1	Símbolos ASCII	11
2.4	Vazios	11
2.5	Modificadores de faixa	11
2.5.1	Localização da faixa	11
2.5.2	Comprimento da faixa	12
2.6	Literais	12
2.6.1	Inteiro	12
2.6.2	Flutuante	12
2.6.3	Caracteres	12
2.6.4	Faixa	12
3	Variáveis	13
3.1	Declarando variáveis	13
3.2	Inicializando variáveis	13
3.3	Exemplos de variáveis	14
3.4	Escopo	14
4	Operadores	15
4.1	Unários	15
4.1.1	Incremento e decremento unitário	15
4.1.2	Sinalizadores aritméticos	16
4.1.3	Negador lógico	16
4.1.4	Complemento binário	17
4.2	Binários	17
4.2.1	Atribuição simples	18
4.2.2	Aritméticos	18
4.2.3	Deslocadores bit-a-bit	19
4.2.4	Lógicos bit-a-bit	20
4.2.5	Atribuição composta	21
4.2.6	Comparadores	21

4.2.7	Lógicos booleanos	22
4.3	Ternário	23
4.4	Precedência de operadores	23
5	Controladores de Fluxo	25
5.1	Decisões na direção do fluxo	25
5.1.1	O <i>if</i> e a estrutura básica de decisão	25
5.1.2	O <i>else</i> e a estrutura complementar	27
5.1.3	Estruturas aninhadas	28
5.1.4	O <i>switch</i> e a estrutura composta	30
5.2	Repetições no fluxo	32
5.2.1	O <i>while</i> e a estrutura de repetição indefinida	33
5.2.2	O <i>do</i> e a estrutura de início obrigatório	35
5.2.3	O <i>for</i> e a estrutura de repetição definida	36
5.2.4	Mudanças bruscas em repetições	38
II	Intermediário	39
6	Procedimentos e funções	40
6.1	Procedimentos	40
6.2	Passagem de argumentos	41
6.3	Funções	42
6.4	Polimorfismo	43
6.5	Recursão	44
7	Mais operadores	45
7.1	Modelador	45
7.2	Indexador	46
7.3	Tamanho em memória	46
7.4	Acesso a membro	46
7.5	Endereço	47
7.6	Acesso indireto	47
7.7	Acesso indireto de membro	47
7.8	*Alocador de memória	47
7.9	*Dealocador de memória	48
8	Sequências de memórias	49
8.1	*Vetores	49
8.2	*Matrizes	50
8.3	Sequências de caracteres	51
9	Tipos abstratos	53
9.1	*Enumeradores	53
9.2	*Estruturas	53
9.3	*Unões	53
10	Ponteiros	54
10.1	Declarando ponteiros	54
10.2	Inicializando ponteiros	54
10.3	Ponteiros de vetores	55
10.4	Ponteiros de ponteiros	55
10.5	Aritimética de ponteiros	55
10.6	Exemplos de ponteiros	55

III	Avançado	58
11	Memória Dinâmica	59
12	Classes e Objetos	60
13	Mais sobre procedimentos e funções	61
A	Tabela ASCII	67
B	Ordem de precedência	68

Introdução

VERSÃO PRELIMINAR

Breve histórico

O C++ é ...

O mínimo código em C++ é apresentado no código 1.

```
1 int main()  
2 {  
3     return 0;  
4 }
```

Código 1: Código mínimo

Este é o código mais simples que um compilador C++ aceitará sem erros ou notas de atenção.

Parte I

Básico

VERSÃO PRELIMINAR

Capítulo 1

Sintaxe básica

1.1 O que é sintaxe

Sintaxe é o “componente do sistema linguístico que determina as relações formais que interligam os constituintes da sentença, atribuindo-lhe uma estrutura”[1]. Ou seja, é a maneira que as palavras de uma determinada linguagem se combinam formando sentenças.

No contexto de programação, sintaxe é o conjunto de regras estruturais que determinam o formato da linguagens. Como qualquer outra linguagem, o C++ tem um conjunto de regras a ser seguido para que possa ser entendido pelo compilador.

1.2 Os blocos de código e o ponto-e-vírgula

Qualquer linha do código que realize uma ação delimitada e determinada é denominada *comando* ou *instrução*. Todo comando precisa de um caractere especial para ser dividido dos demais. Na criação da linguagem foi escolhido o ponto-e-vírgula (;), provavelmente pelo lado poético de terminar uma oração delimitada sem finalizar a frase, o que é coerente, tendo em vista que o programa é uma série de comandos bem delimitadas que montam uma instrução mais complexa. É importante lembrar sempre: todo comando deve apresentar um ponto-e-vírgula (;) em sequência.

Em qualquer lugar é possível criar regiões de código com algumas propriedades especiais, estas regiões são denominadas *blocos de código*. Sua principal função é agrupar comandos, de tal forma que juntos formem uma instrução mais complexa. Pode-se dizer que os comandos são como as instruções passo-a-passo para a confecção de um bolo, e o bloco de código seja a própria expressão *faça um bolo*. Um bloco de código é delimitado pelos caracteres de chave esquerda ({) para abrir o bloco e direita (}) para fechá-lo.

Um bloco de código pode ser criado dentro de outro. É recomendado que, a cada novo bloco, um aumento no recuo da margem do programa seja feito, mantendo todo comando dentro do bloco alinhado, permitindo a fácil visualização do início e fim do bloco de código. Normalmente utiliza-se um caractere de tabulação, e este é o nome do processo de formatação visual do código. Também é possível encontrar programadores que utilizem espaços em branco para o processo de tabulação, porém não é recomendado. /

{;}

1.3 A sensibilidade de caixa e os caracteres ASCII

A *sensibilidade de caixa* é uma das características mais comuns em linguagens de programação modernas, embora possa parecer incoerente em linguagem coloquial, quando escrita pode ser a diferença entre um programa que é compilável ou não, essa sensibilidade é denominada *case sensitive*. De maneira simples, a caixa alta é o conjunto de caracteres de letras maiúsculas e a caixa baixa o conjunto de caracteres de letras minúsculas. Então, se em um lugar do código existir um termo como **palavra**, em outro o termo **Palavra** e em um terceiro lugar o termo **PALAVRA**, o compilador reconhecerá como três símbolos diferentes, isto é, completamente distintos e sem qualquer relação, mesmo que em um contexto coloquial sejam exatamente o mesmo termo.

Por se tratar de uma linguagem relativamente antiga, o C++ tem uma limitação nos caracteres processáveis, então alguns caracteres especiais não são aceitos pelo compilador, gerando um erro. Ainda que a sintaxe esteja perfeita e nenhum erro de lógica exista, caracteres como ã ou ç não são tolerados. Somente os caracteres da tabela ASCII são aceitos, esta pode ser consultada no apêndice ???. Por isso ainda, é comum que programadores escolham escrever seus programas usando termos em linguagens que não apresentem esse caracteres, como inglês, grego ou até latim. Não é incomum encontrar termos como *alpha* e *beta*, ou talvez *fungi* e *monera* como estes termos.

A≠a

1.4 Comentários

Ao longo do desenvolvimento de programas, é comum que o programador dê nome a entidades no programa, nomes que, em programas maiores, podem se confundir. Para evitar isso, o programador pode criar uma tabela externa, como um arquivo de texto ou uma folha num caderno, mas isso se torna cansativo e não progride caso o desenvolvedor não tome um tempo especial para isso. Para garantir que todos os nomes de entidades no programa sejam utilizados da forma correta, uma simples anotação no lugar onde são criados seria suficiente.

Todas as partes de processamento também ficam mais simples de entender quando há uma anotação a seu respeito explicando seu funcionamento. Tais anotações servem para explicar o programa e são consideradas uma boa prática de programação. Pode parecer algo desnecessário para programas pequenos, mas durante a rotina de desenvolvimento de software, não é raro passar um longo período de tempo sem editar alguns segmentos do programa e, depois de tanto tempo, é comum se esquecer o que esta parte é faz e como ela o faz. Um simples comentário pode ser o suficiente para que não sejam perdidas horas de análise de código afim de simplesmente lembrar qual o objetivo deste segmento.

Em C++, existem duas formas de criar comentários, a mais comum utilizando duas barras (//), depois desta sequência, o restante da linha é considerada comentário.

//Linear

A segunda forma, muito utilizada para manter cópias de códigos alternativos no arquivo de código (também chamado de arquivo fonte ou código fonte), é semelhante a declaração de blocos, com um delimitador de direita e outro de esquerda, para abertura e fechamento respectivo. O início do bloco é delimitado com o par barra-asterisco (/*) e o fim por asterisco-barra (*/).

/*Blocular*/

Os comentários devem respeitar o uso de caracteres ASCII, caracteres especiais continuam proibidos, porém qualquer outra coisa pode ser escrita sem maiores problemas.

1.5 Um pequeno exemplo de sintaxe

O código 2 apresenta alguns exemplos de sintaxe. Note a forma que a palavra *int* não está com a coloração diferenciada como no código 1.

```
1 | Int main()    //Esta linha não será aceita por causa da letra maiúscula
2 | {            //O bloco principal inicia aqui
3 |     return 0; //Este comando tem ; no final
4 | }            //O bloco principal acaba aqui
5 |
6 | /*
7 |  Aqui podemos escrever qualquer coisa
8 |  Normalmente este comentário aparece no começo do programa
9 |  Contém informações como data, nome do programa, do programador, etc
10 | int main()
11 | {
12 |     return 0;
```


13

}

14

*/

Código 2: Exemplos de sintaxe no código mínimo

VERSÃO PRELIMINAR

Capítulo 2

Tipos de dados

Uma das primeiras necessidades dentro de programação e de construção de software no geral, é o armazenamento de dados. Valores numéricos, condições lógicas, caracteres e até mesmo frases podem ser armazenadas dentro dos tipos do C++. Um tipo é dito primitiva quando é definido na base da linguagem e é apenas um molde para um armazenamento de dados.

2.1 Valores lógicos

2.1.1 Booleanos

O tipo `bool` armazena valores de estado lógico, ou seja, `verdadeiro` ou `falso`. Seu nome deriva do termo *boolean*.

Em C++, um `bool` pode receber os estados lógicos utilizando os valores numéricos 1 ou 0, respectivamente para verdadeiro ou falso, ou as palavras-chave `true` ou `false`. Um detalhe herdado da linguagem C é que qualquer valor diferente de 0 será considerado verdadeiro, então um número como 54 será considerado verdadeiro.

O espaço de memória ocupado pelo tipo `bool` depende do compilador utilizado. Normalmente ocupa um byte, porém alguns compiladores otimizados podem fazer ocupar apenas um bit.

2.2 Números

2.2.1 Números inteiros

O tipo `int` é o utilizado para números inteiros de uma maneira geral. Seu nome deriva do termo *integer*.

Pode receber números diretamente em sua forma decimal, como 28, ou na forma hexadecimal, como 1C.

O espaço ocupado depende da arquitetura do sistema, em geral ocupa 2 bytes em arquiteturas 32 bits, e 8 bytes em arquitetura 64 bits. Os números registráveis também dependem da arquitetura.

2.2.2 Números flutuante

Existem dois tipos para armazenamento de números flutuantes, eles apenas diferem em precisão. O número flutuante é o equivalente computacional ao número real na matemática, porém apenas um número finito de valores podem ser representados de maneira exata, decorrente ao limite da máquina.

O primeiro é o tipo `float`, seu nome deriva do termo *floating*, que significa flutuante, a escolha do termo é decorrente ao sistema de codificação de número flutuante, onde o ponto decimal troca de lugar, como a diferença entre 3.14 e 31.4. O segundo é o tipo `double`, que tem o dobro da precisão da anterior.

A precisão dos números flutuantes é relativa à quantidade de bits reservado para armazenar cada parte do dado. Basicamente um número flutuante pode ser representado pela equação 2.1.

$$\pm M \cdot B^{\pm e} \quad (2.1)$$

Onde M é a mantissa e representa o número em sua forma fracionária reduzido em sua própria base até que seja menor que 1, por exemplo, 3.1415 será dividido por 10 até que tome o formato 0.31415. B representa a base

numérica escolhida. E e o expoente, que equivale a quantidade de divisões pela necessárias para que o número tome o formato correto. É equivalente à notação científica, então $3.1415 = +0.31415 \cdot 10^{+10}$.

Estes valores são salvos na memória de forma binária, logo a base $B = 2$. O primeiro bit está reservado para o sinal da mantissa, logo em seguida seu valor reduzido, então o sinal e o valor do expoente. A precisão do número flutuante é associada a estes valores.

Um `float` tem precisão de 38 casas decimais e ocupa 4 bytes. Um `double` tem precisão de 308 casas decimais e ocupa 8 bytes. Pode ser comum achar que o maior é melhor, porém ele torna as operações mais lentas.

Podem receber valores diretamente na forma decimal caso os valores sejam inteiros, como 28, com marcação de ponto decimal, como 3.141592, ou em notação científica, como $1.6e-19$, onde $e-19$ é o mesmo que 10^{-19} .

Vale ressaltar que o sinal de demarcação decimal utilizado no C++ é o ponto (`.`), a vírgula é utilizada para outras coisas na linguagem.

• \neq ,

2.3 Caracteres

2.3.1 Símbolos ASCII

O tipo `char` é o padrão para armazenamento de informações de texto de apenas um caractere definido na tabela ASCII. Seu nome deriva do termo *character*.

Em C++, um `char` pode receber qualquer caractere ASCII de três maneiras:

- Através do símbolo diretamente, utilizando-o entre aspas simples (`'`), por exemplo `'M'`.
- Através do código hexadecimal do símbolo, por exemplo, `0x4D` para `'M'`.
- Através do código decimal do símbolo, convertido através do hexadecimal, por exemplo, `77` para `'M'`.

Os itens estão em ordem de usabilidade, o mais comum e mais prático para o programador é utilizar os símbolos do teclado, sem consultar tabelas. O uso da tabela é necessário quando símbolos não presentes no teclado são necessários ao longo do programa.

O espaço ocupado pelo tipo `char` é de um byte. Como a codificação ASCII é baseada em valores numéricos para codificação dos símbolos, o tipo `char` também é considerado armazenamento de número inteiro.

Uma regra de sintaxe importante está na diferença entre aspas simples (`'`) e aspas duplas (`"`), onde a primeira é utilizada para notação de caracteres únicos, já a segunda para sequências de caracteres.

`'a' ≠ "a"`

2.4 Vazios

O tipo `void` é diferente dos demais, ele não armazena valores, portanto não ocupa espaço definido em memória. Basicamente, ele é utilizado para dar nome a algo de valor vazio, que é o significado de seu nome. A utilização dos tipos `void` será exemplificada no capítulo 6.

2.5 Modificadores de faixa

Todo tipo primitivo tem uma faixa de atuação padrão, porém esta faixa pode ser alterada com o uso de certas palavras-chave. A tabela 2.1 apresenta os valores de intervalos dos tipos primitivos puros e os com modificações de faixa, repare que alguns tipos não sofrem alteração.

2.5.1 Localização da faixa

As palavras-chave `signed` e `unsigned` definem, respectivamente, se a declaração de um é tipo com sinal e sem sinal, tipos sem sinal são sempre maiores ou iguais a zero. Estas palavras-chave não alteram o espaço ocupado pelo tipo em memória, porém mudam a faixa de valores registráveis.

Apenas os tipos de armazenamento de inteiros podem receber estes modificadores.

2.5.2 Comprimento da faixa

As palavras-chave `short` e `long` definem, respectivamente, se a declaração de um tipo é encurtada e estendida. Estas palavras-chave alteram o espaço ocupado pelo tipo de memória estendendo a faixa de valores registráveis, porém não talveram o sinal base do tipo.

Nem todas os tipos podem receber este modificador.

