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.

```
{;}
```

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, completamente distintos e sem qualquer relação natural, 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.

$$A \neq a$$

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 1.1 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 1.1: 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` e `falso`. Seu nome deriva do termo *boolean*.

Em C++, um `bool` pode receber os estados lógicos utilizando os valores numéricos `1` e `0`, respectivamente para verdadeiro e falso, ou as palavras-chave `true` e `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.

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 é 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 que o outro, 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.

$$\text{`.'} \neq \text{`,'}$$

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 simbolo 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 simbolo, 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 simbolos 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 simbolos, 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.

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

2.4 Vazios

O tipo `void` é diferente dos demais, ele não armazena valore, 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 explicito 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 leteral 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

Há ainda a opção de definição de faixa em literal, 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 coisas 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 3.1. Respectivamente a lista de modificadores, separados por espaço, o nome do tipo e finalmente o nome da variável.

O nome de uma variável pode contes 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 3.1: Declaração de variável

É recomendada a preferência de nomes de variáveis com letras minúsculas, pois letras maiúsculas costumam ser utilizadas em outras situações 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 porém também 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 utiilizada depois de ter sido decladada, ou seja, não é possível utilizar uma variável e declarar ela duas linhas abaixo.

3.2 Inicializando variáveis

A declaração do código 3.1 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 definido.

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 3.2: Declaração de variável

3.3 Exemplos de variáveis

Alguns exemplos de declaração e inicalização de variáveis são apresentados no código 3.3, 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 3.3: Declarações de variável

3.4 Escopo

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, ela não pode ser acessada fora dele. Uma variável global é declarada fora de um bloco de código, ela pode ser acessada de qualquer parte do programa.

Existe ainda uma situação relativa, onde uma variável é localmente global, ou seja, pode ser acessada em qualquer região dentro do bloco no qual foi declarado, até bloco internos, mas não pode ser acessada fora dele. O código 3.4 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 3.4: Declarações de variável

Capítulo 4

Operadores

Operadores são as entidades de código 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 que o valor armazenada em memória não sofre alteração, como se uma cópia fosse criada e esta cópia 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, então, se ele soma dois números, devolve a soma como um dado de um tipo, normalmente o tipo maior de dados (ou o mais preciso).

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

Existem mais operadores além dos descritos aqui, porém seu uso requer domínio de outras técnicas, portante serão apresentados na capítulo 7.

4.1 Unários

De uma maneira geral, todo operador unário tem uma sintaxe semelhante, apresentada no código 4.1, e um exemplo genérico no código 4.2.

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

Código 4.1: 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 4.2: 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 4.3.

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

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

Código 4.3: 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 4.4.

```
//...
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

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 4.4: 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 4.5.

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

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

Código 4.5: Sintaxe do sinalizadores aritiméticos

Vale notar que, matematicamente, são equivalente a multiplitar 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 4.6.

```
//...
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 4.6: 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. A sintaxe associada é apresentada no código 4.7. Vale ressaltar que não se relaciona à operação de fatorial da matemática, que utiliza o mesmo símbolo.

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

Código 4.7: Sintaxe da negação lógica

Alguns exemplos podem ser encontrador no código 4.8.

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

bool D(!true);
bool E(!false);
//...
```

Código 4.8: 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 4.9 e um exemplo de uso é apresentado no código 4.10. 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 4.9: 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 4.10: Exemplo de negação lógica

Capítulo 5

Controladores de Fluxo

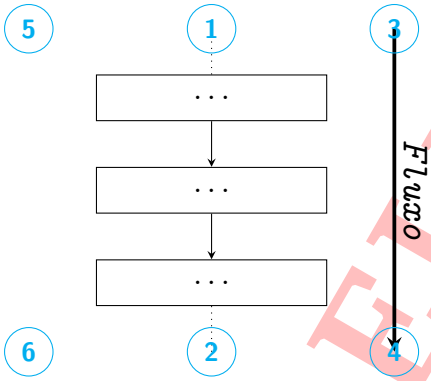

Figura 5.1: Fluxograma de fluxo simples

5.1 Decisões na direção do fluxo

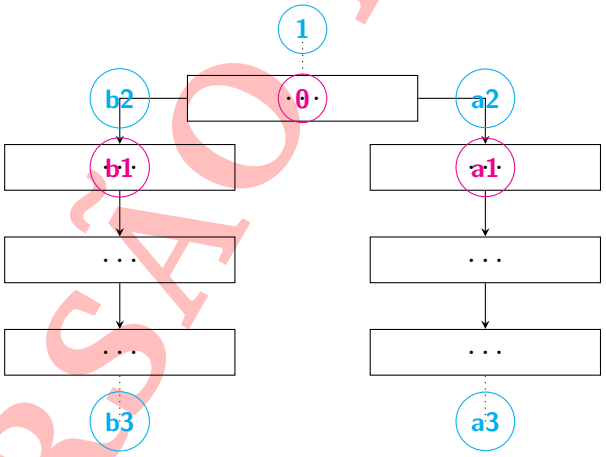

Figura 5.2: Fluxograma de fluxo ambíguo

5.2 O *if* e a estrutura básica de decisão

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

Código 5.1: *if* simples linear

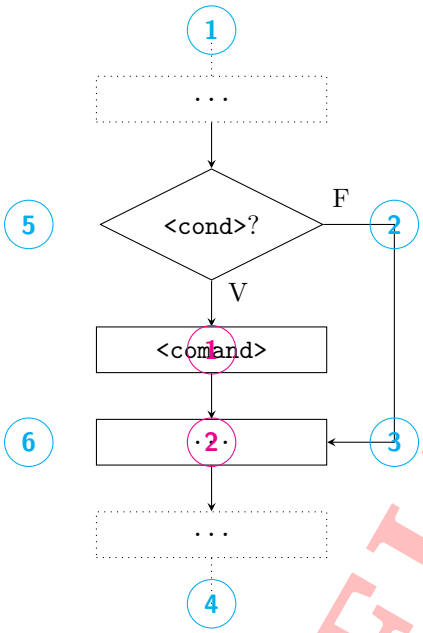

Figura 5.3: Fluxograma de *if* simples linear

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

Código 5.2: *if* simples blocular

5.3 O *else* e as estruturas duplas

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

Código 5.3: *if* composto linear

