

# **Material de apoio Curso de C/C++ DINTER**

21 a 25 de fevereiro de 2011

Propriedades do documento

Categoria: Apostila

Assunto : Linguagem de programação C/C++

Versão : 1.0

<b>elaborado por</b>	<b>data</b>
Waldemar P. Mathias Neto	27.01.2011
<b>revisado por</b>	
Fábio Bertequini Leão	06.02.2011

---

**NOTA:** O conteúdo deste documento foi integralmente retirado, traduzido e adaptado de <http://www.cplusplus.com/doc/tutorial> e outras seções do website <http://www.cplusplus.com/>. Acesso em: 28 de janeiro de 2011.

# Sumário

Capítulo 1: O básico de C++ .....	6
1.1. A estrutura de um programa .....	6
1.1.1 Comentários.....	9
1.2 Variáveis. Tipo de dados.....	10
1.2.1 Identificadores.....	10
1.2.2 Tipos fundamentais de dados .....	11
1.2.3 Declaração das variáveis.....	12
1.2.4 Escopo das variáveis .....	14
1.2.5 Inicialização das Variáveis.....	14
1.2.6 Introdução às cadeias de caracteres .....	15
1.3 Constantes.....	16
1.3.1 Literais .....	16
Numerais inteiros .....	17
Números de ponto flutuante.....	17
Caracteres e literais do tipo <i>string</i> .....	18
Literais booleanas.....	19
1.3.2 Constantes definidas ( <i>#define</i> ) .....	19
1.3.3 Constantes declaradas ( <i>const</i> ) .....	20
1.4 Operadores.....	20
1.4.1 Atribuição (=).....	21
1.4.2 Operadores aritméticos ( +, -, *, /, % ).....	22
1.4.3 Atribuição composta (+=, -=, *=, /=, %=, >>=, <<=, &=, ^=,  =).....	23
1.4.4 Incremento e decremento (++ , --) .....	23
1.4.5 Operadores relacionais e de igualdade ( ==, !=, >, <, >=, <= ) .....	24
1.4.6 Operadores lógicos ( !, &&,    ).....	25
1.4.7 Operador condicional ( ? ) .....	25
1.4.8 Operador virgula ( , ) .....	26
1.4.9 Operadores bit a bit ( &,  , ^, ~, <<, >> ) .....	26
1.4.10 Operadores de conversão explícita de tipo.....	27
1.4.11 sizeof().....	27
1.4.12 Outros operadores .....	27
1.4.13 Precedência dos operadores .....	28
Capítulo 2: Estruturas de controle.....	30
2.1 Estruturas de controle .....	30
2.1.1 Estruturas condicionais: if e else.....	30
2.1.2 Estruturas de iterações (loops) .....	31
O loop while (enquanto) .....	31
O loop do-while (faça-enquanto) .....	32
O loop for (para) .....	33

2.1.3	Instruções de salto.....	34
	O comando break .....	34
	O comando continue .....	35
	O comando goto.....	35
	A função exit.....	36
2.1.4	A estrutura seletiva: switch .....	36
2.2	Funções (Parte I) .....	38
2.2.1	Escopo das variáveis .....	39
2.2.2	Funções void (sem tipo).....	42
2.3	Funções (Parte II).....	43
2.3.1	Argumentos passados por valor e por referência .....	43
2.3.2	Parâmetros com valores padrões .....	45
2.3.3	Sobrecarga de funções .....	46
2.3.4	Recursividade .....	46
2.3.5	Declarando funções .....	47
	Capítulo 3: Tipos de dados compostos .....	50
3.1	Matrizes .....	50
3.1.1	Inicializando matrizes .....	51
3.1.2	Acessando valores de uma matriz .....	51
3.1.3	Matrizes multidimensionais .....	53
3.1.4	Matrizes como parâmetros .....	54
3.2	Sequência de caracteres .....	56
3.2.1	Inicialização de sequências de caracteres terminada por caractere nulo .....	57
3.2.2	Usando uma sequência de caracteres terminada por caractere nulo .....	58
3.3	Ponteiros .....	59
3.3.1	O operador referência (&).....	59
3.3.2	Operador dereferência (dereference)(*) .....	60
3.3.3	Declarando variáveis do tipo ponteiro.....	62
3.3.4	Ponteiros e matrizes .....	64
3.3.5	Inicialização de ponteiros.....	65
3.3.6	Aritmética de ponteiros .....	66
3.3.7	Ponteiros para ponteiros.....	68
3.3.8	Ponteiros void .....	69
3.3.9	Ponteiro nulo .....	70
3.3.10	Ponteiros para funções .....	70
3.4	Alocação dinâmica de memória.....	71
3.4.1	Operadores new e new[] .....	71
3.4.2	Operadores delete e delete[] .....	73
3.5	Estruturas de dados .....	74
3.5.1	Estruturas de dados .....	74
3.5.2	Ponteiros para estruturas.....	77

3.5.3	Estruturas aninhadas.....	79
3.6	Outros tipos de dados .....	80
3.6.1	Tipo de dados definido (typedef) .....	80
3.6.2	Uniões (union).....	80
3.6.3	Uniões anônimas.....	82
3.6.4	Enumerações (enum) .....	82
Anexo A	(alguns cabeçalhos e bibliotecas).....	84
A.1	Algorithms.....	84
A.2	complex .....	86
A.3	cmath (math.h).....	87
A.4	cstdio (stdio.h).....	88
A.5	cstdlib (stdlib.h) .....	91
A.6	ctime (time.h).....	92

# Capítulo 1: O básico de C++

## 1.1. A estrutura de um programa

Provavelmente a melhor maneira para começar a aprender uma linguagem de programação é escrever um programa. Portanto, aqui está o nosso primeiro programa:

<pre>1 // my first program in C++ 2 3 #include &lt;iostream&gt; 4 using namespace std; 5 6 int main () { 7     cout &lt;&lt; "Hello World!"; 8     return 0; 9 }</pre>	<pre>Hello World!</pre>
--	-------------------------

O primeiro quadro (em azul) mostra o código fonte para o nosso primeiro programa. O segundo (em cinza claro) mostra o resultado do programa quando compilado e executado. À esquerda, os números em cinza representam os números de linha - estas não fazem parte do programa, e são mostrados aqui apenas para fins informativos.

