Sumário

Introdução

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

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 e 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 A. 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

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++. Uma 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 dependes da arquitetura.

2.2.2 Números flutuante

Existem dois tipos para armazenamento de números flutuantes, eles apenas diferem em precisão. O número fluante é o equivalente computacional ao número real na matemática, porém apenas um número finito de valores podem ser representada 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.

$$\pm M \cdot B^{\pm e} \tag{2.1}$$

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 utiliza para outras coisas na linguagem.

$$\cdot \neq ,$$

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 caracter 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 usualidade, 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 valore 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.

$$\texttt{'a'} \neq \texttt{"a"}$$

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 tipo `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 extendida. Estas palavras-chave alteram o espaço ocupado pelo tipo de memória extendendo 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
`bool`	1	0	1
`signed char`	1	-127	126
`char`	1	-127	126
`unsigned char`	1	0	255
`signed short int`	2	-32768	32767
`short int`	2	-32768	32767
`unsigned short int`	2	0	64535
`signed int`	4	-2147483648	2147483647
`int`	4	-2147483648	2147483647
`unsigned int`	4	0	4294967295
`signed long int`	8	-9223372036854775808	9223372036854775807
`long int`	8	-9223372036854775808	9223372036854775807
`unsigned long int`	8	0	18446744073709551616
`float`	4	$1.2 \cdot 10^{-38}$	$3.4 \cdot 10^{+38}$
`double`	8	$1.73 \cdot 10^{-308}$	$1.7 \cdot 10^{+308}$
`long double`	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 idea 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 captalização 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 declarar ela 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 aritiméticos, 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 parenteses (`()`). O valor escolhido deve ser pensado de acordo com o objetido 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.

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

Código 4: Declaração de variável

3.3 Exemplos de variáveis

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

```
//...
bool falso(0);           //Com número
bool verdadeiro(true);   //Com palavra-chave
char igual(0x3D);        //Sinal de igual ASCII
char letraA('A');        //Aspas simples

int contador(1), acumulador(0);   //Várias variáveis do mesmo tipo
unsigned int positivo(523);       //Inteiro sem sinal
short doisBytes(93);              //Modificador de comprimento
long grande(32416189349L);        //Número grande
double cargaFundamental(1.6e-19); //Notação científica
float pi(3.14159265358979323846264338327950288419f);  //Flutuante preciso
//...
```

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 perterce. O código 6 mostra um exemplo.

```
//...
int A;       //Somente A pode ser acessado aqui
{
  int B;     //Aqui A será global e B local, C e D são inexistentes
  {
    int C;   //Aqui A e B são globais, C local e D inexistente
  }
  {
    int D;   //Aqui A e B são globais, D local e C inexistente
  }
}
//...
```

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 número.

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.

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

Código 7: Sintaxe geral de operadores unários

```
//...
<tipo> <nome> (<valor>);  //Operadores normalmente atuam sobre variáveis
int A(<operador> <nome>); //Operaores serão utilizados sempre num contexto
    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 intremento é o duplo mais (++). O operador responsável pelo decremento é o duplo menos (--). A sintaxe associada é apresentada no código 9.

```
//...
<nome>++; //Incremento posfixo
++<nome>; //Incremento prefixo

<nome>--; //Decremento posfixo
```

```
--<nome>; //Decremento prefixo
//...
```

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 valtor e entao raliza a operação. Um exemplo é encontrado no código 10.

```
//...
int A(5);   //A vale 5
int B(A++); //A vale 6, B vale 5
int C(++B); //A vale 6, B vale 6, C vale 6
C++;        //Também pode ser usado independente do valor de retorno

int D(5);   //D vale 5
int E(--D); //D vale 4, E vale 4
int F(E--); //D vale 4, E vale 3, F vale 4
//...
```

Código 10: Exemplo de incrementadores e decrementadores initários

4.1.2 Sinalizadores aritiméticos

Todas as variáveis apresentadas até agora estavam armazenando valores não negativos, isso porque é necessário utilizar um operador para descrever um numero 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 negativado de um tipo é o sinal de menos (`-`). O operador que retorna o correspondente positivado de um tipo é o sinal de mais (`+`). A sintaxe associada é apresentada no código 11.

```
//...
-<name>;  //Só aparece em um contexto onde o retorno é utilizado

+<name>;
//...
```

Código 11: Sintaxe do sinalizadores aritiméticos

Vale notar que, matematicamente, são equivalente 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.