É muito comum encontrar estes modificadores definidos de forma que o tipo fique implícito, então, ao invés de utilizar `long int`, utiliza-se apenas `long`. Esta forma também se aplica a `short` e o tipo implícito é sempre `int`.

Tabela 2.1: Relação de faixa e tamanhos de memória para tipos primitivos com modificadores de faixa

código	tamanho (B)	valor mínimo	valor máximo
<code>bool</code>	1	0	1
<code>signed char</code>	1	-127	126
<code>char</code>	1	-127	126
<code>unsigned char</code>	1	0	255
<code>signed short int</code>	2	-32768	32767
<code>short int</code>	2	-32768	32767
<code>unsigned short int</code>	2	0	64535
<code>signed int</code>	4	-2147483648	2147483647
<code>int</code>	4	-2147483648	2147483647
<code>unsigned int</code>	4	0	4294967295
<code>signed long int</code>	8	-9223372036854775808	9223372036854775807
<code>long int</code>	8	-9223372036854775808	9223372036854775807
<code>unsigned long int</code>	8	0	18446744073709551616
<code>float</code>	4	$1.2 \cdot 10^{-38}$	$3.4 \cdot 10^{+38}$
<code>double</code>	8	$1.73 \cdot 10^{-308}$	$1.7 \cdot 10^{+308}$
<code>long double</code>	16	$3.4 \cdot 10^{-4932}$	$3.4 \cdot 10^{+4932}$

2.6 Literais

Literais são os valores digitados de maneira direta no código, sem o uso de variáveis. O tipo do literal é definido com caracteres específicos junto a eles, que deixam explícito o tipo de um dado, deixar claro para o compilador como ele deve ser processado. Este tipo de definição é importante, por exemplo, para operações matemáticas, já que a precisão de um `float` é diferente da de um `double`, o que pode levar à erros de processamento.

2.6.1 Inteiro

Se um número é definido sem qualquer adicional, é considerado `int`, como em 29. Também é possível escrever o número em hexadecimal, como já foi apresentado, utilizando o prefixo `0x` ou `0X`, como em `0x1D`, que também pode ser apresentado como `0X1D`, `0x1d` ou `0X1d`. Ainda é possível escrever o número em forma binária, utilizando-se do prefixo `0b` ou `0B`, como em `0x00011101`.

2.6.2 Flutuante

Quando um literal flutuante é definido, será considerado `double` pelo compilador, a menos que conte com o sufixo `f` ou `F` que o tornará `float`, como em `0.114f`.

2.6.3 Caracteres

Os caracteres, quando utilizando a tabela ASCII, são considerados números. Quando utilizamos as aspas simples ('), o caracter é convertido para o número que o representa. Por exemplo, `'M'` é convertido para 77.

2.6.4 Faixa

É possível definir faixa em literal do tipo `int`, utilizando o sufixo `L` ou `l`, como em `199930L`.

Capítulo 3

Variáveis

Retomando a ideia apresentada no início do capítulo 2, é necessário agora criar entidades que tenham as características dos tipos primitivos e possam ser trabalhadas, lembrando que tipos primitivos são definidos pela linguagem e são apenas moldes de armazenamento de dados.

3.1 Declarando variáveis

As variáveis são os armazenadores de dados, moldados a partir dos seus tipos primitivos, essencialmente a declaração de qualquer tipo de variável é feita da mesma maneira, apresentada no código 3. Respectivamente a lista de modificadores, separados por espaço, o nome do tipo e finalmente o nome da variável.

O nome de uma variável pode conter caracteres nos intervalos $[0,9]$, $[a,z]$, $[A,Z]$ e o underline (`_`). O nome também não pode ter como primeiro caracter um número. Vale lembrar que apenas o uso de caracteres ASCII é válido.

```
1 //...
2 <modificadores> <tipo> <nome>;
3 //...
```

Código 3: Declaração de variável

É recomendada a preferência de nomes de variáveis com letras minúsculas, pois letras maiúsculas costumam ser utilizadas em outras situações que serão descritas posteriormente.

Também recomenda-se utilizar um nome autodescritivo para uma variável, por exemplo, em um calendário, é esperável encontrar uma variável chamada `dia` e não uma chama `abacaxi`. Existem ainda os casos onde apenas uma palavra não será suficiente para tal descrição, ou queremos fazer separações dentro do nome, normalmente utiliza capitalização na primeira letra de cada palavra após a primeira, por exemplo `diaDaSemana`. É menos comum, ainda que permitido, o uso de underlines (`_`) para a separação, por exemplo `dia_do_mes`.

Outra recomendação é nunca utilizar o underline (`_`) como primeiro caracter, pois alguns compiladores tem palavras-chave próprias que tem esta característica em comum, e podem gerar erros inesperados.

É importante se atentar ao fato de que uma variável só pode ser utilizada depois de ter sido declarada, ou seja, não é possível utilizar uma variável e declará-la duas linhas abaixo. Alguns exemplos podem ser encontrados na seção 3.3.

3.2 Inicializando variáveis

A declaração do código 3 não inicializa a variável, ou seja, não passa um valor inicial. Quando uma variável não é inicializada, pode trazer o chamada *lixo de memória*.

O lixo de memória é o resíduo de outros processos do sistema operacional vigente. Quando o processo acaba, é menos custoso deixar a memória com os ultimos valores registrados, e isso pode ser um problema para o próximo programa a utilizar aquele espaço de memória, por isso é recomendado declarar toda variável com um valor de inicialização.

Existem algumas formas de inicializar variáveis que dependem de operadores aritméticos, estas formas são apresentadas no capítulo 4, onde os operadores são descritos. A maneira apresentada aqui não é a mais usual, mas é mais veloz, a execução de um comparador de velocidades pode comprovar.

Além da sequência de declaração, ao fim adiciona-se o valor escolhido entre parênteses (`()`), conforme o código 4. O valor escolhido deve ser pensado de acordo com o objetivo da variável, por exemplo, se ela for um contador, é interessante que comece em um, se for um acumulador de soma, um bom valor inicial é zero.

```
1 //...
2 <modificadores> <tipo> <nome>(<valor>);
3 //...
```

Código 4: Declaração de variável

3.3 Exemplos de variáveis

Alguns exemplos de declaração e inicialização de variáveis são apresentados no código 5, com comentários referentes às declarações.

```
1 //...
2 bool falso(0);           //Com número
3 bool verdadeiro(true);   //Com palavra-chave
4 char igual(0x3D);        //Sinal de igual ASCII
5 char letraA('A');        //Aspas simples
6
7 int contador(1), acumulador(0); //Várias variáveis do mesmo tipo
8 unsigned int positivo(523);    //Inteiro sem sinal
9 short doisBytes(93);           //Modificador de comprimento
10 long grande(32416189349L);     //Número grande
11 double cargaFundamental(1.6e-19); //Notação científica
12 float pi(3.14159265358979323846264338327950288419f); //Flutuante preciso
13 //...
```

Código 5: Declarações de variável

3.4 Escopo

As variáveis podem ser declaradas em quase qualquer lugar do programa, porém não podem ser acessadas de qualquer lugar do programa. Existem dois casos de declaração de variáveis, as locais e as globais:

- Uma variável *local* é declarada dentro de um bloco de código, e não pode ser acessada fora dele. A tentativa de acesso fora da região onde a variável foi declarada gera um erro de escopo.
- Uma variável *global* é declarada fora de qualquer bloco de código, e pode ser acessada de qualquer parte do programa, inclusive dentro de outros blocos.

Existe ainda uma terceira situação, onde uma variável é localmente global, ou seja, pode ser acessada em qualquer região dentro do bloco no qual foi declarado, até em blocos internos, mas não pode ser acessada fora desta região ao qual pertence. O código 6 mostra um exemplo.

```
1 //...
2 int A;           //Somente A pode ser acessado aqui
3 {
4     int B;       //Aqui A será global e B local, C e D são inexistentes
5     {
6         int C;   //Aqui A e B são globais, C local e D inexistente
7     }
8     {
9         int D;   //Aqui A e B são globais, D local e C inexistente
10    }
11 }
12 //...
```

Código 6: Declarações de variável

Capítulo 4

Operadores

Operadores são as entidades capazes de alterar as variáveis, utilizando-as em contas, comparações, etc. Toda utilização de uma variável será por meio de algum operador, alterando seu dado em consulta ou em memória. A alteração em consulta implica no valor armazenado em memória não sofrer alteração, como se uma cópia fosse criada e apenas esta sofresse a alteração em memória. A alteração em memória é a que muda o valor armazenado pela variável, sem possibilidade de recuperação. O operador retorna o valor final da operação. Por exemplo, na soma de dois números de um mesmo tipo o operador retornará o valor do tipo destes números, caso sejam de tipos distintos, retornará um valor do tipo que ocupe mais bytes ou que seja mais preciso (dando preferência ao segundo) dentre os números.

Um operador pode realizar ações sobre uma, duas ou até três variáveis, sendo chamado respectivamente de unário, binário e ternário.

Existem mais operadores além dos descritos aqui, porém seu uso requer domínio de outras técnicas, portanto serão apresentados no capítulo 7.

4.1 Unários

De uma maneira geral, todo operador unário tem uma sintaxe semelhante, apresentada no código 7, e um exemplo genérico no código 8.

```
1 //...
2 <operador> <nome>; //Onde <nome> refere-se ao identificador da variável
3 //...
```

Código 7: Sintaxe geral de operadores unários

```
1 //...
2 <tipo> <nome> (<valor>); //Operadores normalmente atuam sobre variáveis
3 int A(<operador> <nome>); //Operadores serão utilizados sempre num contexto
4 //...
5 //adequado
```

Código 8: Exemplo genérico de operadores unários

4.1.1 Incremento e decremento unitário

Uma variável pode ter seu valor incrementado ou decrementado, isto é, acrescido ou decrescido, respectivamente, em um. O operador responsável pelo incremento é o duplo mais (++). O operador responsável pelo decremento é o duplo menos (--). A sintaxe associada é apresentada no código 9.

```
1 //...
2 <nome>++; //Incremento posfixo
3 ++<nome>; //Incremento prefixo
4
5 <nome>--; //Decremento posfixo
```

```

6  --<nome>; //Decremento prefixo
7  //...

```

Código 9: Sintaxe de incrementadores e decrementadores unitários

Seu uso pode ser prefixo ou posfixo, ambos os casos incrementam, mas de maneiras diferentes. O prefixo realiza a operação e então retorna o valor. O posfixo retorna o valor e então realiza a operação. Um exemplo é encontrado no código 10.

```

1  //...
2  int A(5); //A vale 5
3  int B(A++); //A vale 6, B vale 5
4  int C(++B); //A vale 6, B vale 6, C vale 6
5  C++; //Também pode ser usado independente do valor de retorno
6
7  int D(5); //D vale 5
8  int E(--D); //D vale 4, E vale 4
9  int F(E--); //D vale 4, E vale 3, F vale 4
10 //...

```

Código 10: Exemplo de incrementadores e decrementadores unitários

4.1.2 Sinalizadores aritméticos

Todas as variáveis apresentadas até agora estavam armazenando valores não negativos, isso porque é necessário utilizar um operador para descrever um número negativo. Há um operador que deixa explícito que um número é positivo, porém todo número é implicitamente positivo quando nenhum sinal é colocado, assim como na matemática.

O operador que retorna o correspondente negativo de um tipo é o sinal de menos (-). O operador que retorna o correspondente positivo de um tipo é o sinal de mais (+). A sintaxe associada é apresentada no código 11.

```

1  //...
2  -<name>; //Só aparece em um contexto onde o retorno é utilizado
3
4  +<name>;
5  //...

```

Código 11: Sintaxe dos sinalizadores aritméticos

Vale notar que, matematicamente, são equivalentes a multiplicar por -1 e $+1$, respectivamente, ou seja, o operador do sinal de mais não realiza operação útil neste contexto, porém em outros ele é necessário. Exemplos do uso padrão são encontrados no código 12.

```

1  //...
2  int A(-5); //A vale -5
3  int B(+A); //A vale -5, B vale -5
4  int C(-A); //A vale -5, B vale -5, C vale 5
5
6  int D(5); //D vale 5,
7  int E(-D); //D vale 5, E vale -5
8  int F(+D); //D vale 5, E vale -5, F vale 5
9  //...

```

Código 12: Exemplo de sinalizadores aritméticos

4.1.3 Negador lógico

Em lógica, negar significa inverter o valor, em computação também. O operador de negação utiliza o ponto de exclamação (!), e normalmente é utilizado com tipos booleanos. Realiza a operação *NOT* bit-a-bit. A sintaxe associada é apresentada no código 13. Vale ressaltar que não se relaciona à operação de fatorial da matemática, que utiliza o mesmo símbolo. Uma tabela verdade de referência é apresentada na tabela 4.1.


```

1 //...
2 !<nome>; //Seu uso só é coerente quando o retorno é utilizado
3 //...

```

Código 13: Sintaxe da negação lógica

Alguns exemplos podem ser encontrados no código 14.

```

1 //...
2 bool A(true); //A vale 1
3 bool B(!A); //B vale 0
4 bool C(!B); //C vale 1
5
6 bool D(!true); //D vale 1
7 bool E(!false); //E vale 0
8
9 int F(10); //F vale 10
10 int G(!F); //G vale 0
11 int H(!G); //H vale 1
12 //...

```

Código 14: Exemplo de negação lógica

4.1.4 Complemento binário

O operador de complemento binário é o til (~). Esta operação faz uma inversão bit-a-bit no número, ou seja, transforma 1 em 0 e 0 em 1 para todo o bit. Sua sintaxe é apresentada no código 15 e um exemplo de uso é apresentado no código 16. Vale ressaltar que o sinal de positividade é representado por um bit em tipos não **unsigned**, portanto o complemento binário irá inverter o sinal nesses casos.

```

1 //...
2 ~<nome>; //Novamente, seu uso só é coerente se o retorno é utilizado
3 //...

```

Código 15: Sintaxe do complemento binário

```

1 //...
2 unsigned char A(0b10100101); //A vale 0b10100101 ou 0xA5
3 unsigned char B(~A); //B vale 0b01011010 ou 0x5A
4 /*
5 unsigned char C(Ã); //Tome cuidado para isso não acontecer
6 */
7 //...

```

Código 16: Exemplo de complemento binário

4.2 Binários

Assim como os operadores unários, os binários seguem um padrão na sintaxe, apresentada no código 17 e um exemplo genérico no código 18.

```

1 //...
2 <nome1> <operador> <nome2>; //Onde <nome> refere-se ao identificador da variável
3 //...

```

Código 17: Sintaxe geral para operadores binários

```

1 //...
2 <tipo1> <nome1> (<valor1>);
3 <tipo2> <nome2> (<valor2>);
4 int A(<nome1> <operador> <nome2>); //Operadores aparecem num contexto
   adequado
5 //...

```

Código 18: Exemplo genérico de operadores binários

4.2.1 Atribuição simples

O primeiro operador binário apresentado é o de atribuição simples, que utiliza o sinal de igual (=). É responsável pela atribuição de novos valores às variáveis, depois do momento de sua inicialização. Diz-se que o operador de atribuição simples tem característica de agrupamento da direita para a esquerda, isto é, o valor do tipo à direita do operador é atribuído à variável da esquerda. Sua sintaxe básica é apresentada no código 19.

```

1 //...
2 <alvo> = <item>;
3 //...

```

Código 19: Sintaxe do operador de atribuição

O operador de atribuição é o mais comum ao inicializar uma variável. Alguns exemplos podem ser encontrados no código 20.

```

1 //...
2 int A;
3 A = 10; //A passa a valer 10
4 float B = 5.1; //Operadores podem ser usados na declaração.
5
6 float C(B = 13.25); //Todo operador retorna o valor de sua operação
7
8 int D = A = 20;
9 /*
10  Da direita para a esquerda para a direita, A passa a valer 10
11  Então D passa a ter o valor da operação à direita, 10
12 */
13 //...