A maneira de editar e compilar um programa depende do compilador usado pelo programador. Se ele tem uma interface de desenvolvimento, ou não, e a sua versão. Consulte o manual ou a ajuda que acompanha o seu compilador caso você tenha dúvidas sobre como compilar um programa em C++.

O programa mostrado nos quadros anteriores é o típico programa que os programadores iniciantes aprendem a escrever pela primeira vez, e seu resultado é imprimir na tela a sentença "Hello World!". Este é um dos programas mais simples escritos em C++, mas contém os componentes fundamentais de qualquer programa escrito nesta linguagem. Vamos olhar linha por linha o código que escrevemos:

```
// my first program in C++
```

Esta é uma linha de comentário. Todas as linhas começando por duas barras (//) são consideradas comentários e não têm nenhum efeito sobre o programa. O programador usa elas para incluir pequenos comentários ou observações junto ao código. No exemplo, esta linha é uma breve descrição sobre o que é nosso programa.

```
#include <iostream>
```

Linhas começando com um Cerquilha (#) são diretivas para o pré-processador. Elas são indicações para o pré-processador do compilador e não linhas de código regular. No caso, a diretiva `#include <iostream>` diz ao pré-processador para incluir o arquivo `iostream`. Este arquivo contém declarações da biblioteca básica de entrada e saída em C++ e é incluído pela funcionalidade que será utilizada em linhas posteriores do programa.

```
using namespace std;
```

Todos os elementos da biblioteca padrão do C++ são declarados dentro do que é chamado `namespace`, e o nome do `namespace` padrão é `std`. Para acessar suas funcionalidades nós declaramos esta expressão. Esta linha de código é muito frequente em programas escritos em C++ que usam a biblioteca padrão.

```
int main ()
```

Esta linha corresponde ao início da definição da função `main`. A função `main` é o ponto onde todo programa escrito em C++ inicia sua execução. Não importa qual o número de funções que um programa contenha antes, ou depois da função `main`, ou ainda seus nomes, um programa sempre inicia sua execução pela função `main`.

A palavra `main` é seguida por um par de parênteses `()`. Neste caso trata-se de uma declaração de função: em C++, o que diferencia uma declaração de função de outro tipo de expressão são os parênteses após seu nome. Opcionalmente, estes parênteses podem conter uma lista de parâmetros.

Logo após os parênteses encontramos o corpo do programa. O corpo do programa é apresentado entre duas chaves `{}`. O conteúdo incluído dentro das chaves é o que será executado e onde nós iremos escrever o código do programa.

```
cout << "Hello World!";
```

Esta linha é uma declaração. A declaração é uma expressão simples ou composta que possa realmente produzir algum efeito. Na verdade, esta afirmação só executa a ação que gera um efeito visível no nosso primeiro programa.

`cout` é o nome do comando de saída padrão em C++. O resultado desta declaração é inserir uma sequência de caracteres ("Hello Word") na tela.

`cout` é declarado no arquivo padrão `iostream` dentro do `namespace std`. É por isso que nós precisamos incluir esse arquivo específico e declarar que usaremos este espaço para nome específico no início do nosso código.

Observe que esta instrução termina com um caractere ponto e vírgula `;`. Este é usado para marcar o final da instrução e, na verdade ele deve ser incluído ao final de todas as declarações em todos os programas C++ (um dos erros de sintaxe mais comum é de fato esquecer-se de incluir o ponto e vírgula depois de uma instrução).

```
return 0;
```

A instrução `return` faz com que a função `main` termine. `return` pode ser seguido por um código de retorno (no nosso exemplo é seguido pelo código de retorno com um valor de zero). Um código de retorno de 0 para a função `main` é geralmente interpretado como um correto e esperado funcionamento do programa, sem quaisquer erros durante sua

execução. Esta é a forma mais usual para encerrar um programa em C++.

Você deve ter notado que nem todas as linhas deste programa executam ações quando o código é executado. Há linhas contendo apenas comentários (aquelas que começam por `//`) e linhas com as diretivas para o compilador do pré-processador (aquelas iniciadas por `#`). Em seguida, linhas que iniciam a declaração de uma função (neste caso, a função principal) e, finalmente, as linhas com as declarações (como a linha 7), que foram incluídas dentro do bloco delimitado pelas chaves (`{}`) da função `main`.

O programa foi estruturado em duas linhas diferentes, a fim de ser mais legível, mas em C++, não temos regras rígidas sobre como separar instruções em linhas diferentes. Por exemplo, em vez de

```
1 int main () {  
2     cout << " Hello World!";  
3     return 0;  
4 }
```

Poderíamos ter escrito:

```
int main () { cout << "Hello World!"; return 0; }
```

Tudo isso em apenas uma linha tem exatamente o mesmo significado que o código anterior.

Em C++, a separação entre as afirmações é especificada com um ponto e vírgula (`;`) no final de cada sentença. Portanto a separação em diferentes linhas de código não importa para essa finalidade. Podemos escrever muitas declarações por linha ou escrever uma única instrução que tenha muitas linhas de código. A divisão do código em linhas diferentes é um recurso para tornar mais legível o código para os seres humanos.

Vamos adicionar uma instrução para o nosso primeiro programa:

<pre>1 // my second program in C++ 2 3 #include &lt;iostream&gt; 4 5 using namespace std; 6 7 int main () { 8     cout &lt;&lt; "Hello World! "; 9     cout &lt;&lt; "I'm a C++ program"; 10    return 0; 11 }</pre>	Hello World! I'm a C++ program
--	--------------------------------

Neste caso, são realizadas duas declarações do comando `cout` em duas linhas diferentes. Mais uma vez, a separação em diferentes linhas de código é feita apenas para dar maior legibilidade ao programa. A função `main` poderia ter sido escrita perfeitamente desta maneira:

```
int main () { cout << " Hello World! "; cout << " I'm a C++ program ";  
return 0; }
```



Também poderíamos separar o código em mais linhas se for conveniente:

```
1 int main () {
2     cout <<
3         "Hello World!";
4     cout << "I'm a C++ program";
5     return 0;
6 }
```

E o resultado seria novamente o mesmo dos exemplos anteriores.

As diretivas do pré-processador (aquelas que começam por #) estão fora dessa regra geral, uma vez que não são declarações. Estas linhas são lidas e processadas pelo pré-processador e não produzem qualquer código por si só. Estas diretivas devem ser especificadas em sua própria linha e não devem terminar com um ponto e vírgula (;).