```
//...
if(<cond>)
{
    <comandA1>;
    <comandA2>;
    //...
```

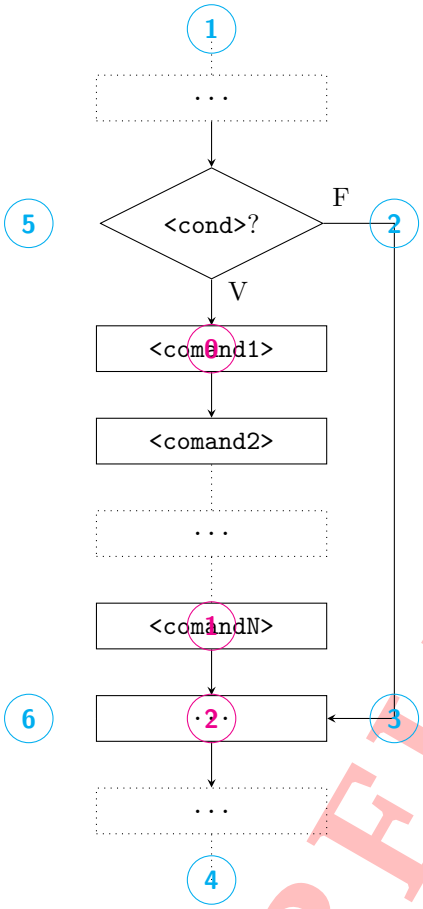

Figura 5.4: Fluxograma de *if* simples blocular

```
    <comandAN>;
}
else
{
    <comandB1>;
    <comandB2>;
    //...
    <comandBM>;
}
//...
```

Código 5.4: *if* composto blocular

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

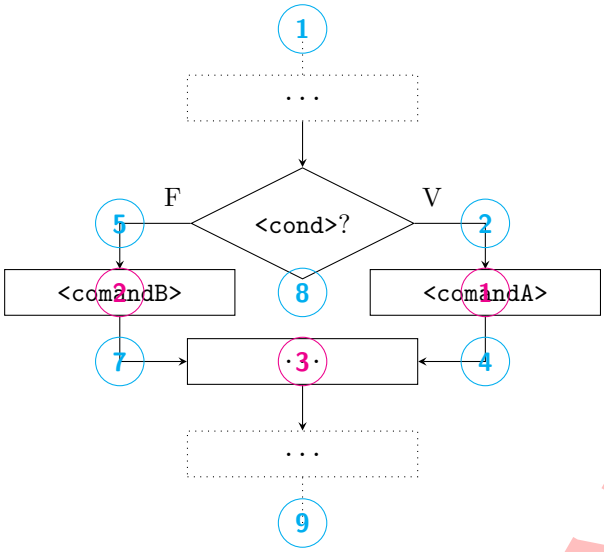

Figura 5.5: Fluxograma de *if* composto linear