```
//...
int A(-5);  //A vale -5
int B(+A);  //A vale -5, B vale -5
int C(-A);  //A vale -5, B vale -5, C vale 5

int D(5);   //D vale 5,
int E(-D);  //D vale 5, E vale -5
int F(+D);  //D vale 5, E vale -5, F vale 5
//...
```

Código 12: Exemplo de sinalizadores aritiméticos

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.

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

Código 13: Sintaxe da negação lógica

Alguns exemplos podem ser encontrador no código 14.

```
//...
bool A(true);    //A vale 1
bool B(!A);      //B vale 0
bool C(!B);      //C vale 1

bool D(!true);   //D vale 1
bool E(!false); //E vale 0

int F(10);       //F vale 10
int G(!F);       //G vale 0
int H(!G);       //H vale 1
//...
```

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.

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

Código 15: Sintaxe do complemento binário

```
//..
unsigned char A(0b10100101);  //A vale 0b10100101 ou 0xA5
unsigned char B(~A);          //B vale 0b01011010 ou 0x5A
/*
unsigned char C(Ã);           Tome cuidado para isso não acontecer
/*
//..
```

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.

```
//...
<nome1> <operador> <nome2>; //Onde <nome> refere-se ao identificador da vari
    ável
//...
```

Código 17: Sintaxe geral para operadores binários

```
//...
<tipo1> <nome1> (<valor1>);
<tipo2> <nome2> (<valor2>);
int A(<nome1> <operador> <nome2>);  //Operaores aparecem num contexto
    adequado
//...
```

Código 18: Exemplo genérico de operadores binários

4.2.1 Atribuição simples

O primeiro operador binário apresentado é o de atribuição simpes, que utiliza o sinal de igual (=). É responsável pela atribuição de novos valores à 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.

```
//...
<alvo> = <item>;
//...
```

Código 19: Sintaxe do operador de atribuição

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

```
//...
int A;
A = 10;              //A passa a valer 10
float B = 5.1;       //Operadores podem ser usados na declaração.

float C(B = 13.25); //Todo operador retorna o valor de sua operação

int D = A = 20;
/*
  Da direita para a esquerda para a direita, A passa a valer 10
  Então D passa a ter o valor da operação à direita, 10
*/
//...
```

Código 20: Exemplo do operador de atribuição

4.2.2 Aritiméticos

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

```
//...
<valor1> + <valor2>;  //O uso só é coerente em casos onde o retorno é
    utilizado
<valor1> - <valor2>;
<valor1> * <valor2>;
<valor1> / <valor2>;
<valor1> % <valor2>;
//...
```

Código 21: Sintaxe dos operadores aritiméticos

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 direita. 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 quociênte, e o módulo retorna o resto da divisão. Exemplos podem ser encontrador no código 22, inclusive o caso de módulo equivalente ao da equação 4.1.

$$\begin{array}{c|c} E & F \\ \hline G & H \end{array} \Rightarrow \begin{array}{c|c} 13 & 5 \\ \hline 3 & 2 \end{array} \tag{4.1}$$

```
//...
int A(45 + 5);  //A vale 50
int B(A - 15);  //B vale 35
int C(B - A);   //C vale  - 15

int D(A + B - C); //D vale o mesmo que 50 + 35 - ( - 15), ou seja 100

int E(13),  F(5);           //E vale 13, F vale 5
int G(E % F), H(E / F);     //G vale 3, H vale 2
int I(F * H);               //I vale 10
int J(I + G);               //J vale 13

float K(13.0f), L(5.0f);
float M(K / L);             //K vale 2.6
float N(L * M);             //N vale 13
//...
```

Código 22: Exemplo de operadores aritiméticos

O operador módulo é especialmente útil quando precisa-se verificar se um número é multiplo 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 mamória 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.

```
//...
(valor) << (deslocamentos); //Só é coerente quando o retorno é utilizado
(valor) >> (deslocamentos);
//...
```

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.

$$V \cdot 2^{+S} \tag{4.2}$$

$$V \cdot 2^{-S} \tag{4.3}$$

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.