### 1.1.1 Comentários

Os comentários são partes do código-fonte desconsideradas pelo compilador. Eles simplesmente não têm qualquer efeito sobre o código. O seu objetivo é apenas permitir que o programador possa inserir notas ou descrições dentro do código fonte.

O C++ suporta duas maneiras para inserir comentários:

```
1 // line comment
2 /* block comment */
```

O primeiro deles, conhecido como linha de comentário, descarta tudo, desde a declaração do par de barras (/ /) até o final dessa mesma linha. O segundo, conhecido como bloco de comentário, descarta tudo entre os caracteres /\* e a primeira declaração dos caracteres \*/ , com a possibilidade de incluir mais de uma linha.

Agora adicionaremos comentários ao nosso segundo programa:

<pre>1  /* my second program in C++ 2     with more comments */ 3 4  #include &lt;iostream&gt; 5  using namespace std; 6 7  int main () { 8      cout &lt;&lt; "Hello World! "; // 9      prints Hello World! 10     cout &lt;&lt; "I'm a C++ program"; // 11     prints I'm a C++ program 12     return 0; 13 }</pre>	<pre>Hello World! I'm a C++ program</pre>
--	---

Se você incluir comentários dentro do código fonte em seus programas sem usar os caracteres de comentário // ou /\* e \*/, o compilador irá considerá-los como se fossem expressões de C++ e, provavelmente, irá causar uma ou várias mensagens de erro quando você compilar.

## 1.2 Variáveis. Tipo de dados.

A utilidade dos programas "Hello World!" mostrados na seção anterior é bastante questionável. Tínhamos que escrever várias linhas de código, compilá-los, e depois executar o programa resultante apenas para obter uma simples frase escrita na tela como resultado. Certamente teria sido muito mais rápido digitar a frase de saída nós mesmos. No entanto, a programação não se limita apenas à impressão simples de textos na tela. Para ir um pouco mais adiante e tornarmos capazes de escrever programas que executam tarefas úteis que realmente irão nos poupar trabalho é necessário introduzir o conceito de variável.

Vamos imaginar que eu peça que você mantenha o número 5 em sua memória, e então eu peça para você memorizar também o número dois, ao mesmo tempo. Você tem apenas dois valores armazenados em sua memória. Agora, se eu pedir para você adicionar 1 ao primeiro número que eu disse, você deve manter o número 6 (que é  $5 + 1$ ) e 2 em sua memória. Valores que nós poderíamos agora, por exemplo, subtrair e obter 4 como resultado.

O processo que você fez apenas com sua memória é semelhante ao que um computador pode fazer com duas variáveis. O mesmo processo pode ser expresso em C++ com a seguinte instrução:

```
1 a = 5;  
2 b = 2;  
3 a = a + 1;  
4 result = a - b;
```

Obviamente, este é um exemplo muito simples, uma vez que só tenho usado dois valores inteiros pequenos, mas considere que o computador pode armazenar milhões de números como estes, ao mesmo tempo sofisticado e realizar operações matemáticas com elas.

Portanto, podemos definir uma variável como uma parte da memória do computador destinada a armazenar um determinado valor.

Cada variável precisa de um identificador que o distingue dos demais. Por exemplo, no código anterior os identificadores de variáveis eram `a`, `b` e `result`. No entanto, poderíamos ter atribuído qualquer nome às variáveis, desde que sejam identificadores válidos.

### 1.2.1 Identificadores

Um identificador válido em C++ é uma sequência de uma ou mais letras, dígitos ou caracteres de sublinhado (`_`). Nem espaços nem sinais de pontuação ou símbolos pode ser parte de um identificador. Somente letras, números e caracteres de sublinhado simples são válidos. Além disso, os identificadores de variável sempre tem que começar com uma letra ou caractere sublinhado (`_`), mas em alguns casos, estes podem ser reservados para determinadas palavras-chave do compilador ou identificadores externos, bem como identificadores contendo dois sucessivos caracteres de sublinhado em qualquer lugar. Em nenhum caso, um identificador pode começar com um dígito.

Outra regra que você deve considerar ao inventar seus próprios identificadores é que eles não podem corresponder a qualquer palavra-chave da linguagem C++ ou do compilador, que são palavras reservadas consideradas especiais. As palavras-chave reservadas padrão são:

```
asm, auto, bool, break, case, catch, char, class, const, const_cast,  
continue, default, delete, do, double, dynamic_cast, else, enum,  
explicit, export, extern, false, float, for, friend, goto, if, inline,
```

`int, long, mutable, namespace, new, operator, private, protected, public, register, reinterpret_cast, return, short, signed, sizeof, static, static_cast, struct, switch, template, this, throw, true, try, typedef, typeid, typename, union, unsigned, using, virtual, void, volatile, wchar_t, while`

Além disso, representações alternativas para alguns operadores não podem ser usadas como identificadores, uma vez que são palavras reservadas em algumas circunstâncias:

`and, and_eq, bitand, bitor, compl, not, not_eq, or, or_eq, xor, xor_eq`

Seu compilador pode também incluir algumas palavras-chave específicas adicionais.

**MUITO IMPORTANTE:** A linguagem C++ é "case sensitive". Isso significa que um identificador escrito em letras maiúsculas não é equivalente a outro com o mesmo nome mas escrito em letras minúsculas. Assim, por exemplo, a variável `RESULT` não é a mesma que a variável `result` ou a variável `Result`. São três identificadores de variáveis diferentes.

## 1.2.2 Tipos fundamentais de dados

Durante a programação, nós armazenamos as variáveis na memória do computador, no entanto, o computador tem que saber o tipo de dados que deseja armazenar, pois ele não vai alocar a mesma quantidade de memória para armazenar um simples número, única letra ou um grande número. Eles não vão ser interpretados da mesma maneira.

A memória dos computadores é organizada em bytes. Um byte é a quantidade mínima de memória que pode ser gerenciada. Um byte pode armazenar uma quantidade relativamente pequena de dados: um único caractere ou um inteiro pequeno (geralmente um número inteiro entre 0 e 255). Além disso, o computador pode manipular dados muito mais complexos, agrupando vários bytes, como números longos ou não-inteiros.