```

Código 20: Exemplo do operador de atribuição

4.2.2 Aritméticos

Os operadores aritméticos são representados por símbolos semelhantes aos utilizados na matemática. A soma pelo sinal de mais (+), subtração pelo sinal de menos (-), a multiplicação utiliza o asterisco (*), a divisão a barra (/) e o módulo (resto) o símbolo de porcentagem (%). Diferente do operador de atribuição simples, os operadores aritméticos não alteram o valor de variáveis, apenas retornam o cálculo. Um exemplo de sintaxe é apresentado no código 21.

```

1 //...
2 <valor1> + <valor2>; //O uso só é coerente em casos onde o retorno é
   utilizado
3 <valor1> - <valor2>;
4 <valor1> * <valor2>;
5 <valor1> / <valor2>;
6 <valor1> % <valor2>;
7 //...

```

Código 21: Sintaxe dos operadores aritméticos

Também são de agrupamento da direita para a esquerda. Estes operadores funcionam da mesma maneira que na matemática. Na soma, o valor à direita é adicionado ao valor à esquerda. Na subtração, o valor à direita é diminuído do valor à esquerda. Na multiplicação, o valor à direita é multiplicado à esquerda. Na divisão e no módulo, o valor à esquerda é o dividendo e o valor à direita o divisor. Para valores do tipo `float` e `double`, a divisão retorna o valor decimal e o operador módulo não está definido. Para valores do tipo `int`, as operações de divisão e resto de funcionam como no mecanismo de divisão com chave, onde a divisão retorna o quociente, e o módulo retorna o resto da divisão. Exemplos podem ser encontrados no código 22, inclusive o caso de módulo equivalente ao da equação 4.1.

$$\begin{array}{l} E \mid F \\ G \mid H \end{array} \Rightarrow \begin{array}{l} 13 \mid 5 \\ 3 \mid 2 \end{array} \quad (4.1)$$

```

1 //...
2 int A(45 + 5); //A vale 50
3 int B(A - 15); //B vale 35
4 int C(B - A); //C vale -15
5
6 int D(A + B - C); //D vale o mesmo que 50 + 35 - (-15), ou seja 100
7
8 int E(13), F(5); //E vale 13, F vale 5
9 int G(E % F), H(E / F); //G vale 3, H vale 2
10 int I(F * H); //I vale 10
11 int J(I + G); //J vale 13
12
13 float K(13.0f), L(5.0f);
14 float M(K / L); //K vale 2.6
15 float N(L * M); //N vale 13
16 //...

```

Código 22: Exemplo de operadores aritméticos

O operador módulo é especialmente útil quando precisa-se verificar se um número é múltiplo de outro, onde seu retorno é 0. Assim como na matemática, a divisão por zero é proibida em C++, gerando um erro e até a finalização incompleta do programa.

4.2.3 Deslocadores bit-a-bit

Uma operação possível em C++ é o deslocamento lateral bit-a-bit, que consiste em mover os bits de uma memória para direita ou esquerda. O operador de deslocamento para a esquerda utiliza o sinal de menor duas vezes (<<). O operador de deslocamento para a direita utiliza o sinal de maior duas vezes (>>). O valor à esquerda é o deslocado, o valor à direita é a quantidade de bits deslocados. A sintaxe associada é encontrada no código 23.

```

1 //...
2 (valor) << (deslocamentos); //Só é coerente quando o retorno é utilizado
3 (valor) >> (deslocamentos);
4 //...

```

Código 23: Sintaxe dos operadores de deslocamento

Matematicamente esta operação equivale a multiplicar ou dividir o valor por uma potência de dois, como na equação 4.2 para o deslocamento à esquerda e na equação 4.3 para o deslocamento à direita. Porém, computacionalmente, os deslocamentos são muito mais rápidos que divisões e multiplicações, por isso é recomendada sua utilização em casos de operações com potências de 2.

$$V \cdot 2^{+S} \quad (4.2)$$

$$V \cdot 2^{-S} \quad (4.3)$$

Onde V é o valor à esquerda do operador e S o valor à direita. Alguns exemplos de operadores de deslocamento podem ser encontrados no código 24.

```

1 //...
2 unsigned char A(0b01100000);
3 unsigned char B(A>>3); //B vale 0b00001100
4 unsigned char C(B*8); //C vale 0b01100000
5
6 int D(30); //D vale 30
7 int E(D>>1); //E vale 30/(2), ou seja 15
8 int F(D<<2); //F vale 30*(4), ou seja, 120
9 //...

```

Código 24: Exemplos dos operadores de deslocamento

4.2.4 Lógicos bit-a-bit

Existem três operadores lógicos bit-a-bit:

- *OR*, que utiliza a barra vertical (`|`). Esta operação retorna um bit para cada par de bits dos valores de entrada. Retorna 1 caso algum dos bits comparados seja 1, e 0 caso ambos sejam 0, sempre comparando bit-a-bit dos valores.
- *AND*, que utiliza o *e comercial* (`&`). Esta operação retorna um bit para cada par de bits dos valores de entrada. Retorna 1 caso ambos os bits comparados sejam 1, e 0 caso algum seja 1, sempre comparando bit-a-bit dos valores.
- *XOR*, que utiliza o circunflexo (`^`). Esta operação retorna um bit para cada par de bits dos valores de entrada. Retorna 1 caso os bits comparados sejam diferentes, e 0 caso sejam iguais.

A sintaxe associada é encontrada no código 25. A tabela 4.1 apresenta uma relação entre as operações lógicas e seus resultados, é denominada tabela verdade. Exemplos dos operadores lógicos bit-a-bit podem ser encontrados no código 26. O conjunto de operações bit-a-bit é apresentado na forma matemática na equação 4.4, vale ressaltar que uma operação bit-a-bit significa que o operador lógico será realizado entre o bit de um valor e o bit correspondente no outro valor, ou seja, o primeiro bit de um opera com o primeiro bit do outro.

```

1 //...
2 <valor1> | <valor2>;
3 <valor1> & <valor2>;
4 <valor1> ^ <valor2>;
5 //...

```

Código 25: Sintaxe dos operadores lógicos bitwise

Tabela 4.1: Tabela verdade para operadores lógicos

A	B	NOT A	NOT B	A OR B	A AND B	A XOR B
0	0	1	1	0	0	0
0	1	1	0	1	0	1
1	0	0	1	1	0	1
1	1	0	0	1	1	0

```

1 //...
2 unsigned char A(0b10101011);
3 unsigned char B(0b01100100);
4 unsigned char C(A|B); //C vale 0b11101111
5 unsigned char D(A&B); //D vale 0b00100000
6 unsigned char E(A^B); //E vale 0b11001111
7 //...

```

Código 26: Exemplo dos operadores lógicos bitwise

$$\begin{array}{rcl}
 & 10101011 & 10101011 & 10101011 \\
 \text{OR} & \frac{01100100}{11101111} & \text{AND} & \frac{01100100}{00100000} & \text{XOR} & \frac{01100100}{11001111}
 \end{array} \quad (4.4)$$

4.2.5 Atribuição composta

O conjunto de operadores de atribuição composta trabalha como uma combinação do operador de atribuição simples com outro operador binário. Todo operador de atribuição composta tem o mesmo formato, que consiste no operador de interesse seguido do operador de atribuição simples, sem espaços vazios. Estes operadores realizam um cálculo sobre o valor de uma variável e atribuem este valor à própria variável. A sintaxe associada é apresentada no código 27.

```

1 //...
2 <nome> = <nome> <operador> <valor>; //Onde nome é uma variável
3 <nome> <operador>= <valor>;          //Operador de atribuição composta
4 //...

```

Código 27: Sintaxe dos operadores de atribuição composta

Uma relação entre os operadores binários descritos e os operadores de atribuição é apresentada na tabela 4.2, alguns exemplos de uso podem ser encontrados no código 28.

Tabela 4.2: Relação de operadores de atribuição composta e seus equivalentes

Composto	Equivalente
A += B;	A = A + B;
A -= B;	A = A - B;
A *= B;	A = A * B;
A /= B;	A = A / B;
A %= B;	A = A % B;
A >>= B;	A = A >> B;
A <<= B;	A = A << B;
A = B;	A = A B;
A &= B;	A = A & B;
A ^= B;	A = A ^ B;

```

1 //...
2 int A(0), B(10); //A vale 0, B vale 10
3 A += 1;          //A vale 1, B vale 10
4 B /= 2;          //A vale 1, B vale 5
5 A *= 100;        //A vale 100, B vale 5
6 B <=< 3;          //A vale 100, B vale 40
7 B &= A;          //A vale 100, B vale 32
8 A %= B;          //A vale 4, B vale 32
9 //...

```

Código 28: Exemplos de operadores de atribuição composta

4.2.6 Comparadores

Existem operadores construídos para a comparação de valores, verificando se são iguais ou diferentes, e ainda qual deles tem o maior ou o menor valor.

O operador comparador de igualdade utiliza o sinal e igual duas vezes(==). Este operador não altera o valor das variáveis. Seu retorno é booleano, sendo **true** caso os valores à direita e à esquerda sejam iguais. Não confundir com o operador de atribuição simples.

O operador comparador de diferença utiliza o ponto de exclamação seguido de um sinal de igual (!=). Uma forma de lembrar deste operador é relacionar ao negador lógico, que utiliza apenas o ponto de exclamação. Retorna

`true` quando os valores são diferentes. Equivale a negar logicamente o retorno de um comparador de igualdade. Estes operadores podem ser usado com qualquer tipo primitivo.

Os demais comparadores verificam se um valor é maior, ou menor, que outro, utilizando os sinais matemáticos correspondentes. Estes operadores precisam de tipos numéricos. Verificar se o valor à esquerda é maior que o à direita, utilizamos o sinal de maior (>). Verificar se o valor à esquerda é menor que o à direita, utilizamos o sinal de menor (<). Caso os valores sejam iguais, o operador retornará `false`. Existem dois operadores adicionais, um que verificam se o valor à esquerda é *maior ou igual* ao valor à direita, utilizando o sinal de maior seguido do sinal de igual (>=). O operador recíproco, que verifica se um valor é *menor ou igual* utiliza o sinal de menor seguido do sinal de igual (<=).

Um exemplo de sintaxe é apresentado no código 29. Exemplos do uso podem ser encontrados no código 30.

```
1 //...
2 <valor1> == <valor2>; //Só é coerente quando o retorno é utilizado
3 <valor1> != <valor2>;
4 <valor1> < <valor2>;
5 <valor1> > <valor2>;
6 <valor1> <= <valor2>;
7 <valor1> >= <valor2>;
8 //...
```

Código 29: Sintaxe de operadores de comparação

```
1 //...
2 bool A(true);
3 int B(10), C(10), D(15);
4
5 bool E(B>C); //E vale 0
6 bool F(A==E); //F vale 0
7
8 bool G(D>=B); //G vale 1
9 bool H(E!=G); //H vale 1
10 bool I(B==C); //I vale 1
11 //...
```

Código 30: Sintaxe de operadores de comparação

4.2.7 Lógicos booleanos

Semelhantes aos lógicos bit-a-bit, os lógicos booleanos utilizam os mesmo símbolos, porém duas vezes, então a operação *OR* utiliza o símbolo de barra vertical duas vezes (||), e a operação *AND* utiliza o símbolo *e* *comercial* duas vezes (&&). Estes operadores tem como finalidade a interação entre valores lógicos booleanos, como os tipo `bool` ou os operadores de comparação. A tabela 4.1 também serve de referência para a relação destes operadores.

O código 31 apresenta a sintaxe associada destes operadores, e o código 32 apresenta exemplos de uso.

```
1 //...
2 <valor1> || <valor2>; //Coerente apenas quando o retorno é utilizado
3 <valor1> && <valor2>;
4 //...
```

Código 31: Sintaxe de operadores lógicos booleanos

```
1 //...
2 bool T(true), F(false);
3
4 bool A(T || F); //A vale 1
5 bool B(T && F); //B vale 0
6 //...
```

Código 32: Exemplos dos operadores lógicos booleanos

4.3 Ternário

Há apenas um operador ternário, sua finalidade é selecionar um valor de retorno a partir de uma condição lógica. A sintaxe do operador ternário é apresentada no código 33. Consiste em um valor lógico booleano, como os tipo `bool` ou os operadores de comparação, um ponto de interrogação (?), o valor para retorno caso a condição lógica seja `true`, dois pontos (:) e um valor caso a condição lógica seja falsa.

```
1 //...
2 <condicional> ? <valor1> : <valor2>;
3 //...
```

Código 33: Sintaxe do operador ternário

Este operador foi construído para situações onde se faz necessária a utilização de valores relativos à uma condição lógica. Alguns exemplos podem ser encontrados no código 34.

```
1 //...
2 bool A(true);
3 int C(10), D(15);
4 bool E(C>=D); //E vale 0
5 bool F(A==E); //F vale 0
6 int G(F?10:50); //G vale 50
7 ///...
```

Código 34: Exemplo de operador ternário

4.4 Precedência de operadores

Alguns operadores tem prioridade em relação à outros, tendo suas operações realizadas antes dos outros operadores, como acontece na matemática com a multiplicação e a soma, onde a multiplicação tem prioridade e é calculada antes da soma. Os operadores do C++ apresentam uma extensa lista de ordem de precedência, presentes na tabela 4.3. Equações podem ser montadas utilizando conjuntos de operadores e valores, respeitando a precedência.

Há um operador especial desenvolvido para priorizar outros, permitindo a sobreposição da precedência padrão, utiliza uma estrutura semelhante ao bloco de código, porém utilizando parênteses (). Dentro deste bloco de preferência podem ser colocadas equações e até outros blocos.

Alguns exemplos de precedência poder ser encontradas no código 35.

Tabela 4.3: Ordem de precedência de operadores

Operador	Descrição
()	preferencial
++, --	posfixo
++, --	prefixo
~, !	lógico
+, -	sinalizadore
*, /, %	aritimético
+, -	aritimético
<<, >>	deslocador
<, <=, >=, >	comparador
==, !=	comparador
&	lógico
^	lógico
	lógico
&&	lógico
	lógico
?:	ternário
=	atribuidor
+=, -=, *=, /=, %=, &=, ^=, =, <=, >=	atribuidor

```

1 //...
2 int A = 25 * 40;
3 int B = 1 << 4;
4 A /= B + 4;
5
6 A = A + B;
7 B = A - B;
8 A = A - B;
9
10 float C = A > 200 ? A * (50.0f - 0.003f) : B % 5;
11 //...

```

Código 35: Exemplo de operadores e precedência

Capítulo 5

Controladores de Fluxo

O fluxo descreve a lista ordenada de comandos que são realizados por um programa. Esta lista segue a ordem de comandos sequencialmente descritos no código, como ilustrado na figura 5.1. Esta sequência é única, ou seja, não existem situações paralelas, pois em C++ não há por padrão a multiexecução (*multi thread*). Porém um fluxo que não sofre alterações nem sempre é útil, por isso existem os controles de fluxo, que podem mudar a direção do fluxo e criar repetições.

A representação gráfica de um fluxo é chamada de fluxograma. Os blocos retangulares representam os comandos.