```
//...
unsigned char A(0b01100000);
unsigned char B(A>>3);   //B vale 0b00001100
unsigned char C(B*8);    //C vale 0b01100000

int D(30);               //D vale 30
int E(D>>1);             //E vale 30/(2), ou seja 15
int F(D<<2);             //F vale 30*(4), ou seja, 120
//...
```

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 casa 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 oprações 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.

```
//...
<valor1> | <valor2>;
<valor1> & <valor2>;
<valor1> ^ <valor2>;
//...
```

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

```
//...
unsigned char A(0b10101011);
unsigned char B(0b01100100);
unsigned char C(A|B);          //C vale 0b11101111
unsigned char D(A&B);          //D vale 0b00100000
unsigned char E(A^B);          //E vale 0b11001111
//...
```

Código 26: Exemple dos operadores lógicos bitwise

$$\begin{array}{r} 10101011 \\ OR\ 01100100 \\ \hline 11101111 \end{array} \quad \begin{array}{r} 10101011 \\ AND\ 01100100 \\ \hline 00100000 \end{array} \quad \begin{array}{r} 10101011 \\ XOR\ 01100100 \\ \hline 11001111 \end{array} \tag{4.4}$$

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.

```
//...
<nome> = <nome> <operador> <valor>; //Onde nome é uma variável
<nome> <operador>= <valor>;         //Operador de atribuição composta
//...
```

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;

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

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 lógicamente 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 esquera é maior que o à direita, utilizamos o sinal de maior (`>`). Verificar se o valor à esquera é menor que o à direita, utilizamos o sinal de maior (`<`). 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 encontrador no código 30.

```
//...
<valor1> == <valor2>; //Só é coerente quando o retorno é utilizado
<valor1> != <valor2>;
<valor1> < <valor2>;
<valor1> > <valor2>;
<valor1> <= <valor2>;
<valor1> >= <valor2>;
//...
```

Código 29: Sintaxe de operadores de comparação

```
//...
bool A(true);
int B(10), C(10), D(15);

bool E(B>C);     //E vale 0
bool F(A==E);    //F vale 0

bool G(D>=B);    //G vale 1
bool H(E!=G);    //H vale 1
bool I(B==C);    //I vale 1
//...
```

Código 30: Sintaxe de operadores de comparação

4.2.7 Lógicos booleanos

Semelhantes aos lógicos bit-a-bit, os lógicos boleanos 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 simbolo *e comencial* 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.

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

Código 31: Sintaxe de operadores lógicos booleanos

```
//...
bool T(true), F(false);

bool A(T || F); //A vale 1
bool B(T && F); //B vale 0
//...
```

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.

```
//...
<codicional> ? <valor1> : <valor2>;
//...
```

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 encontrador no código 34.

```
//...
bool A(true);
int C(10), D(15);
bool E(C>=D);      //E vale 0
bool F(A==E);      //F vale 0
int G(F?10:50);    //G vale 50
///...
```

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

```
//...
int A = 25 * 40;
int B = 1 << 4;
A /= B + 4;

A = A + B;
B = A - B;
A = A - B;

float C = A > 200 ? A * (50.0f - 0.003f) : B % 5;
//...
```

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 sequêncialmente descritos no código, como ilustrado na figura 5.1. Esta sequência é única, ou seja, não existem situações parelelas, pois em C++ não há por padrão a multiexecução (*muti 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.

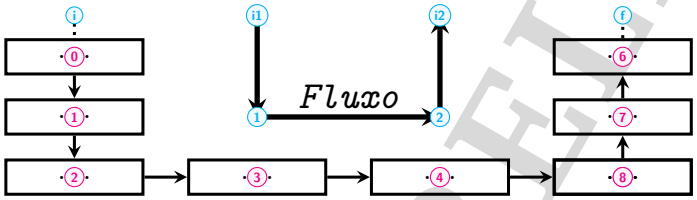

Figura 5.1: Fluxograma de fluxo simples

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.

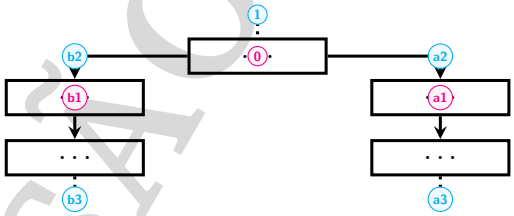

Figura 5.2: Fluxograma de fluxo ambíguo

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 realização, 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 || //...
```

```
if(<cond>) <comand>;
//...
```