Em seguida você tem um resumo dos tipos de dados básicos fundamentais em C++, bem como o intervalo de valores que podem ser representados com cada um:

Name	Description	Size*	Range*
<code>char</code>	Character or small integer.	1byte	signed: -128 to 127 unsigned: 0 to 255
<code>short int</code> ( <code>short</code> )	Short Integer.	2bytes	signed: -32768 to 32767 unsigned: 0 to 65535
<code>int</code>	Integer.	4bytes	signed: -2147483648 to 2147483647 unsigned: 0 to 4294967295
<code>long int</code> ( <code>long</code> )	Long integer.	4bytes	signed: -2147483648 to 2147483647 unsigned: 0 to 4294967295
<code>bool</code>	Boolean value. It can take one of two values: true or false.	1byte	true or false
<code>float</code>	Floating point number.	4bytes	+/- 3.4e +/- 38 (~7 digits)
<code>double</code>	Double precision floating point number.	8bytes	+/- 1.7e +/- 308 (~15 digits)
<code>long double</code>	Long double precision floating point number.	8bytes	+/- 1.7e +/- 308 (~15 digits)
<code>wchar_t</code>	Wide character.	2 or 4 bytes	1 wide character

\* Os valores das colunas *Size* e *Range* dependem do sistema em que o programa é compilado. Os valores apresentados acima são os encontrados na maioria dos sistemas de 32 bits. Mas para outros sistemas, a especificação geral é que `int` tem o tamanho natural sugerido pela arquitetura do sistema (uma "*palavra*") e os quatro tipos de inteiros `char`, `short`, `int` e `long` devem ser de cada um, pelo menos, tão grande quanto o anterior. No entanto, o `char` será sempre de um byte de tamanho. O mesmo se aplica aos tipos de ponto flutuante `float`, `double` e `long double`, onde cada um deve fornecer pelo menos tanta precisão como o anterior.

**NOTA:** Para variáveis do tipo complexo veja o anexo A.2.

### 1.2.3 Declaração das variáveis

A fim de usar uma variável em C++, devemos primeiro declará-la especificando que tipo de dados que queremos armazenar. A sintaxe para declarar uma nova variável é escrever o especificador do tipo de dados desejado (como, `bool`, `int`, `float` ...) seguido por um identificador de variável válido. Por exemplo:

```
1 int a;  
2 float mynumber;
```

Estas são duas declarações válidas de variáveis. O primeiro declara uma variável do tipo `int` com o identificador `a`. O segundo declara uma variável do tipo `float` com o identificador `mynumber`. Uma vez declaradas, as variáveis `a` e `mynumber` podem ser usadas no resto do programa.

Se você deseja declarar mais de uma variável do mesmo tipo, você pode declarar todas elas em uma única instrução, separando seus identificadores com vírgulas. Por exemplo:

```
int a, b, c;
```

Este código declara três variáveis (`a`, `b` e `c`), todas do tipo `int`, e tem exatamente o mesmo significado que:

```
1 int a;  
2 int b;  
3 int c;
```

Aos tipos de dados inteiros `char`, `short`, `long` e `int` podem ser atribuídos os especificadores *signed* (com sinal) ou *unsigned* (sem sinal). Se atribuirmos o especificador *signed*, a uma determinada variável, esta poderá representar tanto valores positivos quanto negativos. Se atribuirmos o especificador *unsigned* somente valores positivos (e zero) serão permitidos. Ambos os especificadores (*signed* ou *unsigned*) devem ser declarados antes do nome do tipo da variável. Por exemplo:

```
1 unsigned short int NumberOfSisters;  
2 signed int MyAccountBalance;
```

Por padrão, se não for especificado `signed` ou `unsigned` a uma variável, o compilador assumirá o tipo `signed`. Portanto poderíamos ter escrito a segunda declaração acima assim:

```
int MyAccountBalance;
```

com exatamente o mesmo significado (com ou sem a palavra-chave `signed`)

Uma exceção a essa regra geral é o tipo de dado `char`, que existe por si só e é considerado outro tipo de dado fundamental, diferente de `signed char` e `unsigned char`, destinada a armazenar caracteres. Se você pretende armazenar valores numéricos em uma variável do tipo `char` deve usar os especificadores `signed` ou `unsigned`.

`short` e `long` podem ser usados sozinhos como especificadores de tipo. Neste caso, eles se referem a seus respectivos tipos inteiros: `short` é equivalente a `short int` e `long` é equivalente a `long int`. As duas seguintes declarações de variáveis são equivalentes:

```
1 short Year;
2 short int Year;
```

Finalmente, `signed` e `unsigned` também podem ser usados como especificadores de tipo, e possuem o mesmo significado que `signed int` e `unsigned int`, respectivamente. As duas seguintes declarações são equivalentes:

```
1 unsigned NextYear;
2 unsigned int NextYear;
```

Para ver como as declarações de variáveis aparecem dentro de um programa, vamos observar o código C++ a seguir, sobre o exemplo da memória proposto no início desta seção:

```
1 // operating with variables
2
3 #include <iostream>
4 using namespace std;
5
6 int main (){
7     // declaring variables:
8     int a, b;
9     int result;
10
11     // process:
12     a = 5;
13     b = 2;
14     a = a + 1;
15     result = a - b;
16
17     // print out the result:
18     cout << result;
19
20     // terminate the program:
21     return 0;
22 }
```

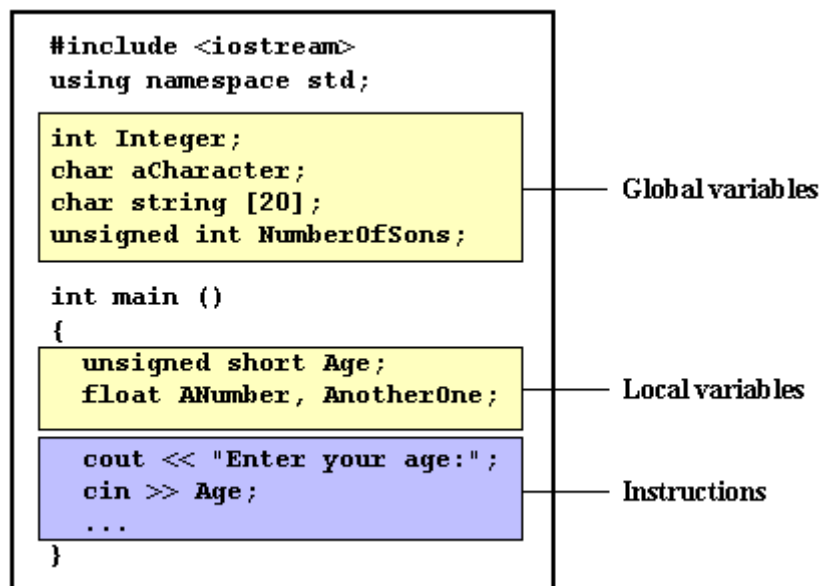
4

Não se preocupe se algumas das declarações, além das variáveis, parecem um pouco estranhas para você. Você vai ver o restante das definições em detalhes nas próximas seções.