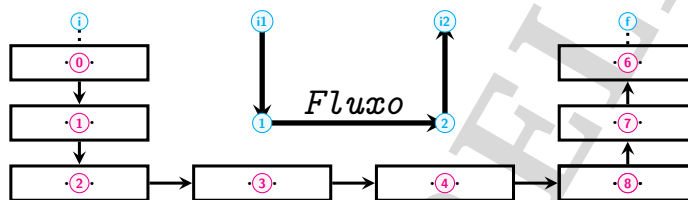


Figura 5.1: Fluxograma de fluxo simples

5.1 Decisões na direção do fluxo

Quando o fluxo pode tomar mais de um rumo é necessário um critério de decisão, indicando qual dos caminhos tomar, sendo o outro ignorado. Um exemplo de fluxo com ambiguidade é apresentado na figura 5.2. Note que não é possível escolher um lado sem um critério de decisão, gerando um problema de ambiguidade. Precisamos de uma estrutura de decisão.

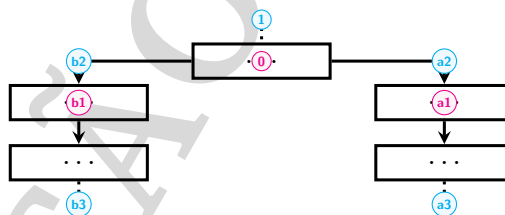


Figura 5.2: Fluxograma de fluxo ambíguo

5.1.1 O *if* e a estrutura básica de decisão

A mais simples estrutura de decisão é definida pela palavra-chave *if*, que recebe o que é chamado de *argumento* entre parênteses. O argumento é do tipo *bool*, uma condição lógica onde, caso verdadeiro, o comando associado ao *if* será realizado, caso contrário, não. Um exemplo da sintaxe linear básica é apresentada no código 36, e a figura 5.3 apresenta um fluxograma que ilustra as direções do fluxo. Nos fluxogramas, os losangos representam decisões, conforme a indicação de verdadeiro ou falso.

1 || //...

```

2  | if(<cond>) <comand>;
3  | //...

```

Código 36: Estrutura de *if* simples linear

Quando apenas um comando é condicionado pelo *if*, adicionamos este comando seguido da definição do direcionador.

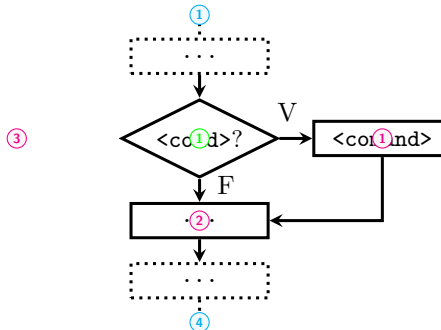


Figura 5.3: Fluxograma de *if* simples linear

Quando mais de um comando está condicionado ao mesmo estado lógico, é possível fazer uma sequência de estruturas repetindo o direcionador de fluxo. Porém toda vez que o fluxo encontra um direcionador, um pequeno tempo é gasto processando tal alteração. Para evitar este processamento desnecessário, utiliza-se o direcionador de fluxo junto a um bloco de código, já que este pode ser considerado como uma composição de comandos. O código 37 apresenta um exemplo de sintaxe de estrutura de decisão simples utilizando um bloco de código, a figura 5.4 apresenta um fluxograma que representa esta estrutura genérica.

```

1  | //...
2  | if(<cond>)
3  | {
4  |     <comand1>;
5  |     //...
6  |     <comandN>;
7  | }
8  | //...

```

Código 37: Estrutura de *if* simples blocular

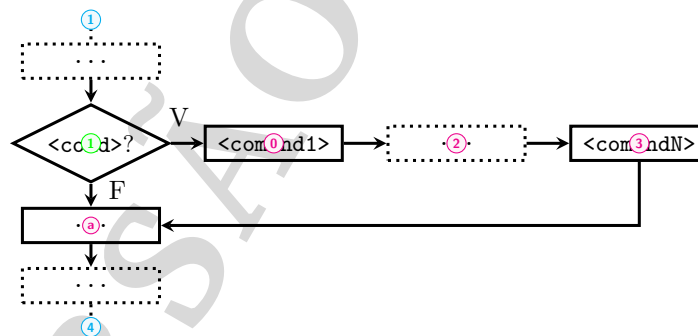


Figura 5.4: Fluxograma de *if* simples blocular

O *if* é uma controlador de fluxo versátil, que pode ser usado nas mais diversas situações, porém existem mais nuances sobre seu uso que serão apresentadas posteriormente. Alguns exemplos de uso do *if* podem ser encontrados no código 38.

```

1  | //...
2  | int A(50), B(25), C(15);

```

```

3  if (A<B) C *= 2;  //Alteração de valor condicionada
4  if (C<A)          //Inversão de valores de variáveis
5  {
6      B = A;
7      A = C;
8      C = B;
9  }
10 //...

```

Código 38: Estrutura de *if* Exemplo de utilização do *if*

5.1.2 O *else* e a estrutura complementar

Existem casos onde comandos somente podem ser realizados perante a condições lógicas, nestes casos utilizamos o *if*. Existem também os casos onde comando são mutuamente excludentes, ou seja, a condição que permite o primeiro, inibe o segundo.

Num primeiro momento, pode-se imaginar que utiliza-se uma sequência de estruturas de decisão simples, com condições inversas. Tal método funciona, porém a verificação de condição requer processo de máquina, de maneira que em programas grandes é perceptível a baixa no desempenho.

Para evitar isso, existe o complementar na estrutura de decisão. Utilizando a palavra-chave *else* para condições complementares. Somente quando a condição o *if* for falsa o comando associado ao *else* será executado. O código 39 apresenta um exemplo da sintaxe linear complementar, e a figura 5.5 apresenta o fluxograma associado.

```

1  //...
2  if(<cond>) <comandA>;
3  else <comandB>;
4  //...

```

Código 39: Estrutura de *if* complementares linear

Assim como no *if* linear, a estrutura do *if else* pode ser escrita adicionando o comando associado seguido da definição do direcionador.

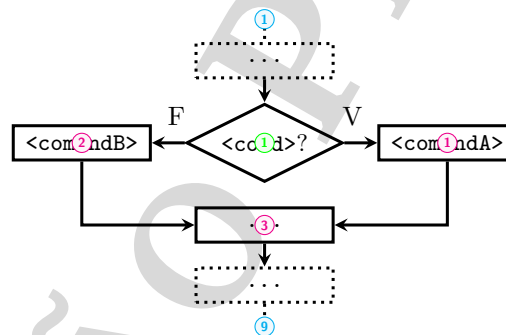


Figura 5.5: Fluxograma de *if* complementares linear

Igualmente a estrutura de um *if* simples, o *if else* também pode trabalhar com blocos de código, seguindo a sintaxe apresentada no código 40, a figura 5.6 apresenta o fluxograma associado.

```

1  //...
2  if(<cond>)
3  {
4      <comandA1>;
5      //...
6      <comandAN>;
7  }
8  else
9  {

```

```

10     <comandB1>;
11     //...
12     <comandBM>;
13 }
14 //...

```

Código 40: Estrutura de *if* complementares blocular

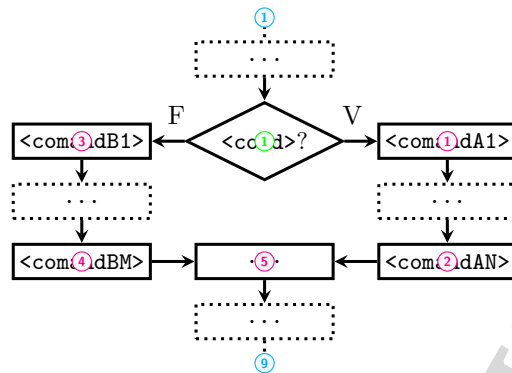


Figura 5.6: Fluxograma de *if* complementares blocular

5.1.3 Estruturas aninhadas

Dentro de qualquer bloco de código é possível declarar controladores de fluxo, inclusive controladores com blocos, e sucessivamente. A este tipo de estrutura é utilizado o termo *estrutura aninhada*, que consiste em controladores de fluxo dentro de controladores de fluxo.

O código 41 mostra um exemplo simples de sintaxe de estrutura aninhada de *if* composto com *else*, e a figura 5.7 apresenta um fluxograma mostrando uma estrutura aninhada de *if else*, note que o controle de fluxo é colocado dentro de um bloco de código com outros comandos.

```

1  //...
2  if (<cond0>)
3  {
4      //...
5      if (<condA>)
6      {
7          <comandA1>;
8          //...
9          <comandAN>;
10     }
11     else
12     {
13         <comandB1>;
14         //...
15         <comandBM>;
16     }
17     //...
18 }
19 else
20 {
21     //...
22     if (<condB>)
23     {
24         <comandC1>;
25         //...
26         <comandCP>;

```

```

27 }
28 else
29 {
30     <comandD1>;
31     //...
32     <comandDQ>;
33 }
34 //...
35 }
36 //...

```

Código 41: Estrutura de *if* aninhado

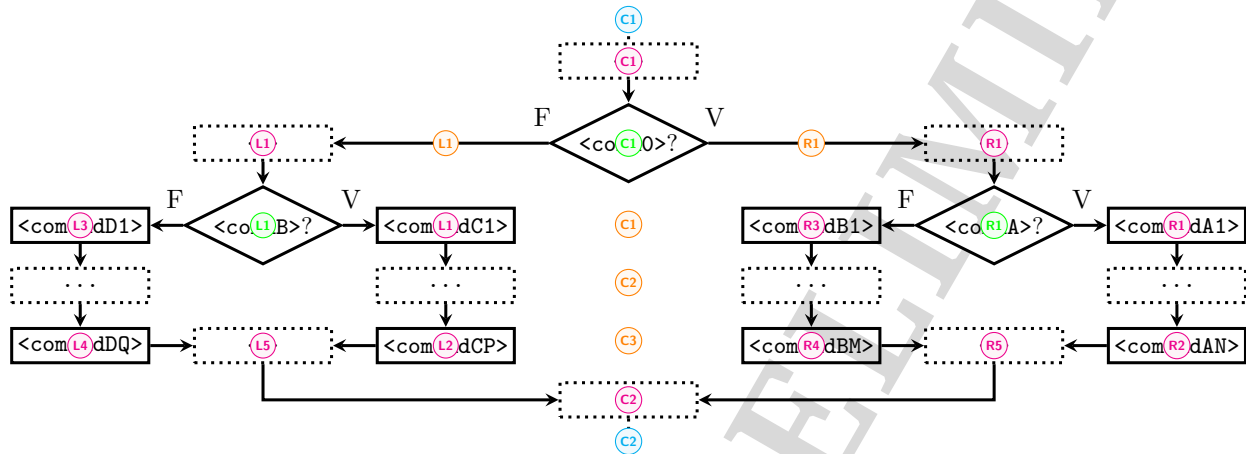


Figura 5.7: Fluxograma de *if* aninhado

Um exemplo interessante de estrutura aninhada é apresentada no código 42. Note como a escolha de condicionais não leva em consideração um intervalo delimitado, mas considera verificações de números pelo topo, ou seja, começando do maior. Esta verificação tem características especiais que melhoram o desempenho do programa.

```

1 //...
2 float nota(8.7);
3 char conceito('\0');
4 //...
5     if (nota >= 9.5)    conceito = 'A';
6 else if (nota >= 8.0)  conceito = 'B';
7 else if (nota >= 7.0)  conceito = 'C';
8 else if (nota >= 4.5)  conceito = 'D';
9 else                  conceito = 'F';
10 //...

```

Código 42: Exemplo de *if else* aninhado

Em uma comparação rápida entre os códigos 42 e 43 não é difícil notar que todo *if* no segundo exemplo será processado, o que é desnecessário, visto que apenas um deles poderá ser verdadeiro, portanto um caso de complemento seria suficiente, como no primeiro exemplo.

```

4 //...
5 if (10.0>=nota && nota >= 9.5) conceito = 'A';
6 if (9.5 > nota && nota >= 8.0) conceito = 'B';
7 if (8.0 > nota && nota >= 7.0) conceito = 'C';
8 if (7.0 > nota && nota >= 4.5) conceito = 'D';
9 if (4.5 > nota && nota >= 0.0) conceito = 'F';
10 //...

```

Código 43: Exemplo de *if* sequencial

Similar ao exemplo do código 42, o código 44 utiliza o método espelho, verificando os números pela base, ou seja, começando do menor.

```
4 //...
5     if (nota < 4.5)    conceito = 'F';
6 else if (nota < 7.0)    conceito = 'D';
7 else if (nota < 8.0)    conceito = 'C';
8 else if (nota < 9.5)    conceito = 'B';
9 else                   conceito = 'A';
10 //...
```

Código 44: Exemplo de *if else* aninhado

A compreensão do código 42 pode não ser clara em um primeiro momento para programadores iniciantes, o código 45 é equivalente, deixando cada bloco de código explicitamente declarado, facilitando a leitura e compreensão. Vale notar, para a estrutura *if else*, um *else* só é permitido se associado a um *if*, e, em uma estrutura aninhada, cada *else* é associado ao *if* anterior mais próximo.

```
4 //...
5 if (nota >= 9.5)    conceito = 'A';
6 else
7 {
8     if (nota >= 8.0)    conceito = 'B';
9     else
10 {
11     if (nota >= 7.0)    conceito = 'C';
12     else
13 {
14     if (nota >= 4.5)    conceito = 'D';
15     else conceito = 'F';
16 }
17 }
18 }
19 //...
```

Código 45: Exemplo de *if else* aninhado com blocos explicitamente delimitados

5.1.4 O *switch* e a estrutura composta

Quando uma grande gama de opções de fluxo é possível, e todas estas opções estão relacionadas a mesma variável para decisão, podemos utilizar uma grande sequência de *if else*, como no código 46.

```
1 //...
2 int A;
3 //...
4 if (A == 1)
5 {
6     <comand1A>;
7     <comand1B>;
8     <comand1C>;
9     <comand1D>;
10    <comand1E>;
11 }
12 else
13 {
14     if (A == 2) <comand2>;
15     else
16     {
17         if (A == 3)
```

```

18 {
19     <comand3>;
20     <comand4A>;
21     <comand4B>;
22 }
23 else
24 {
25     if (A == 4)
26     {
27         <comand4A>;
28         <comand4B>;
29     }
30     else <comand0>;
31 }
32 }
33 }
34 //...

```

Código 46: Estrutura composta formada por *if*

Porém nem sempre é conveniente construir uma sequência, para isso existe a estrutura **switch case**. Um fluxo equivalente ao apresentado no código 46 é mostrado no fluxograma da figura 5.8, e o código 47 apresenta a o mesmo fluxo utilizando o **switch**, e mostra alguns detalhes da sintaxe.

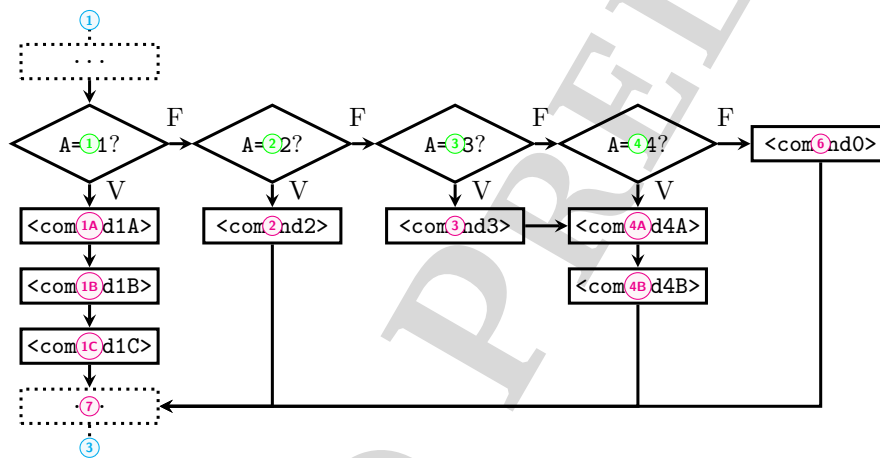


Figura 5.8: Fluxograma de estrutura de decisão composta

Os valores de um **switch** são colocados à direita da palavra-chave **case**, seguidos de dois pontos (:). A estrutura do **switch** é fundamentada em condições de igualdade, então apenas se faz útil quando um conjunto discreto de valores é possível. Sequências como no código 42 não podem ser construídas utilizando um **switch**.