Código 36: Estrutura de *if* simples linear

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

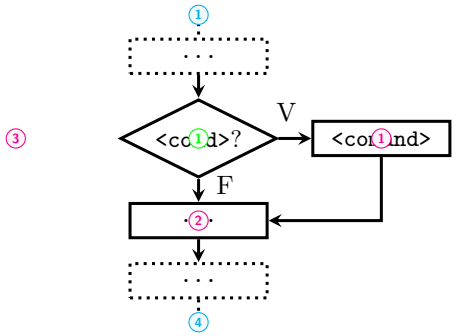

Figura 5.3: Fluxograma de *if* simples linear

Quando mais de um comando está condicionado ao mesmo estado lógico, é possível fazer uma sequencia 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.

```
//...
if(<cond>)
{
    <comand1>;
    //...
    <comandN>;
}
//...
```

Código 37: Estrutura de *if* simples blocular

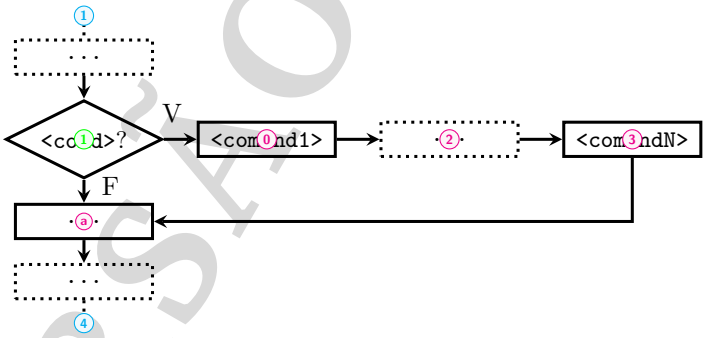

Figura 5.4: Fluxograma de *if* simples blocular

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.

```
//...
int A(50), B(25), C(15);
```

```
if (A<B) C *= 2;   //Alteração de valor condicionada
if (C<A)           //Inversão de valores de variáveis
{
  B = A;
  A = C;
  C = B;
}
//...
```

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 me máquina, de maneira que em programas grandes é perceptível a baixa no desenpenho.

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.

```
//...
if(<cond>) <comandA>;
else <comandB>;
//...
```

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.

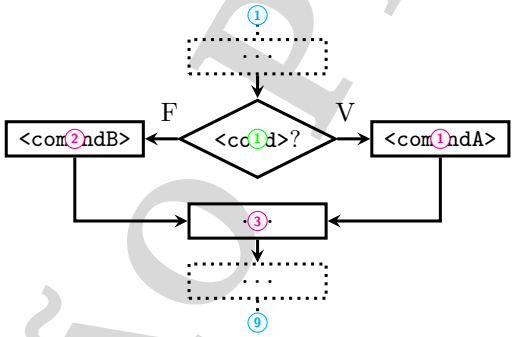

Figura 5.5: Fluxograma de *if* complementares linear

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.

```
//...
if(<cond>)
{
    <comandA1>;
    //...
    <comandAN>;
}
else
{
```

```
    <comandB1>;
    //...
    <comandBM>;
}
//...
```

Código 40: Estrutura de *if* complementares blocular

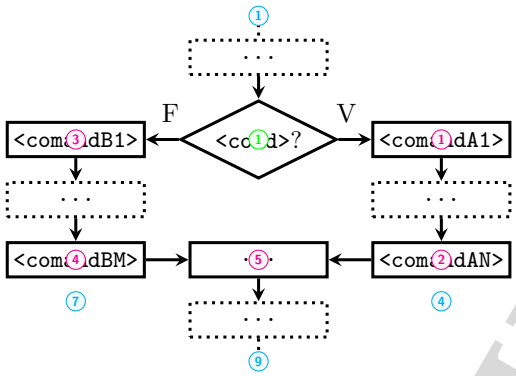

Figura 5.6: Fluxograma de *if* complementares blocular

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 estrutua 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.

```
//...
if(<cond0>)
{
    //...
    if(<condA>) <comandAA>;
    else <comandAB>;
    //...
}
else
{
    //...
    if(<condB>) <comandBA>;
    else <comandBB>;
    //...
}
//...
```