## 1.2.4 Escopo das variáveis

Todas as variáveis que pretendemos usar em um programa devem ter sido declaradas com um especificador de tipo em algum ponto anterior no código, como fizemos no código anterior, no início do corpo da função *main*, quando declaramos *a*, *b*, e *result* do tipo *int*.

Uma variável pode ser local ou global. Uma variável global é uma variável declarada no corpo principal do código-fonte, fora de todas as funções, enquanto uma variável local é declarada dentro do corpo de uma função específica ou um bloco.



As variáveis globais podem ser referidas em qualquer lugar do código (dentro de funções) sempre após a sua declaração.

O escopo das variáveis locais é limitado ao bloco entre chaves (`{}`) onde elas são declaradas. Por exemplo, se elas são declaradas no início do corpo de uma função (como na função *main*), seu alcance é entre a sua declaração e final dessa função. No exemplo acima, isto significa que, se existisse outra função além da *main*, as variáveis locais declaradas dentro da função *main* não poderiam ser acessadas dentro da outra função e vice-versa.

## 1.2.5 Inicialização das Variáveis

Quando declaramos uma variável, seu valor é, por padrão, indeterminado. Mas você pode querer uma variável para armazenar um valor no mesmo momento em que é declarada. Para fazer isso, você pode inicializar a variável. Há duas maneiras de fazer isso em C++:

O primeiro, conhecido como *c- inicialização*, é realizada acrescentando um sinal de igual seguido do valor para o qual a variável será inicializada:

```
type identifier = initial_value ;
```

Por exemplo, se quisermos declarar uma variável do tipo *int*, denominada *a*, e inicializada com um valor de 0, no momento em que for declarada, podemos escrever:

```
int a = 0;
```

A outra maneira de inicializar variáveis, conhecido como *inicialização do construtor*, é realizada colocando o valor inicial entre parênteses ( *()* ):

```
type identifier (initial_value) ;
```

Por exemplo:

```
int a (0);
```

Ambas as maneiras de inicializar as variáveis são válidas e equivalentes em C++.

```
1 // initialization of variables
2
3 #include <iostream>
4 using namespace std;
5
6 int main () {
7     int a=5;    // initial value = 5
8     int b(2);   // initial value = 2
9     int result; // initial value undetermined
10    a = a + 3;
11    result = a - b;
12    cout << result;
13
14    return 0;
15 }
```

6

## 1.2.6 Introdução às cadeias de caracteres

Variáveis que podem armazenar valores não-numéricos que são mais do que um caractere único são conhecidos como cadeias de caracteres.

A biblioteca da linguagem C++ oferece suporte para a utilização de cadeias de caracteres através da classe padrão *string* (veja o anexo A.7). Este não é um tipo fundamental, mas ele se comporta de forma semelhante aos tipos fundamentais, o que simplifica sua utilização.

A primeira diferença com tipos de dados fundamentais é que, a fim de declarar e utilizar objetos (variáveis) desse tipo é preciso incluir um arquivo de cabeçalho adicional em nosso código fonte: *<string>* e ter acesso ao espaço de nomes *std* (que já tínhamos em todos os nossos programas anteriores graças ao uso do *using namespace std*).

```
1 // my first string
2 #include <iostream>
3 #include <string>
4 using namespace std;
```

This is a string

```

5
6 int main (){
7     string mystring = "This is a
string";
8     cout << mystring;
9     return 0;
10 }

```

Como você pode ver no exemplo anterior, `strings` podem ser inicializados com qualquer cadeia de caracteres válida, como variável do tipo numérica pode ser inicializada com qualquer valor numérico válido. Ambos os formatos são válidos de inicialização em `strings`:

```

1 string mystring = "This is a string";
2 string mystring ("This is a string");

```

Strings também podem executar todas as operações básicas de outros tipos de dados fundamentais como ser declarada sem um valor inicial e atribuir valores durante a execução:

<pre> 1 // my first string 2 #include &lt;iostream&gt; 3 #include &lt;string&gt; 4 using namespace std; 5 6 int main (){ 7     string mystring; 8     mystring = "This is the initial string content"; 9     cout &lt;&lt; mystring &lt;&lt; endl; 10    mystring = "This is a different string content"; 11    cout &lt;&lt; mystring &lt;&lt; endl; 12    return 0; 13 } </pre>	<pre> This is the initial string content This is a different string content </pre>
---	--

## 1.3 Constantes

*Constantes* são expressões que possuem um valor fixo.

### 1.3.1 Literais

Literais são os tipos mais óbvios de constantes. Eles são usados para expressar valores específicos dentro do código fonte do programa. Nós já usamos os literais anteriormente para dar valores às variáveis ou para expressar as mensagens que queríamos que nosso programa imprimisse, por exemplo, quando escrevemos:

```
a = 5;
```

o 5, neste pedaço de código. é uma **constante literal**.



Constantes literais podem ser divididas em **Numerais Inteiros**, **Numerais de Ponto Flutuante**, **Caracteres**, **Strings** e **Valores Booleanos**.

## Numerais inteiros

```
1 1776
2 707
3 -273
```

Eles são constantes numéricas que identificam valores decimais inteiros. Repare que para expressar uma constante numérica não temos que escrever aspas (") nem qualquer caractere especial. Não há dúvida de que representamos uma constante ao escrever 1776 em um programa, estaremos nos referindo ao valor 1776.

Além de números decimais (aqueles que todos nós estamos acostumados a usar todos os dias), C++ permite o uso de números octais (**base 8**) e números hexadecimais (**base 16**) como constantes literais. Se quisermos expressar um número octal temos de precedê-lo com um 0 (um caractere **zero**). E a fim de expressar um número hexadecimal temos de precedê-lo com os caracteres 0x (**zero, x**). Por exemplo, as seguintes constantes literais são todas equivalentes entre si:

```
1 75          // decimal
2 0113        // octal
3 0x4b        // hexadecimal
```

Todos estes representam o número 75 (setenta e cinco), expresso como uma base-10, octal e hexadecimal, respectivamente.