Dentro de cada **case** podem existir vários comandos, inclusive controles de fluxo, como **if else** e até **switch**, e os demais descritos posteriormente.

Cada vez que uma das condicionais é verdadeira, todos os comandos subsequentes serão realizados, até se estiverem relacionados a outras condicionais, isso apenas não ocorre quando a palavra-chave **break** é encontrada, que causa uma quebra no fluxo e sai da estrutura do **switch case** e retoma o fluxo normal. O **break** tem uma função semelhante em outras estruturas que serão mostradas posteriormente.

A última característica a ser notada é o **default**, que somente é executada quando nenhum dos outros casos é verdadeiro (ou quando não há **break** previamente). Este caso especial serve, principalmente, para situações de erro, onde nenhuma das condições foi aceita.

```

1 //...
2 int A;
3 //...
4 switch(A)

```

```

5 {
6     case 1:
7         <comand1A>;
8         <comand1B>;
9         <comand1C>;
10        <comand1D>;
11        <comand1E>;
12        break;
13    case 2:
14        <comand2>;
15        break;
16    case 3:
17        <comand3>;
18    case 4:
19        <comand4A>;
20        <comand4B>;
21        break;
22    default:
23        <comand0>;
24 }
25 //...

```

Código 47: Estrutura composta formada por *switch*, *case*, *break* e *default*

Note como a estrutura do *switch* é mais limpa que a sequência de *if else* equivalente. Dentro de cada *case* pode-se delimitar um bloco de código para melhor compreensão, mas não é obrigatório.

Um exemplo de *switch* é mostrado no código 48. Este exemplo é sequencial ao código 42.

```

10 //...
11 bool aprovado;
12 //...
13 switch(conceito)
14 {
15     case 'A':
16     case 'B':
17     case 'C':
18     case 'D':
19         aprovado = true;
20         break;
21     case 'F':
22         aprovado = false;
23         break;
24 }
25 //...

```

Código 48: Exemplo de *switch*

5.2 Repetições no fluxo

Em muitas situações é comum ao programador se deparar com repetições de comandos ou até blocos de código. Naturalmente é possível copiar o código a quantidade de vezes necessária para que a repetição de uma parte seja efetuada, porém isso tende a gerar problemas.

Por exemplo, uma das técnicas para o cálculo de mínimo divisor comum (MDC) consiste em subtrair o menor valor do maior sucessivamente até que um deles valha 0, então o outro valor será o MDC obtido pelo processo, a esta técnica é dado o nome de *algoritmo de Euclides*. Sem utilizar um controle de fluxo que possibilita a repetição de comandos, o programador precisa conhecer os valores a serem calculados. O código 49 mostra como fica a estrutura sem a utilização de estruturas de repetição.


```

1  //...
2  int a, b, X, Y;
3  a=45; b=93;
4  //...
5  X = a>b?a:b;  //#1
6  Y = a<b?a:b;
7  a = X-Y;
8  b = Y;
9
10 X = a>b?a:b;  //#2
11 Y = a<b?a:b;
12 a = X-Y;
13 b = Y;
14
15 //Repete-se 18 vezes!
16
17 X = a>b?a:b;  //#17
18 Y = a<b?a:b;
19 a = X-Y;
20 b = Y;
21
22 X = a>b?a:b;  //#18
23 Y = a<b?a:b;
24 a = X-Y;      //a vale 3, o resultado
25 b = Y;        //b vale 0
26 //...

```

Código 49: Algoritmo de Euclides sem estrutura de repetição

Porém não faz sentido a utilização do código para calcular o valor a cada vez que o programa é executado, se o programador sabe quantas vezes será repetido, já conhece os valores a serem calculados, pode calcular e utilizar os valores finais no código, que o deixaria muito mais eficaz.

Naturalmente existirão situações onde o programador conhece a quantidade de repetições necessárias porém não pode fornecer o resultado final diretamente, e para estes casos vale ressaltar: copiando blocos de código, se há um erro de digitação ou de lógica, este erro será copiado, então localizar e consertar tal erro será mais difícil.

Uma repetição genérica de comandos pode ser um problema de ambiguidade, como apresentado na figura 5.9. É de se esperar que um `if` sirva para se livrar de tal ambiguidade, a estrutura mais simples de repetição faz isso.

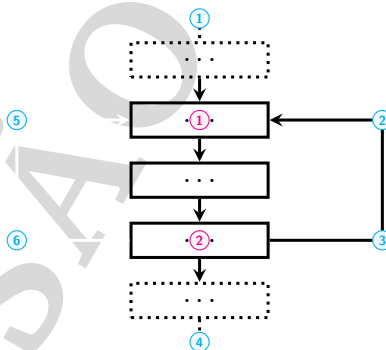


Figura 5.9: Fluxograma de fluxo repetitivo

5.2.1 O `while` e a estrutura de repetição indefinida

A mais simples estrutura de repetição é definida pela palavra-chave `while`, que recebe um tipo `bool` como argumento. Enquanto esta condição lógica for verdadeira, a repetição será realizada, terminando quando a verificação é realizada e a condição for falsa.

Um exemplo da sintaxe linear básica é apresentada no código 50, e a figura 5.10 apresenta um fluxograma que ilustra as direções do fluxo.

Assim como no **if**, é permitida a utilização de blocos de código. Vale notar que a verificação de condição apenas é feita uma vez a cada repetição, portanto a condição do argumento pode ser alterada várias vezes dentro do bloco associado, mas só será considerado o valor no momento de verificação

```

1 //...
2 while(<cond>) <comand>;
3 //...
```

Código 50: Estrutura de *while* linear

A estrutura do **while** é idêntica à do **if** simples.

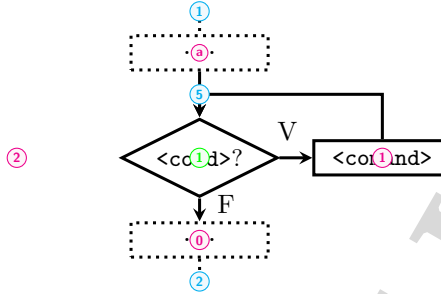


Figura 5.10: Fluxograma de *while* linear

O exemplo de **while** construído com bloco de código é apresentado no código 51 e o fluxograma associado está na figura 5.11.

```

1 //...
2 while(<cond>)
3 {
4     <comand1>;
5     //...
6     <comandN>;
7 }
8 //...
```

Código 51: Estrutura de *while* blocular

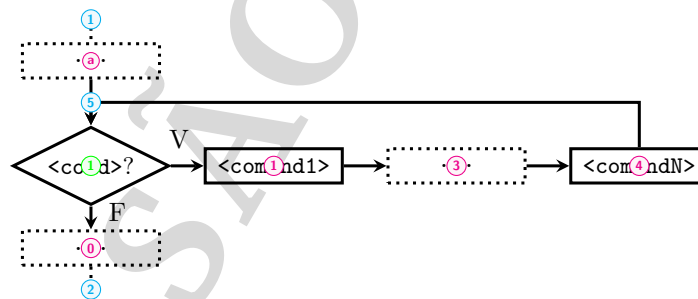


Figura 5.11: Fluxograma de *while* blocular

O mesmo algoritmo de Euclides do código 49 pode ser escrito como no código 52. Note que nesse caso o programador não precisa conhecer os valores iniciais, note também que o algoritmo depende de uma condição de parada, que indica que seu término foi atingido, cabe ao programador construir um critério de parada adequado ao problema.

De muito vale ressaltar: no **while**, caso a condicional não seja verdadeira no primeiro instante, a repetição não se inicia.

```

4 //...
5 while (b!=0)
6 {
7     X = a>b?a:b;
8     Y = a<b?a:b;
9     a = X-Y;
10    b = Y;
11 }
12 //...

```

Código 52: Algoritmo de Euclides com estrutura repetição

5.2.2 O *do* e a estrutura de início obrigatório

Esta estrutura é muito semelhante ao *while*, porém é garantida que o comando seja executado ao menos uma vez.

O *do while* consiste em uma estrutura de repetição onde a palavra-chave *do* fica antes do comando, que será executado ao menos uma vez, seguido da palavra-chave *while* com o argumento da condicional. A principal diferença a ser considerada em comparação ao *while* o local da comparação, onde no primeiro é feita no começo e nesse é feita no final.

A sintaxe básica é apresentada no código 53 e seu fluxograma na figura 5.12.

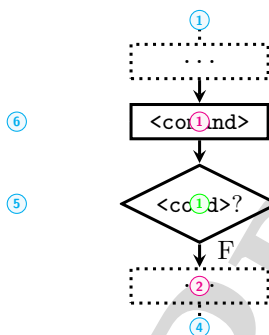


Figura 5.12: Fluxograma de *do while* linear

```

1 //...
2 do <comand>; while(<cond>);
3 //...

```

Código 53: Estrutura de *do while* linear

Note que, após o *while* há um ponto-e-vírgula (;), que indica término da descrição da estrutura, porém não é permitido a construção de blocos de repetição sem a delimitação de blocos. A sintaxe associada ao *do while* blocular é apresentada no código 54 e seu fluxograma na figura 5.13.

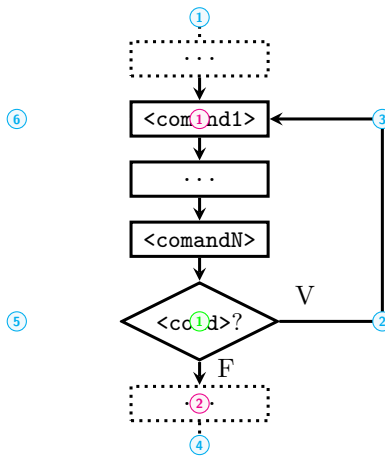


Figura 5.13: Fluxograma de *do while* blocular

```

1 //...
2 do
3 {
4   <comand1>;
5   //...
6   <comandN>;
7 }
8 while(<cond>);
9 //...

```

Código 54: Estrutura de *do while* blocular

Um uso comum para o **do while** é a construção de menus, já que as opções sempre serão exibidas ao menos uma vez, e então alguma opção é selecionada, saindo da repetição.

5.2.3 O *for* e a estrutura de repetição definida

Existem casos onde repetições tem quantidades exatas de ocorrências, e mesmo nesses casos não vale a pena fazer cópias do código. Pode-se sempre criar estruturas definidas com **while**, utilizando um contador de voltas, como no código 55, onde há uma variável *i* do tipo **int** iniciando em 0. Dentro da estrutura do **while** é necessária uma alteração na varável *i* para que a repetição termine.

O termo utilizado para descrever cada repetição de uma sequência de código é *iteração* (não confundir com interação). Normalmente utiliza-se uma variável que controla o número da iteração e comumente seu nome é dado pela letra *i*, referenciando seu significado de *iteração*. É comum iniciar variáveis de iteração com o valor nulo (0), a explicação para isso será dada no capítulo 8.

```

1 //...
2 int i(0);
3 while(i<10) //Quantas iterações acontecem aqui?
4 {
5   //...
6   i++;
7 }
8 //...

```

Código 55: Repetição com número de iterações definida utilizando *while*

Nem sempre é confortável criar essa estrutura toda vez que uma repetição definida é utilizada no código, por isso existe um controlador de fluxo desenvolvido especialmente para situações como esta.

A palavra-chave **for** define uma estrutura onde o valor de início, a condição de continuação e o passo de iteração são colocados como argumentos. Um exemplo da sintaxe linear é apresentado no código 56 e de sintaxe blocular no código 57.

```

1 //...
2 for(<inic>; <cond>; <passo>) <comand>;
3 //...

```

Código 56: Estrutura de *for* linear

```

1 //...
2 for(<inic>; <cond>; <passo>)
3 {
4     <comand1>;
5     //...
6     <comandN>;
7 }
8 //...

```

Código 57: Estrutura de *for* blocular

Existem algumas facilidades no uso do *for* que fazem o programador preferir seu uso:

- A declaração da variável de controle pode ser feita na região de inicialização;
- Mais de uma variável pode ser criada e utilizada (todas do mesmo tipo);
- As variáveis de iteração podem ser utilizadas dentro da estrutura de repetição;
- A condição de continuação não precisa ser atrelada as variáveis criadas;
- O passo de iteração pode alterar todas as variáveis criadas de forma independente;
- Não é necessário inicializar a variável de iteração se esta já vem com um valor estabelecido (ou seja, que não é lixo de memória);
- O passo de iteração não precisa alterar somente as variáveis de iteração;
- As variáveis declaradas na região de inicialização são locais para o comando (ou bloco) da repetição.

Alguns destes itens podem não fazer sentido num primeiro momento, mas ficarão mais claros com o exemplo do código 58

```

1 //...
2 int acc(0);
3 for (int i(0); i < 10; i++) acc += i; //Soma todos os números de 0 a 9
4
5 int A(13);
6 int B(-13);
7 for (int i(A); i > 0; i--) B += 1;
8
9 for (; A > B; B++) ; //Note como o ponto-e-vírgula está sozinho, ou seja, o
10 //comando é vazio, porém B é incrementado
11
12 int sum(0);
13 for (int i(0), j(0); i < 10; i++) for (int k(0); k < 10; k++, j++) sum += i
14 * j; //Aninhamento
15
16 long prod(1);
17 for (A = 25, B = 50; A < B; A++, B -= 2)
18 {
19     prod *= B;
20     prod /= A;
21 }
22 //...

```

Código 58: Exemplos de *for*

Vale notar uma peculiaridade do `for`, os três argumentos são deixados vazios a repetição de torna infinita, semelhante ao `while` com `true` de argumento. Repetições infinitas são úteis para programas que devem repetir sua estrutura principal muitas vezes. Porém, desta maneira, os programas não finalizam, por isso existem controles de fluxo especialmente criados para escapar de repetições.

5.2.4 Mudanças bruscas em repetições

Em várias situações se faz necessária a saída brusca em uma estrutura de repetição, ou talvez apenas ignorar a execução de uma das iterações, e, para ambas as situações existem palavras-chave que realizam tal alteração no fluxo. Para o caso onde a saída do fluxo de repetição, utiliza-se o `break`, que quebra a estrutura, saindo da repetição independente da condição de parada, semelhante ao caso do `switch`. Para o caso onde apenas uma iteração deve ser ignorada, utiliza-se o `continue`, que pula todos os subsequentes comandos da iteração, indo à próxima verificação de continuação.

Suas sintaxes são apresentadas no código 59, e um exemplo de uso é apresentado no código 60

```
1 //...
2 continue; //Normalmente são utilizados com um if
3
4 break;
5 //...
```

Código 59: Sintaxe de `continue` e `break`

Por alterarem o fluxo sempre que encontrados, normalmente estão associados a uma estrutura de decisão, como o `if`. Sem esta estrutura de decisão, valeria mais a pena rever as condições de parada, pois há algum problema no desenvolvimento do código.

```
1 //...
2 int A(100), B(5);
3 //...
4 while (A > -50)
5 {
6     if (B == 0)
7     {
8         B -= 1;
9         continue;
10    }
11    A /= B; //Note como a necessidade do else é suprida
12    if (A == B) break;
13    A += B - 5;
14 }
15 //...
```

Código 60: Exemplo de `continue` e `break`

Alterações adicionais no fluxo são apresentadas no capítulo 6.