```
14        //...
15    }
16    //...
```

Código 5.5: *if* aninhado

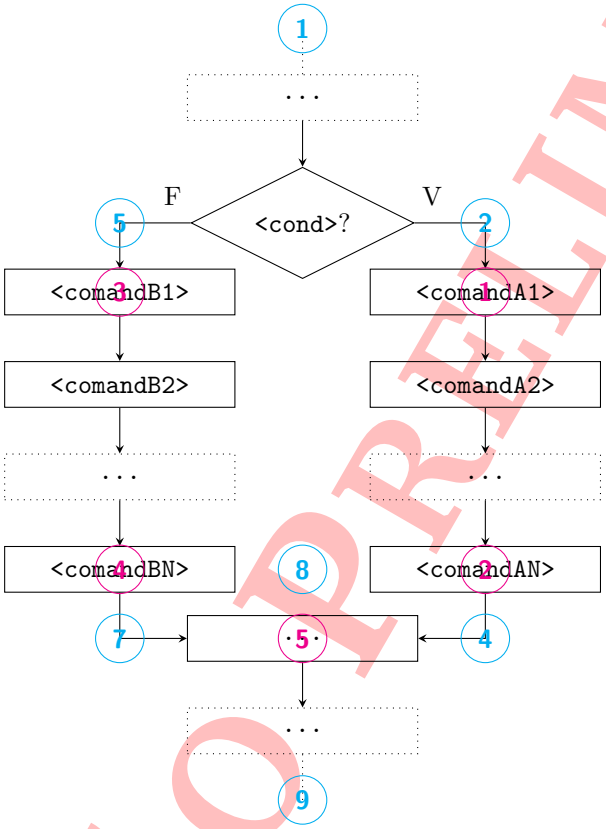

Figura 5.6: Fluxograma de *if* composto blocular

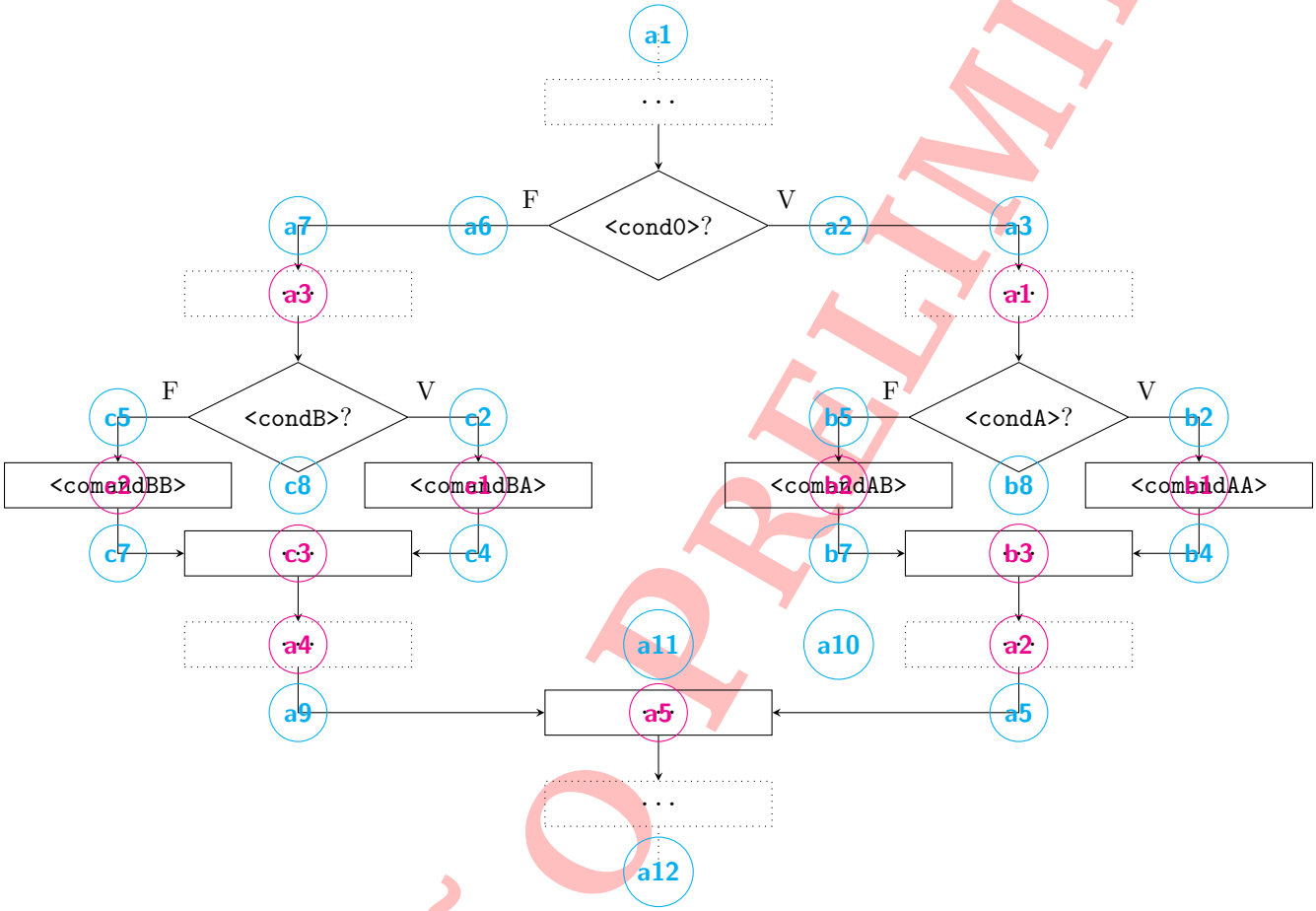

Figura 5.7: Fluxograma de *if* aninhado

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

Apêndice A

Tabela ASCII

A tabela A.1 apresenta a codificação ASCII extendida, dividida em duas partes, na superio 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.

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.