Constantes literais, como as variáveis, são considerados como um tipo de dado específico. Por padrão, literais inteiros são do tipo int. No entanto, pode forçar o sistema a ser signed, anexando o caráter *u* a ele, ou long, acrescentando *l*:

```
1 75          // int
2 75u         // unsigned int
3 75l         // long
4 75ul        // unsigned long
```

Em ambos os casos, o sufixo pode ser especificado usando tanto letras maiúsculas quanto minúsculas.

## Números de ponto flutuante

Eles expressam números com casas decimais e/ou expoentes. Eles podem incluir um ponto decimal, o caractere e (que expressa "dez elevado a X", onde X é um valor inteiro que segue o caractere e), ou ambos um ponto decimal e o caractere e:

```
1 3.14159     // 3.14159
2 6.02e23     // 6.02 x 10^23
3 1.6e-19     // 1.6 x 10^-19
4 3.0         // 3.0
```

Na figura acima são expressos quatro números decimais válidos em C++. O primeiro número é PI, o segundo é o número de Avogadro, o terceiro é a carga elétrica de um elétron (um número extremamente pequeno) – todos valores aproximados – e o último é o número três, expresso como um ponto flutuante literal numérico. OBS: a constante PI pode ser facilmente calculada por meio da seguinte expressão  $PI = 4 * \text{atan}(1.0)$ , em que `atan` é o arco tangente.

O tipo padrão para literais de ponto flutuante é o `double`. Se você quer expressar explicitamente um número como `float` ou `long double`, você pode usar os sufixos `f` ou `l`, respectivamente:

```
1 3.14159L    // long double
2 6.02e23f    // float
```

Qualquer das letras que podem ser parte de uma constante de ponto flutuante (`e`, `f`, `l`) pode ser escrito usando letras maiúsculas ou minúsculas, quer sem qualquer diferença em seus significados e/ou interpretação pelo compilador.

## Caracteres e literais do tipo *string*

Existem também as constantes não numéricas, como:

```
1 'z'
2 'p'
3 "Hello world"
4 "How do you do?"
```

As duas primeiras expressões representam constantes de um único caractere, e os dois seguintes representam cadeias de literais compostas por vários caracteres. Observe que para representar um único caractere nos devemos colocá-lo entre aspas simples (`'`) e para expressar uma sequência de caracteres (que geralmente consiste em mais de um caractere) devemos colocá-lo entre aspas duplas (`"`).

Ao escrever ambos literais alfanuméricos, simples ou sequência de caracteres, é necessário inseri-los entre aspas para distingui-los dos identificadores de possíveis variáveis ou palavras-chave reservadas. Observe a diferença entre essas duas expressões:

```
1 x
2 'x'
```

Na primeira linha `x` (sem aspas) remete para uma variável cujo identificador é `x`, enquanto que `'x'` (entre aspas simples) remete para o carácter constante `'x'`.

Caracteres e *strings* têm certas peculiaridades, como os códigos de escape. Estes são caracteres especiais que são difíceis ou impossíveis de expressar de outra forma no código fonte de um programa, como nova linha (`\n`) ou tabulação (`\t`). Todos eles são precedidos por uma barra invertida (`\`) ao código. Aqui você tem uma lista de alguns:

<code>\n</code>	newline
<code>\r</code>	carriage return
<code>\t</code>	Tab
<code>\v</code>	vertical tab
<code>\b</code>	backspace

\f	form feed (page feed)
\a	alert (beep)
\'	single quote (')
\"	double quote (")
\?	question mark (?)
\\	backslash (\)

Por exemplo:

```
1 '\n'
2 '\t'
3 "Left \t Right"
4 "one\ntwo\nthree"
```

Além disso, você pode expressar qualquer caractere através de seu código ASCII, escrito por meio de uma barra invertida (\) seguido do código ASCII expresso como um octal (base 8) ou hexadecimal (base 16). No primeiro caso (octal) os dígitos devem seguir imediatamente a barra invertida (por exemplo, \ 23 ou \ 40), no segundo caso (hexadecimal), um caractere X deve ser escrito antes dos dígitos (por exemplo, \ x20 ou \ x4A ).

Strings podem ser entendidas como mais de uma linha de código ao se colocar uma barra invertida (\) no final de cada linha que se deseja continuar.

```
1 "string expressed in \
2 two lines"
```

Você pode também concatenar várias cadeias de caracteres separando-as por um ou vários espaços em branco, tabulações, quebra de linha ou de qualquer outro caractere branco válido:

```
"this forms" "a single" "string" "of characters"
```

## Literais booleanas

Há apenas dois valores booleanos válidos: o verdadeiro e o falso. Estes podem ser expressos em C++ como valores do tipo `bool`, utilizando os literais booleanos `true` e `false`.

### 1.3.2 Constantes definidas (#define)

Você pode definir seus próprios nomes para constantes que você usa muito frequentemente sem ter que recorrer a variáveis, e, conseqüentemente, consumo de memória, simplesmente usando a diretiva de pré-processamento *# define*. Seu formato é:

```
#define identifier value
```

Por exemplo:

```
1 #define PI 3.14159
```

```
2 #define NEWLINE '\n'
```

Isso define duas novas constantes: PI e NEWLINE. Uma vez definidas, você pode usá-las no restante do código como se fosse qualquer outra constante, por exemplo:

```
1 // defined constants: calculate 31.4159
2 circumference
3
4 #include <iostream>
5 using namespace std;
6
7 #define PI 3.14159
8 #define NEWLINE '\n'
9
10 int main () {
11     double r=5.0;    // radius
12     double circle;
13
14     circle = 2 * PI * r;
15     cout << circle;
16     cout << NEWLINE;
17
18     return 0;
19 }
```

Na verdade a única coisa que o pré-processador do compilador faz quando encontra a diretiva #define é, literalmente, substituir qualquer ocorrência de seu identificador (no exemplo anterior, estes eram PI e NEWLINE) pelo código para o qual foram pré-definidos (3,14159 e ' \ n ', respectivamente).