Parte II

Intermediário

VERSÃO PRELIMINAR

Capítulo 6

Procedimentos e funções

Ao longo do desenvolvimento de programas, não é difícil notar que alguns conjuntos de comandos se repetem ao longo do código. Comandos usuais como exibir valores na tela, ler entradas do usuário, fazer conjuntos de contas ou validar valores lidos costumam ocorrer em muitos pontos do programa. Assim como no caso das estruturas de repetição, não vale a pena copiar blocos de código pelos mesmos fatores, como alterações posteriores, busca de erros, etc.

Para evitar a necessidade de cópias desnecessárias, melhorar a legibilidade, facilitar a busca por erros e a manutenção posterior do código foram desenvolvidos os procedimentos e as funções. Eles consistem em blocos de código que recebem nomes próprios (assim como variáveis), e podem ser invocados em qualquer parte do programa.

Procedimentos e funções devem ser declarados de maneira global e antes de serem invocados. Podem interagir com variáveis globais e locais (de seus blocos). Diferente dos controles de fluxo, necessariamente há um bloco de código associado.

Os mais simples são os procedimentos sem argumentos, sem valores de retorno, sem características especiais. Simplesmente são blocos de código com nome próprio que podem ser invocados.

6.1 Procedimentos

Sua estrutura é tão simples que a apresentação da sintaxe direta serve para entender sua declaração. Lembrando que todo bloco pode contar com controladores de fluxo. O código 61 apresenta a sintaxe básica de um procedimento chamado `foo`.

Os principais detalhes são: o tipo `void`, a declaração global e os parênteses `()` após o nome. Estes parênteses definem a declaração tipo `void` como procedimento.

```
1 void foo(void)
2 {
3     <comand1>;
4     //...
5     <comandN>;
6 }
```

Código 61: Sintaxe básica de procedimento sem argumentos

O código 62 mostra um programa completo que calcula o MDC entre duas variáveis utilizando um procedimento chamado `MDCab`. Note como os valores das variáveis são alterados pelo procedimento, isso pode gerar problemas no código.

```
1 int A, B;
2
3 void MDCab(void)
4 {
5     while (B != 0)
6     {
7         int R(A%B);
8         A=B;
```



```

9      B=R;
10    }
11  }
12
13  int main()
14  {
15      A = 93;
16      B = 45;
17      MDCab();
18      int C(A); //C vale 3
19      return 0;
20  }

```

Código 62: Exemplo de procedimento sem argumentos

Vale ressaltar que, da maneira que está escrito, o algoritmo de Euclides depende que a variável A seja a de maior valor. Procedimentos que recebem argumentos facilitam situações como estas.

6.2 Passagem de argumentos

Procedimentos podem receber valores de variáveis para seus blocos, através de variáveis auxiliares, que copiam os valores passados aos argumentos. Tais variáveis são ditas os *parâmetros* do procedimento (ou da função). Estas variáveis devem ter o tipo definido, e os argumentos devem ser passados conforme o tipo. Múltiplos argumentos de múltiplos tipos podem ser utilizado por procedimento, porém a ordem de passagem de argumentos deve ser respeitada.

Os argumentos são como variáveis declaradas entre os parênteses do procedimento, que podem ser utilizadas em seu corpo. Estas variáveis receberão os valores passados na invocação do procedimento durante o código principal. Cada argumento precisa ser separado por vírgulas, inclusive seus tipos. A sintaxe geral é apresentada no código 63.

```

1  void foo(<tipo1> <var1>, ... , <tipoN> <varN>)
2  {
3      <comand1>;
4      //...
5      <comandM>;
6  }

```

Código 63: Sintaxe básica de procedimento com N argumentos

Lembrando sempre: respeitar a ordem de declaração, o código 64 apresenta o mesmo algoritmo de Euclides, num procedimento com passagem de argumentos.

```

1  int C(0);
2
3  void MDCab(int A, int B)
4  {
5      while(B!=0)
6      {
7          int R(A%B);
8          A=B;
9          B=R;
10     }
11     C = A;
12 }
13
14 int main()
15 {
16     int A(93);
17     int B(45);

```

```

18 | MDCab(A,B);
19 | //C vale 3
20 | return 0;
21 | }

```

Código 64: Exemplo de procedimento com argumentos

Note que, mesmo com a passagem de argumentos para o procedimento, o retorno ainda é feito por uma variável global, o que pode ser inconveniente em muitos casos. Se o procedimento pudesse ser usado como variável, seria muito mais simples.

Isso é possível, e o procedimento que devolve um valor é chamado de *função*.

6.3 Funções

Assim como um procedimento, uma função pode ser invocada em qualquer lugar do código, a diferença é que neste lugar haverá o retorno de um valor (como os operadores, que retornam o valor da operação), então o valor retornado pode ser utilizado em uma operação, assim como uma variável. A função é declarada utilizando o tipo de seu retorno, então se uma função deve retornar um valor tipo `int`, deve ser declarada como `int`. Basicamente, a função é um procedimento que retorna um valor.

Assim como o procedimento, a função tem um nome próprio, e pode receber argumentos. A sintaxe geral de declaração de uma função com argumento é apresentada no código 63. Lembrando que, assim como no procedimento, os argumentos são opcionais.

O detalhe importante sobre a função é a necessidade de um retorno: toda função deve apresentar a palavra-chave `return`, seguida do valor a ser retornado. Vale notar que o `return` pode ser encarado como um controle de fluxo, uma vez que ele retorna da função o valor dado, independente do lugar que for encontrado pelo fluxo, saindo da função.

```

1 | <tipo> foo(<tipo1> <var1>, ... , <tipoN> <varN>)
2 | {
3 |     <comand1>;
4 |     //...
5 |     <comandM>;
6 |     return <retorno>;
7 | }

```

Código 65: Sintaxe básica de função com N argumentos

A mesma implementação do algoritmo de Euclides, porém utilizando uma função do tipo `int` é apresentada no código 66

```

1 | int MDCab(int primeiro, int segundo)
2 | {
3 |     while(segundo!=0)
4 |     {
5 |         int resto(primeiro%segundo);
6 |         primeiro=segundo;
7 |         segundo=resto;
8 |     }
9 |     return primeiro;
10 | }
11 |
12 | int main()
13 | {
14 |     int A(93);
15 |     int B(45);
16 |     int C(MDCab(A,B));
17 |     return 0;
18 | }

```

Código 66: Exemplo de função com argumentos

Vale notar que a função `main` é do tipo `int`, e tem o retorno em 0, este valor informa ao sistema operacional que não houveram erros do programa.

6.4 Polimorfismo

Além da possibilidade de receber valores no local de invocação e devolver um valor, há outra possibilidade interessante para procedimentos e funções em C++. É possível criar procedimentos que utilizem o mesmo nome, porém façam coisas diferentes. O termo *polímorfo* se refere ao procedimento que apresenta apenas um nome, porém muitas possibilidades de processamento, ou seja, mais de uma forma.

Procedimentos polimorficos podem ser utilizados para diversas situações, onde um nome é ideal para descrever um conjunto de comandos, porém existe mais de uma forma de construir o bloco de comandos. Num primeiro momento, pode parecer confuso, porém o principal uso deste artifício está na facilidade no entendimento do código. Um exemplo é apresentado no código 67, onde duas funções diferentes fazem coisas semelhantes.

```
1 double mediaDois(double A, double B)
2 {
3     return (A+B)/2.0;
4 }
5
6 double mediaTres(double A, double B, double C)
7 {
8     return (A+B+C)/3.0;
9 }
10
11 double mediaQuatro(double A, double B, double C, double D)
12 {
13     return (A+B+C+D)/4.0;
14 }
```

Código 67: Funções com objetivos equivalentes

Note como as funções realizam operações equivalentes, porém os comandos internos não são iguais. Exatamente neste caso utiliza-se o polimorfismo, é possível criar duas funções que tenham o mesmo nome, porém recebam uma quantidade diferente de argumentos. Características como quantidade de argumentos e tipo de retorno definem qual destas funções o compilador escolherá na hora da compilação. O código 68 apresenta o polimorfismo de uma função de média para duas, três e quatro argumentos.

```
1 double media(double A, double B)
2 {
3     return (A+B)/2.0;
4 }
5
6 double media(double A, double B, double C)
7 {
8     return (A+B+C)/3.0;
9 }
10
11 double media(double A, double B, double C, double D)
12 {
13     return (A+B+C+D)/4.0;
14 }
```

Código 68: Funções com nomes iguais e objetivos equivalentes

Note como o bloco das funções não sofreu qualquer alteração, apenas o nome foi alterado, de tal forma que o significado associado a ele não mude, porém a capacidade de operar cresça. Neste caso, a escolha é feita pela quantidade de argumentos passados na invocação.

O conceito de polimorfismo será abordado novamente no capítulo 12.

6.5 Recursão

O corpo principal do programa existe dentro da função `main`, e todo o comando e invocação de qualquer procedimento pode ser feita lá. Seguindo esta lógica, quaisquer procedimentos podem ser invocados dentro de quaisquer procedimentos, isso significa que é possível um procedimento que invoque a si próprio, e a este artifício é dado o nome *recursão*.

Porém tal característica pode criar uma repetição indefinida, sem condição de parada, caso um procedimento sempre invoque a si, para evitar isso todo procedimento que utiliza recursão apresenta um controle de fluxo que obriga a saída da recursão. Esta condição é associada a uma simplificação de um problema maior.

A técnica de recursão consiste em tratar um problema complicado dividindo-o em partes menores e mais simples, através de um *passo de recursão*, e a cada uma destas em partes menores, sucessivamente, até que se atinja o dito *caso fundamental*, onde o problema é pequeno o bastante para ser facilmente resolvido, a esta situação é dado o nome de *caso trivial*. Com os problemas triviais resolvidos, é mais fácil combinar suas soluções e encontrar a solução ao problema mais complicado associado.

O exemplo mais conhecido de função recursiva é o fatorial, que consiste na multiplicação de um número por todos os antecessores até o número 1, descrito na equação 6.1, e generalizada na equação 6.2.

$$f(n) = n! \equiv n \cdot (n-1) \cdot (n-2) \cdot \dots \cdot 2 \cdot 1 \quad (6.1)$$

$$n! = \begin{cases} n \leq 1 & \Rightarrow 1 \\ n \not\leq 1 & \Rightarrow n \cdot (n-1)! \end{cases} \quad (6.2)$$

$$f(n) = n! = n \cdot (n-1)! = n \cdot f(n-1) \quad (6.3)$$

Neste caso o passo de recursão é repetir a função para o valor subtraído em 1 ($n-1$), o caso trivial é definido com o valor igual a 1 ou 0 ($n=1$ ou $n=0$), onde a função devolve o valor 1 e a combinação dos resultados consiste em retornar a multiplicação do valor recebido como argumento pelo retornado pela recursão.

A transformação da equação 6.2 em uma função é feita de maneira direta, e esta função é apresentada no código 69, equivalente a equação 6.3.

```
1 unsigned int fatorial(unsigned int N)
2 {
3     if(N<=1) return 1;
4     else return N*fatorial(N-1);
5 }
```

Código 69: Função recursiva

Capítulo 7

Mais operadores

Em um primeiro momento, é interessante conhecer os operadores que são mais utilizados. A partir desse ponto se torna interessante conhecer os operadores de alteração profunda em variáveis, que permitem comandos mais interessantes.

7.1 Modelador

É possível inicializar variáveis utilizando literais, mas nem sempre os tipos certos são utilizados. Por exemplo, se declara uma variável tipo `float` e a inicializa com o valor nulo (0), é comum utilizar apenas 0, ou até 0.0, que é menos comum. Idealmente deve-se inicializar uma variável tipo `float` com um literal tipo `float`, no caso, 0.0f.

Quando não se tem a certeza do tipo de literal utilizado, ou ainda, é necessário utilizar o valor de uma variável de um tipo na inicialização de uma variável de outro tipo, utiliza-se o conceito de *casting*. O casting consiste em converter um tipo em outro de maneira explícita, diferente do caso anterior, onde a conversão era implícita.

Também é um operador, portanto tem uma posição da ordem de precedência, onde fica abaixo dos operadores de negação. Esse operador não altera o valor salvo na variável, nem altera o tipo dela, apenas retorna um valor do tipo escolhido.

O apêndice B apresenta a tabela completa de precedência de operadores. Existem duas sintaxes para casting explícito, são semelhantes e podem ser utilizadas juntas, ambas apresentadas no código 70.

```
1 //...
2 <tipo>(<valor>);
3 (<tipo>)<valor>;
4 //...
```

Código 70: Sintaxes de casting explícito

O casting implícito acontece quando utilizamos o operador de atribuição (=), e seus variantes, com valores de tipos diferentes. A maneira mais direta de pensar nisso parte da atribuição de um valor do tipo `float` em uma variável do tipo `int`, como o segundo apenas pode armazenar valores inteiros, a parte decimal do primeiro é ignorada. A esta operação é dado o nome de *truncamento*. Note que não é feito um arredondamento, mesmo que o valor atribuído possua uma dízima como 0.9 (0.999...).

Um exemplo útil do uso de casting é apresentado no código 71, onde uma função tipo `bool` retorna `true` se o argumento de entrada for um número natural.

```
1 //...
2 bool ehNaturalI(float N)
3 {
4     int valor(N);           //Casting implícito
5     float M(valor);
6     return M == N;
7 }
8
9 bool ehNaturalE(float N)
10 {
```

```

11 |     return N == float(int(N)); //Casting explícito
12 | }
13 | //...

```

Código 71: Exemplo do uso de casting

7.2 Indexador

Quando trabalha-se com sequências de memórias, apresentadas no capítulo 8, é necessário realizar a escolha de qual os espaços de memória será utilizado, para isso, utiliza-se o operador de indexação. É um operador que recebe dois argumentos, sendo o primeiro o nome da variável indexada e o segundo o valor da indexação.

É um operador que permite acesso completo, de tal maneira que seu retorno pode ser tratado como uma variável do tipo da sequência de memória, permitindo leitura e escrita, além dos demais operadores.

Seu símbolo consiste um par de colchetes (`[]`) seguindo o nome da sequência de variáveis. Entre os colchetes, o índice da indexação. No C++, a indexação começa sempre em 0, explicações são dadas no capítulo 8.

O código 72 apresenta a sintaxe de uso do operador. Outro tipo de uso, além de exemplos são apresentados no capítulo 8.

```

1 | //...
2 | <nome>[<índice>];
3 | //...

```

Código 72: Sintaxe do operador de indexação

Vale lembrar que, assim como os demais operadores, o uso do indexador é incoerente sozinho, fazendo-se útil apenas quando implica em diferença no programa, ou seja, quando alguma alteração é feita.

7.3 Tamanho em memória

Ao longo do desenvolvimento de software, pode se fazer necessário saber o tamanho de espaços de memória, seja o espaço ocupado por variáveis, seja o ocupado por vetores (capítulo 8). É possível criar novas variáveis apenas para armazenar estas informações, porém isso gastaria mais memória, o que pode ser infortúnio em muitos casos.

O operador `sizeof` retorna o tamanho, em *bytes*, da variável, vetor ou tipo passado como argumento. Sua sintaxe é semelhante a de uma função, e é apresentada no código 73.

```

1 | //...
2 | sizeof(<nome>);
3 | sizeof(<tipo>);
4 | //...

```

Código 73: Sintaxe do operador de indexação

Este operador não retorna o tamanho alocado em memória dinâmica (capítulo 11), também não funciona com ponteiros (capítulo 10).

7.4 Acesso a membro

Ao trabalhar com tipos abstratos ou classes, descritos respectivamente nos capítulos 9 e 12, é necessário acessar os membros internos para trabalhar com eles, para realizar este acesso, utiliza-se um operador próprio. O símbolo utilizado é o ponto (`.`).