Código 41: Estrutura de *if* aninhado

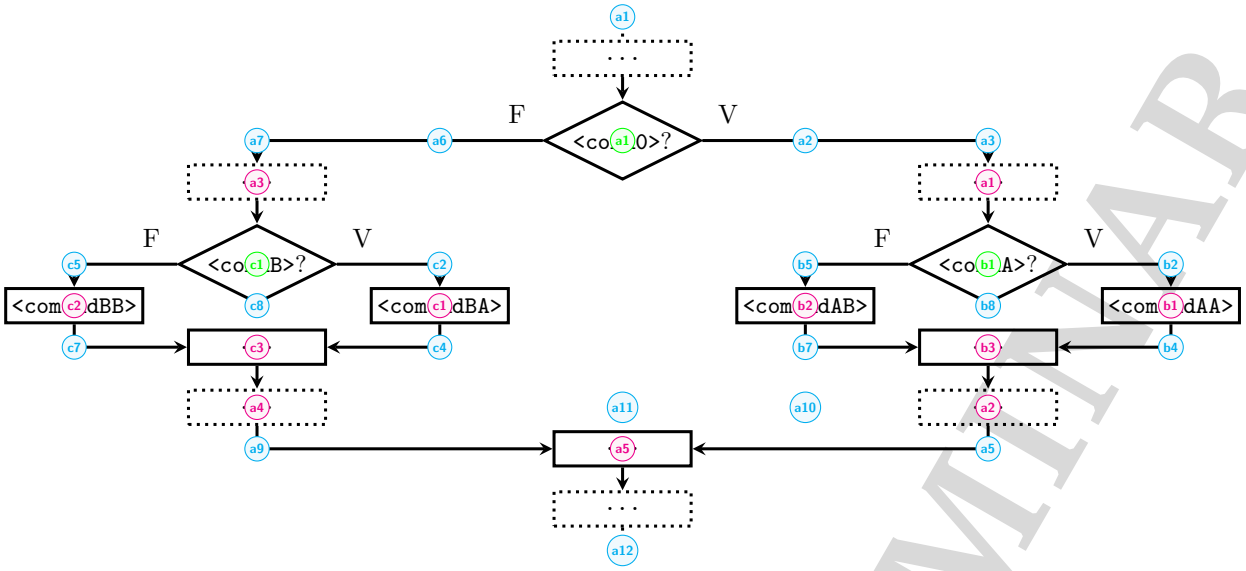

Figura 5.7: Fluxograma de *if* aninhado

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 desenpenho do programa.

```
//...
float nota(8.7);
char conceito('\0');
//...
     if (nota >= 9.5)    conceito = 'A';
else if (nota >= 8.0)    conceito = 'B';
else if (nota >= 7.0)    conceito = 'C';
else if (nota >= 4.5)    conceito = 'D';
else                     conceito = 'F';
//...
```

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.

```
//...
if (10.0>=nota && nota >= 9.5) conceito = 'A';
if (9.5 > nota && nota >= 8.0) conceito = 'B';
if (8.0 > nota && nota >= 7.0) conceito = 'C';
if (7.0 > nota && nota >= 4.5) conceito = 'D';
if (4.5 > nota && nota >= 0.0) conceito = 'F';
//...
```

Código 43: Exemplo de *if* sequencial

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

```
//...
     if (nota < 4.5)  conceito = 'F';
else if (nota < 7.0)  conceito = 'D';
else if (nota < 8.0)  conceito = 'C';
else if (nota < 9.5)  conceito = 'B';
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
31          {
32            if (A == 5)
33            {
34              <comand5A>;
35              <comand5B>;
36              <comand5C>;
37              <comand5D>;
38            }
39            else <comand0>;
40          }
41        }
42    }
43  }
44  //...
```

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.