A diretiva #define não é uma diretiva do C++, mas uma diretiva para o pré-processador, desta forma, como uma diretiva, não necessita de ponto e vírgula (;) ao final da linha. Se você acrescentar um caractere vírgula (;) no final, ele também será anexado em todas as ocorrências do identificador no corpo do programa que o pré-processador substituir.

### 1.3.3 Constantes declaradas (const)

Com o prefixo const você pode declarar constantes com um tipo específico da mesma forma como você faria com uma variável:

```
1 const int pathwidth = 100;
2 const char tabulator = '\t';
```

Aqui, pathwidth e tabulator são duas constantes. Eles são tratados como variáveis regulares, exceto que seus valores não podem ser modificados após a sua definição.

## 1.4 Operadores

Uma vez que sabemos da existência de variáveis e constantes, podemos começar a realizar operações com eles. Para isto, a linguagem C++ integra alguns operadores. Diferente de outras

linguagens cujos operadores são principalmente palavras-chave, operadores em C++ são feitos principalmente de sinais que não são parte do alfabeto, mas estão disponíveis em todos os teclados. Isto torna os códigos escritos C++ menores e mais flexíveis, já que se baseia menos em palavras em Inglês, mas exige um pouco de esforço de aprendizagem no início.

Você não tem de memorizar todo o conteúdo desta página. A maioria dos detalhes é fornecida apenas para servir como uma referência futura no caso de você precisar dele.

### 1.4.1 Atribuição (=)

O operador de atribuição atribui um valor a uma variável.

```
a = 5;
```

Esta afirmação atribui o valor inteiro 5 à variável *a*. A parte do lado esquerdo do operador de atribuição (=) é conhecida como o *lvalue* (valor à esquerda) e o da direita como *rvalue* (valor à direita). O *lvalue* tem que ser uma variável enquanto o *rvalue* pode ser uma constante, uma variável, o resultado de uma operação ou qualquer combinação destes. A informação mais importante nesta etapa é a regra de atribuição: A operação de atribuição ocorre sempre da direita para a esquerda e nunca de outra maneira:

```
a = b;
```

Este código atribui a variável *a* (o *lvalue*) o valor contido na variável *b* (o *rvalue*). O valor que foi armazenado, até este momento na variável *a* não é considerado na operação, e, na verdade, o possível valor armazenado em *a* após a operação é perdido.

Considere também que estamos atribuindo o valor de *b* para *a* somente no momento da operação de atribuição. Portanto, uma mudança posterior em *b* não afetará o valor de *a*.

Por exemplo, vamos olhar o código a seguir – foi incluído o conteúdo armazenado nas variáveis assim como comentários:

<pre>1 // assignment operator 2 3 #include &lt;iostream&gt; 4 using namespace std; 5 6 int main () { 7     int a, b;           // a:?, b:? 8     a = 10;             // a:10, b:? 9     b = 4;              // a:10, b:4 10    a = b;               // a:4, b:4 11    b = 7;               // a:4, b:7 12 13    cout &lt;&lt; "a:"; 14    cout &lt;&lt; a; 15    cout &lt;&lt; " b:"; 16    cout &lt;&lt; b; 17 18    return 0; 19 }</pre>	<pre>a:4 b:7</pre>
--	--------------------

Esse código nos dará como resultado que o valor contido em `a` é 4 e em `b` é 7. Observe como `a` não foi afetado pela modificação final de `b`, uma vez que a atribuição (`a = b`) foi declarada anteriormente.

Uma propriedade que C++ tem sobre outras linguagens de programação é que a operação de atribuição pode ser utilizada como `rvalue` (ou parte de um `rvalue`) de outra operação de atribuição. Por exemplo:

```
a = 2 + (b = 5);
```

É equivalente a:

```
1 b = 5;  
2 a = 2 + b;
```

que significa: primeiro atribuir 5 a variável `b` e depois atribuir a `a` o valor 2, mais o resultado da atribuição de `b` (ou seja, 5), deixando `a` com um valor final de 7.

A expressão a seguir também é válida em C++:

```
a = b = c = 5;
```

Ele atribui o valor 5 às três variáveis: `a`, `b` e `c`.

## 1.4.2 Operadores aritméticos ( +, -, \*, /, % )

As cinco operações aritméticas suportadas pela linguagem C++ são:

+	adição
-	Subtração
*	Multiplicação
/	Divisão
%	Resto

As operações de adição, subtração, multiplicação e divisão correspondem literalmente com seus respectivos operadores matemáticos. O único operador que pode não parecer muito familiar é o operador resto ( `%` ). O operador resto retorna o resto da divisão de dois valores e, portanto, só deve ser utilizado com valores inteiros. Por exemplo, se nós escrevemos:

```
a = 11 % 3;
```

na variável `a` irá conter 2, uma vez que 2 é o resto da divisão entre 11 e 3.

### 1.4.3 Atribuição composta (+=, -=, \*=, /=, %=, >>=, <<=, &=, ^=, |=)

Quando queremos modificar o valor de uma variável realizando uma operação no valor atualmente armazenado nessa variável, podemos usar os operadores de atribuição composta:

expression	is equivalent to
value += increase;	value = value + increase;
a -= 5;	a = a - 5;
a /= b;	a = a / b;
price *= units + 1;	price = price * (units + 1);

o procedimento é o mesmo para todos os operadores. Por exemplo:

```
1 // compound assignment operators
2
3 #include <iostream>
4 using namespace std;
5
6 int main () {
7     int a, b=3;
8     a = b;
9     a+=2;      // equivalent to a=a+2
10    cout << a;
11    return 0;
12 }
```

5

### 1.4.4 Incremento e decremento (++ , --)

Os operadores incremento (++) e decremento (--) aumentam e reduzem em uma unidade o valor armazenado em uma variável. Eles são equivalentes a +=1 e -=1 , respectivamente. Assim:

```
1 c++;
2 c+=1;
3 c=c+1;
```

são todos equivalentes em sua funcionalidade: aumentar de uma unidade o valor de c.

Nos primeiros compiladores C, as três expressões anteriores provavelmente produziam códigos executáveis diferentes dependendo de qual expressão foi usada. Atualmente, este tipo de otimização de código é geralmente realizada automaticamente pelo compilador, portanto, as três expressões devem produzir exatamente o mesmo código executável.