Primeiro o nome da variável em questão, seguido do operador, então o membro a ser acessado. O código 74 apresenta a sintaxe do operador de acesso a membro. Exemplos de uso são apresentados nos capítulos 9 e 12.

```

1 | //...
2 | <nome>.<membro>; //Caso de variável membro
3 | <nome>.<membro>(); //Caso de função membro
4 | //...

```

Código 74: Sintaxe do operador de acesso a membro

7.5 Endereço

Existem situações onde se faz necessária a passagem de variáveis para procedimentos de tal forma as alterações lá feitas, sejam efetivas no restante do programa. Para isso, utiliza-se ponteiros (capítulo 10), além dos operadores de endereço e acesso indireto.

O acesso é feito utilizando o endereço de memória do espaço que a variável ocupa, o operador em questão retorna este endereço. Seu símbolo é o *e comencial* (&). Este operador também é chamado de *referenciador*.

Um exemplo de sintaxe é apresentado no código 75. Valor notar que o uso só é coerente quando seu retorno é aproveitado.

```
1 //...
2 &<nome>;
3 //...
```

Código 75: Sintaxe do operador endereço

Um uso peculiar e muito útil para esse operador é encontrado no capítulo 13.

7.6 Acesso indireto

Também chamado de *derreferenciador*, realiza a operação inversa ao operador de endereço (*referenciador*), acessando o armazenado em memória através de seu endereço. Também utilizado com ponteiros (capítulo 10).

Seu uso, como no indexador, permite acesso completo. Inclusive, seu uso equivale ao uso do indexador com índice zero ([0]), sem símbolo é o asterisco (*). Sua sintaxe é apresentada no código 76. Lembrando que o uso só é coerente se alguma alteração no programa é feita.

```
1 //...
2 *<nome>;
3 //...
```

Código 76: Sintaxe do operador endereço

7.7 Acesso indireto de membro

Existem casos onde ponteiros para tipos abstratos são necessários (capítulos 10 e 9), e para acessar os membros internos se faz necessário o uso do operador de acesso indireto, além do operador de acesso a membro, conforme a sintaxe apresentada no código 77.

```
1 //...
2 (*<nome>).<membro>; //Caso de variável membro
3 (*<nome>).<membro>(); //Caso de função membro
4 //...
```

Código 77: Exemplo de acesso a membro de forma indireta sem operador apropriado

Por ser uma estrutura ambígua e que depende de alterações na ordem de precedência, utilizando os parênteses (()), foi criado um operador que realiza o acesso indireto a membros. Seu símbolo é composto por um sinal de menos, seguido por um sinal de menor (->), se assemelha a uma seta. Este operador é especializado para realizar o acesso de membros internos através de ponteiros. Em situações como estruturas de dados, é amplamente utilizados, a sintaxe associada é apresentada no código 78.

```
1 //...
2 <nome>-><membro>; //Caso de variável membro
3 <nome>-><membro>(); //Caso de função membro
4 //...
```

Código 78: Exemplo de acesso a membro de forma indireta sem operador apropriado

7.8 *Alocador de memória

...

7.9 *Dealocador de memória

...

VERSÃO PRELIMINAR

Capítulo 8

Sequências de memórias

Em muitos casos, programas recebem dezenas de valores como entrada, processam todos esses valores com uma metodologia repetitiva e então devolvem todos na ordem recebida. É possível criar dezenas de variáveis para fazer a leitura, criar uma função para este processamento repetitivo e devolver todos estes valores. Cem variáveis são criadas no código 79, porém há uma que não segue o padrão das demais, o que tende a gerar erros ao longo do desenvolvimento do programa.

```
1 int V00(0), V01(0), V02(0), V03(0), V04(0), V05(0), V06(0), V07(0), V08(0), V09(0);
2 int V10(0), V11(0), V12(0), V13(0), V14(0), V15(0), V16(0), V17(0), V18(0), V19(0);
3 int V20(0), V21(0), V22(0), V23(0), V24(0), V25(0), V26(0), V27(0), V28(0), V29(0);
4 int V30(0), V31(0), V32(0), V33(0), V34(0), V35(0), V36(0), V37(0), V38(0), V39(0);
5 int V40(0), V41(0), V42(0), V43(0), V44(0), V45(0), V46(0), V47(0), V48(0), V49(0);
6 int V50(0), V51(0), V52(0), V53(0), V54(0), V55(0), V56(0), V57(0), V58(0), V59(0);
7 int V60(0), V61(0), V62(0), V63(0), V64(0), V65(0), V66(0), V67(0), V68(0), V69(0);
8 int V70(0), V71(0), V72(0), V73(0), V74(0), V75(0), V76(0), V77(0), V78(0), V79(0);
9 int V80(0), V81(0), V82(0), V83(0), V84(0), V85(0), V86(0), V87(0), V88(0), V89(0);
10 int V90(0), V91(0), V92(0), V93(0), V94(0), V95(0), V96(0), V97(0), V98(0), V99(0);
```

Código 79: Declarando dezenas de variáveis

Além de digitar cem variáveis ser uma tarefa cansativa, também é necessário digitar um processamento em relação a cada uma delas. Como seus identificadores (nomes) são distintos, uma estrutura de repetição não serve de muito, já que são processamentos correspondentes, porém o código muda em cada iteração.

Existe uma forma de agrupamento de variáveis especial criada para situações como esta: o vetor.

8.1 *Vetores

O termo *vetor* deriva da matemática, onde vetores podem ser representados conjuntos de números sequenciados, referentes a uma base. Para o caso aqui descrito, a ideia de valores sequenciados é o bastante. Vetores são variáveis declaradas de forma sequencial na memória, acessadas a partir de um identificador (nome) comum para todas, diferindo em um índice de acesso.

A sintaxe de declaração de um vetor é semelhante a de uma variável normal, tendo um tipo e um nome, acrescentando apenas a quantidade de valores salvos entre colchetes ([]) antes do ponto-e-vírgula, conforme o código 80.

```
1 //...
2 <tipo> <nome>[<tamanho>];
3 //...
```

Código 80: Declarando vetor com n elementos

A inicialização de um vetor é diferente da inicialização de uma variável comum, já que possui vários valores associados, três casos de inicialização podem ser encontrados no código 81. O acesso ao valor do vetor também utiliza os colchetes ([]) como artifício, passando a eles o índice utilizado pelo valor salvo no vetor. É muito importante ressaltar que o primeiro valor possui o índice 0, logo o último tem o índice $n - 1$, então o **for** do código de inicialização tem como seus limites estes valores. A tabela 8.1 exibe este detalhe.

Tabela 8.1: Representação de vetor de n valores

0	1	2	3	...	$n-1$

```

1 //...
2 <tipo> <nome>[<tam>]{<val1>, ..., <valn>; //Sintaxe de Inicialização
3 //...
4 int V1[5]{}; //Todos os valores como 0
5 int V2[5]{1,2,3,4,5}; //Valores conforme sequência
6 int V3[] {1,2,3,4,5}; //Tamanho do vetor implícito
7 //...
8 for (int i(0); i<5; i++) V1[i]=V2[i]+V3[i]; //Inicialização póstera
9 //...

```

Código 81: Inicializações para vetores

O código 82 apresenta um caso especialmente útil do uso de vetores, para calcular a média entre vários valores, todos salvos em um único vetor. É o típico caso onde vários valores são lidos e então processados de forma equivalente.

```

1 //...
2 float valores[10];
3 //...
4 float media(0.0f);
5 for(int i=0; i<10; i++) media +=valores[i];
6 media /= 10;
7 //...

```

Código 82: Exemplo de uso para vetores calculando média

Assim como é possível criar vetores de variáveis, é possível criar vetores de vetores, e a estes damos o nome de matriz.

8.2 *Matrizes

Um vetor é como uma matriz de uma dimensão. A declaração de uma matriz se assemelha a de um vetor, diferindo apenas na quantidade de dimensões passadas. É importante ressaltar que cada dimensão é passada dentro de um colchete próprio. A sintaxe de declaração de uma matriz é apresentada no código 83.

```

1 //...
2 <tipo> <nome>[<tamanho1>][<tamanho2>;
3 //...

```

Código 83: Declarando matriz $m \times n$

Assim como no caso de um vetor, os valores salvos em uma matriz também são acessados utilizando um índice. E ainda como no caso dos vetores, o primeiro valor é 0. A tabela 8.2 apresenta este detalhe.

Tabela 8.2: Representação de matriz de $m \times n$

	0	1	2	3	...	$n-1$
0						
1						
2						
3						
⋮						
$m-1$						

As formas de inicialização de uma matriz são semelhantes às formas de inicialização de um vetor e são apresentadas no código 84.

```

1 //...
2 <tipo> <nome>[<tam1>][<tam2>]{<val11>, ..., <val1n>}, ..., {<valm1>, ..., <
  valmn>}}; //Sintaxe de Inicialização
3 //...
4 int V1[2][3]{}; //Todos os valores como 0
5 int V2[2][3]{{1,2,3},{4,5,6}}; //Valores separados por vetor
6 int V3[2][3]{1,2,3,4,5,6}; //Valores distribuídos automaticamente
7 int V3[][2]{1,2,3,4,5,6}; //Distribuição com um tamanho implícito
8 //Atribuição só pode ser implícita para a primeira dimensão
9 //...
10 for(int i(0); i<2; i++) for(int j(0); j<2; j++) V1[i][j]=V2[i][j]+V3[i][j];
11 //Inicialização póstera
12 //...

```

Código 84: Inicializações para matrizes

Assim como no caso dos vetores, as matrizes tem grande uso para conjuntos de memórias, o código ?? apresenta um método para o cálculo da média de cada linha na matriz e atribuindo estas médias a um vetor de médias.

```

1 //...
2 float valores[8][12];
3 //...
4 float media[8]{};
5 for(int i=0; i<8; i++) for(int j=0; j<12; j++) media[i] +=valores[i][j];
6 for(int i=0; i<8; i++) media[i] /=8;
7 //...

```

Código 85: Exemplo de uso para matriz calculando média das linhas

É possível generalizar matrizes para mais do que duas dimensões, tais estruturas na matemática são denominadas *tensores*, porém não é necessária a abordagem especial deste tipo de estrutura já que o passo de passagem de um vetor para uma matriz é idêntico ao passo de passagem de uma matriz para um tensor. É interessante ressaltar que matrizes são tensores de ordem 2 (ou segunda ordem), e vetores são tensores de ordem 1 (ou primeira ordem).

Vale notar que o controle de fluxo `for` foi desenvolvido especialmente para esta aplicação.

8.3 Sequências de caracteres

Existe um caso especial de vetor utilizado para gravação de texto, não apenas caracteres como no caso de uma variável tipo `char`. São vetores comuns de caracteres, a grande diferença está em sua inicialização.

O termo *string* descreve o vetor de caracteres. A inicialização é feita como do código 87, onde a frase *Hello World!* é salva na forma de *string*. Note que o tamanho do vetor está implícito, isso tem um motivo.

```

1 //...
2 char fraseA[]("Hello World!");
3 char fraseB[]="Hello World";
4 //...

```

Código 86: Sintaxe inicialização

O tamanho implícito está associado a uma característica importante da declaração da string como literal. Quando utilizamos as aspas duplas como notação, é adicionado ao final um caractere especial, o caractere nulo ('0'). Este caractere é adicionado para possibilitar o processamento deste tipo de vetor sem que o tamanho seja conhecido, onde a condição de um `for` trabalhe verificando se o próximo valor armazenado seja o nulo. O tamanho do vetor não é passado pois seria trabalhoso lembrar deste caractere especial toda vez que um vetor desse tipo for declarado, logo a inicialização precisa ser feita na declaração.

O código 87 mostra uma forma de contar o tamanho de uma string, considerando apenas os caracteres não nulos. Note como no segundo caso, a contagem falha.

```

1 //...
2 char A[]("string");
3 char B[]{'s','t','r','i','n','g','\0'}; //Equivalente a A

```

```

4 char C[]{'s','t','r','i','n','g'};           //Não equivale a A
5 char D[]{'H','e','l','l','o','\0',' ','W','o','r','l','d','!'};
6 //Espaço em branco não é nulo
7 //...
8 int size(0);
9 for(int i=0; A[i]!='\0'; i++) size++;        //Contagem tamanho da string
10 size = 0;
11 for(int i=0; D[i]!='\0'; i++) size++;        //Note que a contagem falha
12 //...

```

Código 87: Exemplo de *string*

Capítulo 9

Tipos abstratos

9.1 *Enumeradores

...

9.2 *Estruturas

...

9.3 *Unões

...

Capítulo 10

Ponteiros

Toda vez que uma variável é declarada, um espaço de memória é reservado para ela, o nome utilizado nesta declaração será associado a um endereço da memória. O endereço de memória normalmente é exibido como um valor hexadecimal de 12 dígitos, como `0xfedcba987654`, e pode ser obtido com o operador de acesso (referenciador). O nome da variável é apenas um símbolo que referencia aquele espaço de memória, um símbolo que aponta para tal espaço.

É possível criar variáveis especiais especializadas em armazenar endereços, e utilizando elas, alterar os valores salvos nestes endereços de maneira indireta. Estas variáveis são chamadas *ponteiros*. Ponteiros apresentem os mesmos tipos que as variáveis normais.

Pode parecer estranho que um ponteiro precise de um tipo, já que o valor que armazena é um endereço, e todos os endereços de memória tem o mesmo tamanho, seu tipo define qual o tipo de variável será apontada, especificando quantos bytes de memória deve interagir quando utilizado.

10.1 Declarando ponteiros

Semelhante à declaração de variáveis normais, os ponteiros precisam de um tipo e um nome, que segue as mesmas regras que as demais variáveis.

A grande diferença entre suas declarações está na existência do operador de acesso indireto, indicando que a entidade a ser declarada é um ponteiro para o tipo definido. O código 88 apresenta a sintaxe de declaração de um ponteiro. Assim como variáveis, ponteiro respeitam escopo.

```
1 //...
2 <modificadores> <tipo> *<nome>;
3 //...
```

Código 88: Declaração de ponteiro

Exemplos de ponteiros são apresentados na seção 10.6.

10.2 Inicializando ponteiros

A inicialização de um ponteiro é semelhante à de uma variável em questão de sintaxe, a principal diferença está na passagem do valor, que não pode ser de maneira direta, já que ponteiros armazenam endereços de memória. Logo, a inicialização de um ponteiro recebe o endereço de uma variável, com o operador de endereço. A inicialização também pode ser feita utilizando outro ponteiro como valor.

Lembrando que um ponteiro deve apontar para uma variável do mesmo tipo que sua própria declaração. O código 89 apresenta a sintaxe de inicialização de um ponteiro.

```
1 //...
2 <modificadores> <tipo> <var>;
3 //...
4 <modificadores> <tipo> *<nome>(&<var>);
5 //...
```

Código 89: Inicialização de ponteiro

10.3 Ponteiros de vetores

Vetores são variáveis especiais, que tem associado o endereço de memória de um espaço equivalente a N vezes o tamanho padrão do seu tipo, ou seja, um vetor `int V[10]` ocupa um espaço de 10 vezes o tamanho padrão de um `int`.

Assim como a variável normal, vetores tem um endereço próprio que pode ser obtido com o operador de endereço, mas vetores apresentam a característica de trabalharem como um vetor, porém não podem alterar o endereço apontado. Quando o objetivo é passar o endereço de um vetor para um ponteiro, o uso do operador é suprido, pois o vetor já trabalha como um ponteiro, então já armazena um endereço, o endereço do bloco de memória reservado para ele.

A passagem do endereço de um vetor para um ponteiro é igual a passagem do valor de um ponteiro para outro. Exemplos de ponteiros para vetores são apresentados na seção 10.6.