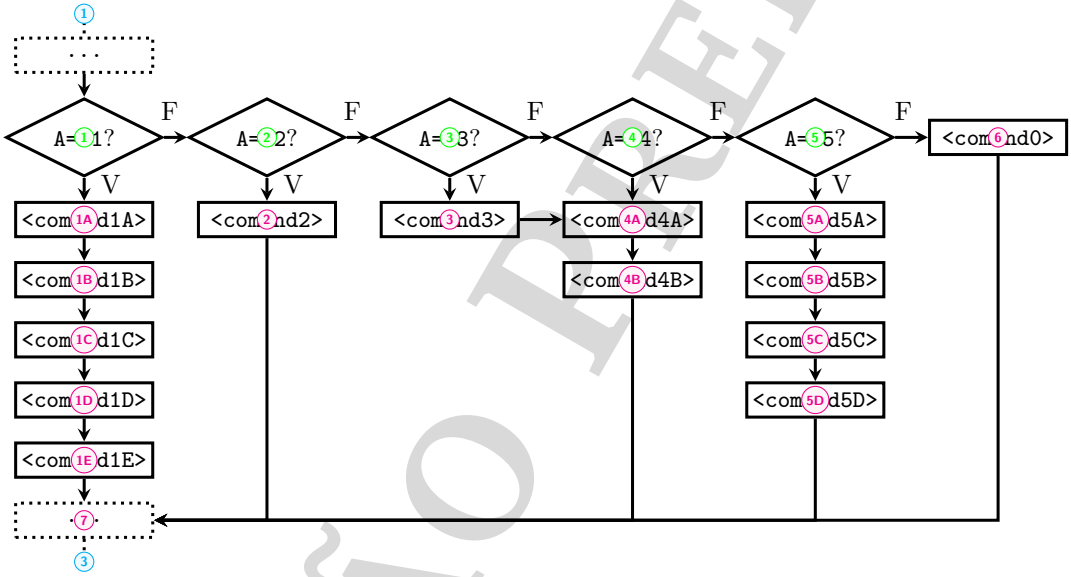

Figura 5.8: Fluxograma de esturura de decisão composta

Os valore de um `switch` são colocadas a direita do 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 comando 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 quano não há `break` previamente). Este caso especial serve, principalmente, para situações de erro, onde nenhuma das condições foi aceita.

```
//...
int A;
//...
switch(A)
{
  case 1:
    <comand1A>;
    <comand1B>;
    <comand1C>;
    <comand1D>;
    <comand1E>;
    break;
  case 2:
    <comand2>;
    break;
  case 3:
    <comand3>;
  case 4:
    <comand4A>;
    <comand4B>;
    break;
  case 5:
    <comand5A>;
    <comand5B>;
    <comand5C>;
    <comand5D>;
    break;
  default:
    <comand0>;
}
//...
```

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.

```
//...
bool aprovado;
//...
switch(conceito)
{
  case 'A':
  case 'B':
  case 'C':
  case 'D':
    aprovado = true;
    break;
  case 'F':
    aprovado = false;
    break;
}
//...
```

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 calculo 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 estutura sem a utilização de estruturas de repetição.

```
//...
int a, b, X, Y;
a=45; b=93;
//...
X = a>b?a:b;   //#1
Y = a<b?a:b;
a = X-Y;
b = Y;

X = a>b?a:b;   //#2
Y = a<b?a:b;
a = X-Y;
b = Y;

//Repete-se 18 vezes!

X = a>b?a:b;   //#17
Y = a<b?a:b;
a = X-Y;
b = Y;

X = a>b?a:b;   //#18
Y = a<b?a:b;
a = X-Y;       //a vale 3, o resultado
b = Y;         //b vale 0
//...
```

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.

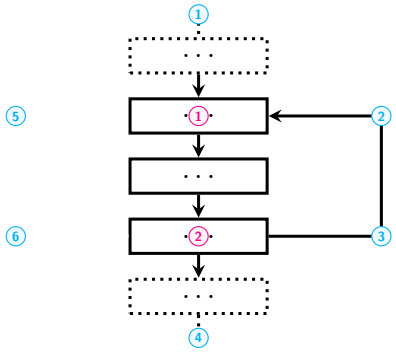

Figura 5.9: Fluxograma de fluxo repetitivo

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

```
//...
while(<cond>) <comand>;
//...
```

Código 50: Estrutura de *while* simples linear

A estrutura do `while` é idêntica à do `if`.

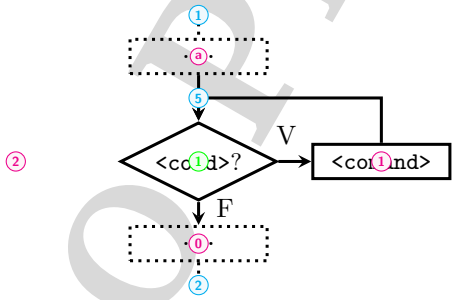

Figura 5.10: Fluxograma de *while* simples linear

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

```
//...
while(<cond>)
{
  <comand1>;
  <comand2>;
  //...
  <comandN>;
}
//...
```

Código 51: Estrutura de *while* simples blocular

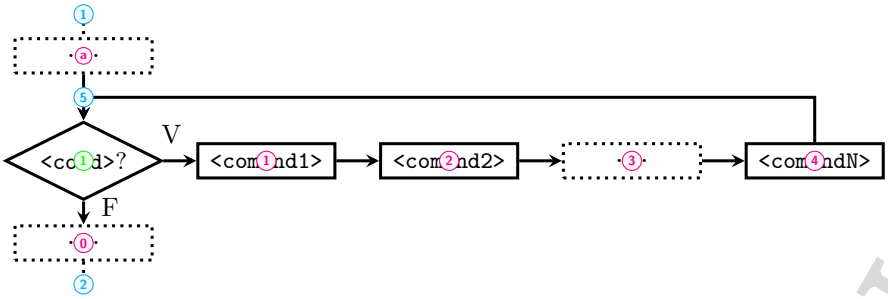

Figura 5.11: Fluxograma de *while* simples blocular

O mesmo algoritmo de Euclides do código 49 pode ser escrito como no código 52. Note que nesse caso o programdor 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.

```
//...
while(b!=0)
{
   X = a>b?a:b;
   Y = a<b?a:b;
   a = X-Y;
   b = Y;
}
//...
```

Código 52: Algoritmo de Euclides com estrutura repetição

5.2.2 O *do* e a estrutura de inicio obrigatório

Esta estrutura é muito semelhante ao `while` simples, 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.

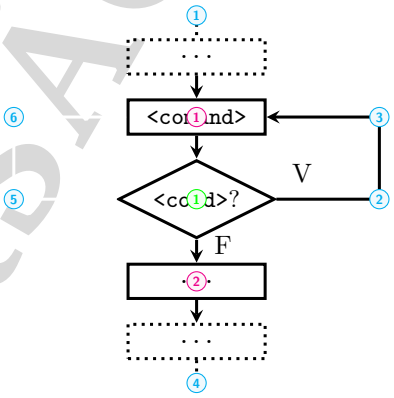

Figura 5.12: Fluxograma de *do while* simples linear

```
//...
do <comand>; while(<cond>);
//...
```

Código 53: Estrutura de *do while* simples linear

Note que, após o `while` há um ponto-e-vírgula (`;`), que indica térimino 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.

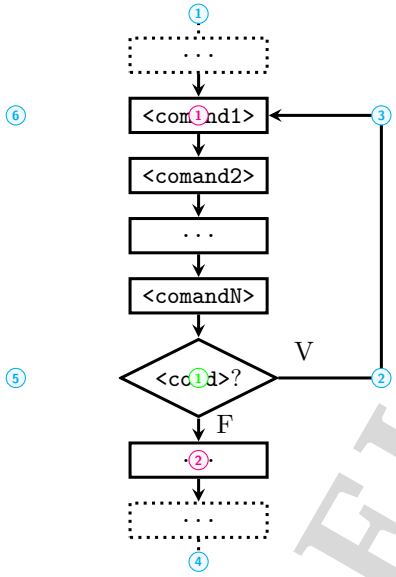

Figura 5.13: Fluxograma de *do while* simples blocular

```
//...
do
{
  <comand1>;
  <comand2>;
  //...
  <comandN>;
}
while(<cond>);
//...
```

Código 54: Estrutura de *do while* simples blocular

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

Parte II

Intermediário

Capítulo 6

Funções

Capítulo 7

Mais operadores

Parte III

Avançado

Lista de Figuras

Códigos

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

Apêndice A

Tabela ASCII

A tabela A.1 apresenta a codificação ASCII extendida, 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` nao 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 caracter `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 extendida incompleta[1].

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	å	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	À	©	╣	║	╗	╝	¢	¥	┐
CX	└	┴	┬	├	─	┼	ã	Ã	╚	╔	╩	╦	╠	=	╬	¤
DX			Ê	Ë	È	ı	Í	Î	Ï	┘	┌	▓	▓	¦	Ì	▓
EX	Ó	ß	Ô	Ò	õ	Õ	µ			Ú	Û	Ù	ý	Ý	¯	´
FX		±		¾	¶		÷	¸	°	¨	·	¹	³	²		nbsp

[1]Problemas 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.