10.4 Ponteiros de ponteiros

Assim como qualquer outra variável, ponteiros tem endereços próprios, e é possível criar ponteiros para ponteiros. Ponteiros de ponteiros também apresentam tipo definido, o mesmo tipo que o apontado aponta, ou seja, um ponteiro para ponteiro do tipo `int` é do tipo `int` também. Ponteiros de ponteiros são muito utilizados quando se trabalha com memória dinâmica (capítulo 11), para a criação de matrizes.

Exemplos do uso de ponteiro de ponteiros é apresentado na seção 10.6.

10.5 Aritimética de ponteiros

Quanto utilizando o operador de acesso, o ponteiro se comporta exatamente como uma variável do seu tipo. Porém existe um pequeno conjunto de operadores que fazem alterações especiais em ponteiros.

Os operadores de incremento e decremento unitário fazer a operação de avanço e retrocesso no endereço de memória, as diferenças nos operadores, utilizados como prefixo e posfixo, é a mesma do caso de variáveis normais. O uso destes operadores apenas é coerente com o uso de ponteiros para vetores ou outros ponteiros. Nos demais casos, ponteiros estarão apontando para variáveis simples.

Também é possível utilizar os operadores de soma e subtração, que trabalharão como os incrementadores, alterando o endereço salvo no vetor. Toda alteração é proporcional, ou seja, somar 2 em um vetor implica em apontar para o espaço de memória de duas vezes o tamanho de memória do tipo do vetor, no caso de um ponteiro de `int`, com 4 bytes, implica em apontar para um endereço maior em oito unidades do original. Cada unidade alterado no valor (hexadecimal) armazenado em um ponteiro implica na alteração de um byte no espaço de memória.

Existem um problema associado ao uso destes operadores. Quando utilizados, alteram o valor do endereço armazenado no ponteiro. Se não é possível recuperar o endereço do espaço de memória, ele ficará alocado até o fim da execução do programa, mais detalhes serão apresentados no capítulo 11.

Para evitar este problema, é possível utilizar o operador de indexação, apresentado no capítulo 7. Este operador fará o acesso do valor salvo na memória através de um índice, e não causará a perda do endereço do ponteiro. A desvantagem deste operador em relação ao caso do incrementador está na necessidade de uma variável como indexador, ou seja, uma variável para ser incrementada ao longo da passagem pelo vetor.

10.6 Exemplos de ponteiros

A diversidade de situações onde ponteiros são úteis ou necessários faz com que apenas um exemplo seja insuficiente. O código 90 mostra casos de declaração e inicialização de ponteiros.

```
1 //...
2 bool falso(0);           //Booleano
3 char letraA('A');        //Caractere
4 int cont(1), *Pcont(&cont); //Inteiro, e ponteiro para inteiro
5 float dados[50]{};       //Vetor de flutuantes
6 float *Pdados(dados);    //Ponteiro para vetor
7 char *PletraA(&letraA);  //Apontando para a variável lateraA
8 bool *Pfalso(&falso);    //Apontando para a variável falso
```

```

9 | int *P2cont(Pcont);           //Apontando para o mesmo que o ponteiro Pcont
10| //...

```

Código 90: Exemplo de declaração e inicialização

O código 91 apresenta uma função onde as variáveis são passadas por endereço, também chamada de passagem por referência.

```

1 | //...
2 | void MDC(int *primeiro, int *segundo) //Parâmetros são ponteiros
3 |     while(*segundo!=0)
4 |     {
5 |         int resto(*primeiro%*segundo);
6 |         *primeiro=*segundo;           //A memória é alterada
7 |         *segundo=resto;
8 |     }
9 | }
10| //...
11| int main()
12| {
13|     int alpha(234), beta(5493);
14|     MDC(&alpha, &beta);               //O endereço é o argumento
15|     int C(alpha);                     //C vale 3
16|     return 0;
17| }

```

Código 91: Exemplo função que utiliza ponteiros

O código 92 contém um caso de ponteiros como vetor, além do uso do operador de indexação com vetores.

```

1 | float mediaN1(int N, float *V)
2 | {
3 |     float acc = 0;
4 |     for(int i = 0; i<N; i++) acc += V[i];
5 |     return acc/float(N);
6 | }
7 | //...
8 | #define SIZE 50
9 | int main()
10| {
11|     float vetor[SIZE]{};
12|     float media;
13|     //... //Atribui valor ao vetor
14|     media = mediaN1(SIZE, vetor);
15|     //...
16|     return 0;
17| }

```

Código 92: Exemplo de ponteiro como vetor

O código 93 exemplifica o uso de acesso indireto a membro.

```

1 | struct meuTipo
2 | {
3 |     int valor1, valor2;
4 | };
5 | //...
6 | int somarMt(meuTipo *S)
7 | {
8 |     return S->valor1 + S->valor2;
9 | }

```



```

10 //...
11 int main()
12 {
13     meuTipo mem;
14     int soma;
15     mem.valor1 = 10;
16     mem.valor2 = 20;
17     //...           //Atribui valor ao vetor
18     soma = somarMt(&mem);
19     //...
20     return 0;
21 }

```

Código 93: Exemplo de acesso indireto a membro

O código 94 utiliza aritmética de ponteiros de diversas formas. Note que o endereço original permanece referenciado, mesmo que de maneira indireta.

```

1 float mediaN2(int N, float *V)
2 {
3     float acc = 0;
4     //Dê atenção aos argumentos do for a seguir
5     for(float *aux(V + N); V<aux; V++) acc += *V;
6     return acc/float(N);
7 }
8 //...
9 #define SIZE 50
10 int main()
11 {
12     float vetor[SIZE]{};
13     float media;
14     //...           //Atribui valor ao vetor
15     media = mediaN2(SIZE, vetor);
16     //...
17     return 0;
18 }

```

Código 94: Exemplo com aritmética de ponteiros

VERSÃO PRELIMINAR

Parte III
Avançado

Capítulo 11

Memória Dinâmica

VERSÃO PRELIMINAR

Capítulo 12

Classes e Objetos

VERSÃO PRELIMINAR

Capítulo 13

Mais sobre procedimentos e funções

VERSÃO PRELIMINAR

Lista de Figuras

5.1	Fluxograma de fluxo simples	25
5.2	Fluxograma de fluxo ambíguo	25
5.3	Fluxograma de <i>if</i> simples linear	26
5.4	Fluxograma de <i>if</i> simples blocular	26
5.5	Fluxograma de <i>if</i> complementares linear	27
5.6	Fluxograma de <i>if</i> complementares blocular	28
5.7	Fluxograma de <i>if</i> aninhado	29
5.8	Fluxograma de estrutura de decisão composta	31
5.9	Fluxograma de fluxo repetitivo	33
5.10	Fluxograma de <i>while</i> linear	34
5.11	Fluxograma de <i>while</i> blocular	34
5.12	Fluxograma de <i>do while</i> linear	35
5.13	Fluxograma de <i>do while</i> blocular	36

Códigos

1	Código mínimo	5
2	Exemplos de sintaxe no código mínimo	8
3	Declaração de variável	13
4	Declaração de variável	14
5	Declarações de variável	14
6	Declarações de variável	14
7	Sintaxe geral de operadores unários	15
8	Exemplo genérico de operadores unários	15
9	Sintaxe de incrementadores e decrementadores unitários	15
10	Exemplo de incrementadores e decrementadores unitários	16
11	Sintaxe dos sinalizadores aritméticos	16
12	Exemplo de sinalizadores aritméticos	16
13	Sintaxe da negação lógica	17
14	Exemplo de negação lógica	17
15	Sintaxe do complemento binário	17
16	Exemplo de complemento binário	17
17	Sintaxe geral para operadores binários	17
18	Exemplo genérico de operadores binários	18
19	Sintaxe do operador de atribuição	18
20	Exemplo do operador de atribuição	18
21	Sintaxe dos operadores aritméticos	18
22	Exemplo de operadores aritméticos	19
23	Sintaxe dos operadores de deslocamento	19
24	Exemplos dos operadores de deslocamento	20
25	Sintaxe dos operadores lógicos bitwise	20
26	Exemplos dos operadores lógicos bitwise	20
27	Sintaxe dos operadores de atribuição composta	21
28	Exemplos de operadores de atribuição composta	21
29	Sintaxe de operadores de comparação	22
30	Sintaxe de operadores de comparação	22
31	Sintaxe de operadores lógicos booleanos	22
32	Exemplos dos operadores lógicos booleanos	22
33	Sintaxe do operador ternário	23
34	Exemplo de operador ternário	23
35	Exemplo de operadores e precedência	24
36	Estrutura de <i>if</i> simples linear	25
37	Estrutura de <i>if</i> simples blocular	26
38	Estrutura de <i>if</i> Exemplo de utilização do <i>if</i>	26
39	Estrutura de <i>if</i> complementares linear	27
40	Estrutura de <i>if</i> complementares blocular	27
41	Estrutura de <i>if</i> aninhado	28
42	Exemplo de <i>if else</i> aninhado	29
43	Exemplo de <i>if</i> sequencial	29
44	Exemplo de <i>if else</i> aninhado	30
45	Exemplo de <i>if else</i> aninhado com blocos explicitamente delimitados	30

46	Estrutura composta formada por <i>if</i>	30
47	Estrutura composta formada por <i>switch</i> , <i>case</i> , <i>break</i> e <i>default</i>	31
48	Exemplo de <i>switch</i>	32
49	Algoritmo de Euclides sem estrutura de repetição	33
50	Estrutura de <i>while</i> linear	34
51	Estrutura de <i>while</i> blocular	34
52	Algoritmo de Euclides com estrutura repetição	35
53	Estrutura de <i>do while</i> linear	35
54	Estrutura de <i>do while</i> blocular	36
55	Repetição com número de iterações definida utilizando <i>while</i>	36
56	Estrutura de <i>for</i> linear	37
57	Estrutura de <i>for</i> blocular	37
58	Exemplos de <i>for</i>	37
59	Sintaxe de <i>continue</i> e <i>break</i>	38
60	Exemplo de <i>continue</i> e <i>break</i>	38
61	Sintaxe básica de procedimento sem argumentos	40
62	Exemplo de procedimento sem argumentos	40
63	Sintaxe básica de procedimento com <i>N</i> argumentos	41
64	Exemplo de procedimento com argumentos	41
65	Sintaxe básica de função com <i>N</i> argumentos	42
66	Exemplo de função com argumentos	42
67	Funções com objetivos equivalentes	43
68	Funções com nomes iguais e objetivos equivalentes	43
69	Função recursiva	44
70	Sintaxes de casting explícito	45
71	Exemplo do uso de casting	45
72	Sintaxe do operador de indexação	46
73	Sintaxe do operador de indexação	46
74	Sintaxe do operador de acesso a membro	46
75	Sintaxe do operador endereço	47
76	Sintaxe do operador endereço	47
77	Exemplo de acesso a membro de forma indireta sem operador apropriado	47
78	Exemplo de acesso a membro de forma indireta sem operador apropriado	47
79	Declarando dezenas de variáveis	49
80	Declarando vetor com <i>n</i> elementos	49
81	Inicializações para vetores	50
82	Exemplo de uso para vetores calculando média	50
83	Declarando matriz $m \times n$	50
84	Inicializações para matrizes	51
85	Exemplo de uso para matriz calculando média das linhas	51
86	Sintaxe inicialização	51
87	Exemplo de <i>string</i>	51
88	Declaração de ponteiro	54
89	Inicialização de ponteiro	54
90	Exemplo de declaração e inicialização	55
91	Exemplo função que utiliza ponteiros	56
92	Exemplo de ponteiro como vetor	56
93	Exemplo de acesso indireto a membro	56
94	Exemplo com aritimética de ponteiros	57

Referências Bibliográficas

- [1] A. Houaiss, M. Villar, F. de Mello Franco, and I. A. H. de Lexicografia, *Dicionário Houaiss da língua portuguesa*. Objetiva, 2009. [Online]. Available: <https://books.google.com.br/books?id=1LQKqAAACAAJ>

Apêndices

VERSÃO PRELIMINAR

Apêndice A

Tabela ASCII

A tabela A.1 apresenta a codificação ASCII estendida, dividida em duas partes, na superior os tipos padrão, na inferior os tipos especiais.

A divisão da tabela é devida à codificação em bytes com um bit de paridade, o que diminui pela metade a capacidade de codificação de um byte.

O método do bit de paridade consiste em um sistema de detecção de erros de transmissão de dados. Todo caractere possui um código binário, por exemplo, o caractere 'A', pela tabela ASCII, tem o código 0x41 ou 0b01000001. Utilizando o método do bit de paridade, acrescenta-se o bit necessário para que a soma de bits 1 seja par, que é o mesmo de trocar o primeiro bit do byte para o correspondente à operação de *ou exclusivo* entre os demais bits, que é a operação eletrônica realizada.

No caso de 'A', 0b01000001 não sofre alterações, porém no caso de 'C', 0b01000011 é alterado para 0b11000011. A tabela ASCII padrão tem metade da capacidade de codificação de caracteres por isso.

A interpretação da tabela é feita a partir das coordenadas do caractere escolhido, por exemplo, o caractere C está na linha 4X e na coluna x3, portanto seu código é 0x43, ou seja, o número 43 em hexadecimal, corresponde a 67 na base decimal.

Tabela A.1: Codificação ASCII estendida incompleta¹.

xX	x0	x1	x2	x3	x4	x5	x6	x7	x8	x9	xA	xB	xC	xD	xE	xF
0X	NULL	SOH	STX	ETX	EOT	ENQ	ACK	BEL	BS	HT	LF	VT	FF	CR	SO	SI
1X	DLE	DC1	DC2	DC3	DC4	NAK	SYN	ETB	CAN	EM	SUB	ESC	FS	GS	RS	US
2X		!	"	#	\$	%	&	'	()	*	+	,	-	.	/
3X	0	1	2	3	4	5	6	7	8	9	:	;	<	=	>	?
4X	@	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
5X	P	Q	R	S	T	U	V	W	X	Y	Z	[\]	^	_
6X	`	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o
7X	p	q	r	s	t	u	v	w	x	y	z	{		}	~	DEL
8X	Ç	ü	é	â	ä	à	ã	ç	ê	ë	è	ï	î	ì	Ä	Å
9X	É	æ	Æ	ô	ö	ò	û	ù	ÿ	Ö	Ü	ø	Ø	×		f
AX	á	í	ó	ú	ñ	Ñ	ª	º	¿	®	¬	½	¼	¡	¥	¬
BX						Á	Â	Ã	©	¶	¶	¶	¶	¶	¥	¬
CX																
DX			Ê	Ë	Ì	Í	Î	Ï								
EX	Ó	ß	Ô	Õ	Ö	×	Ù			Ú	Û	Ü	Ý	Þ		
FX		±					÷	,	°	²	³	´	µ	¶		nbsp

¹Problemas na codificação de caracteres impediram a renderização de alguns tipos, os espaços em branco na parte inferior são os destes tipos. Uma busca na internet pode localizar facilmente uma tabela completa.

Apêndice B

Ordem de precedência

A ordem de precedência dos operadores segue a tabela B.1.

Tabela B.1: Ordem de precedência de operadores

Operador	Descrição
()	preferencial
++, --	posfixo
type(var)	modelador
array[i]	indexador
foo()	invocação
., ->	acesso a membro
++, --	prefixo
~, !	lógico
(type) var	modelador
*	acesso indireto
&	endereço
sizeof	tamanho
new, new[]	alocar
delete, delete[]	dealocar
+, -	sinalizadore
*, /, %	aritimético
+, -	aritimético
<<, >>	deslocador
<, <=, >=, >	comparador
==, !=	comparador
&	lógico
^	lógico
	lógico
&&	lógico
	lógico
?:	ternário
=	atribuidor
+=, -=, *=, /=, %=, &=, ^=, =, <<=, >>=	atribuidor