Uma característica desse operador é que pode ser usado tanto como um prefixo quanto como um sufixo. Isso significa que podem ser escritos antes do identificador da variável (++a) ou depois dele (a++). Embora em expressões simples como a++ ou ++a ambos têm exatamente o mesmo significado, em outras expressões, concomitantemente com o operador de atribuição, o resultado da operação incremento ou decremento utilizado como prefixo ou sufixo representam uma sensível diferença: no caso que o operador incremento é utilizado como um prefixo (++a) o valor é incrementado antes de ser atribuído. No caso em que é usado como um sufixo (a++) o valor é incrementado depois de ser atribuído. Observe a diferença:

Example 1	Example 2
<pre>B=3; A=++B; // A contains 4, B contains 4</pre>	<pre>B=3; A=B++; // A contains 3, B contains 4</pre>

No Exemplo 1, B é aumentado antes que seu valor seja copiado para A. Enquanto no Exemplo 2, o valor de B é copiado para A e, em seguida, B é aumentado.

### 1.4.5 Operadores relacionais e de igualdade ( ==, !=, >, <, >=, <= )

Para realizar uma comparação entre duas expressões devemos utilizar os operadores relacionais ou de igualdade. O resultado de uma operação relacional é um valor booleano que só pode ser verdadeiro ou falso, de acordo com seu resultado booleano.

Se quisermos comparar duas expressões, por exemplo, para saber se elas são iguais, ou se uma é maior do que outra, devemos utilizar estes operadores. Aqui está uma lista dos operadores relacionais e de igualdade que podem ser usados em C++:

==	Equal to
!=	Not equal to
>	Greater than
<	Less than
>=	Greater than or equal to
<=	Less than or equal to

Aqui estão alguns exemplos:

```
1 (7 == 5)      // evaluates to false.
2 (5 > 4)       // evaluates to true.
3 (3 != 2)      // evaluates to true.
4 (6 >= 6)      // evaluates to true.
5 (5 < 5)       // evaluates to false.
```

É claro que, em vez de usar somente constantes numéricas, podemos usar qualquer expressão válida, incluindo variáveis. Suponha que a=2, b=3 e c=6,

```
1 (a == 5)      // evaluates to false since a is not equal to 5.
2 (a*b >= c)    // evaluates to true since (2*3 >= 6) is true.
3 (b+4 > a*c)    // evaluates to false since (3+4 > 2*6) is false.
4 ((b=2) == a)  // evaluates to true.
```

**CUIDADO!** O operador = (um sinal de igualdade) não é o mesmo que o operador == (dois sinais de igual), o primeiro é um operador de atribuição (atribui o valor de seu direito à variável à sua esquerda) e o outro (==) é o operador de igualdade que compara se as duas expressões em ambos lados de uma expressão são iguais. Assim, na última expressão ((b = 2) == a), primeiro atribui o valor 2 a b e então compararam a a, que também armazenava o valor 2, então o resultado da operação é *true*.



### 1.4.6 Operadores lógicos ( !, &&, || )

O operador `!` é o operador C++ que realiza a operação booleana NOT. Ele tem apenas um operando, localizado a sua direita, e a única coisa que ele faz é retornar o inverso dele, produzindo *false* se seu operando é *true* e *true* se seu operando é *false*. Basicamente, ele retorna o valor booleano oposto de seu operando. Por exemplo:

```
1 !(5 == 5)    // evaluates to false because the expression at its right
   (5 == 5) is true.
2 !(6 <= 4)    // evaluates to true because (6 <= 4) would be false.
3 !true        // evaluates to false
4 !false       // evaluates to true.
```

Os operadores lógicos `&&` e `||` são usados na avaliação de duas expressões para obter um único resultado relacional. O operador `&&` corresponde à operação lógica booleana AND. Esta operação resulta *true* se ambos os seus dois operandos são verdadeiros e *false* caso contrário. O quadro a seguir mostra o resultado do operador `&&`, resultado da avaliação da expressão `a && b`:

#### && OPERATOR

a	b	a && b
true	true	True
true	false	False
false	true	False
false	false	False

O operador `||` corresponde à operação lógica OR. Esta operação resulta *true* se qualquer um de seus dois operandos é verdadeiro, sendo *false* somente quando ambos os operandos são falsos. Aqui estão os resultados possíveis de `a || b`:

#### || OPERATOR

a	b	a    b
true	true	true
true	false	true
false	true	true
false	false	false

Por exemplo:

```
1 ( (5 == 5) && (3 > 6) ) // evaluates to false ( true && false ).
2 ( (5 == 5) || (3 > 6) ) // evaluates to true ( true || false ).
```

### 1.4.7 Operador condicional ( ? )

O operador condicional avalia uma expressão retornando um valor se essa expressão é verdadeira e outro valor se a expressão é falsa. Seu formato é:

```
condition ? result1 : result2
```

Se condition é verdadeira o operador retornará result1, caso contrário result2.

```
1 7==5 ? 4 : 3    // returns 3, since 7 is not equal to 5.
2 7==5+2 ? 4 : 3  // returns 4, since 7 is equal to 5+2.
3 5>3 ? a : b     // returns the value of a, since 5 is greater than 3.
4 a>b ? a : b     // returns whichever is greater, a or b.
```

```
1 // conditional operator
2
3 #include <iostream>
4 using namespace std;
5
6 int main (){
7     int a,b,c;
8
9     a=2;
10    b=7;
11    c = (a>b) ? a : b;
12
13    cout << c;
14
15    return 0;
16 }
```

7

Neste exemplo, a é igual a 2 e b igual a 7. A expressão a ser avaliada era (a>b) e o resultado foi verdadeiro, desta forma, o valor especificado após o ponto de interrogação foi descartado em favor do segundo valor (depois dos dois pontos), que era b, e possuía valor igual a 7.

### 1.4.8 Operador virgula ( , )

O operador vírgula (,) é usado para separar duas ou mais expressões que são incluídas onde somente uma expressão é esperada. Quando o conjunto de expressões tem que ser avaliado para um valor, apenas a expressão mais à direita é considerada.

Por exemplo, o seguinte código:

```
a = (b=3, b+2);
```

Em primeiro lugar, atribui-se o valor 3 a b, e depois b+2 à variável a. Assim, no final, a variável a contém o valor 5, enquanto b variável contém o valor 3.

### 1.4.9 Operadores bit a bit ( &, |, ^, ~, <<, >> )

Os operadores bit a bit modificam as variáveis (inteiras) considerando os valores que elas representam no sistema binário.

operator	asm equivalent	description
&	AND	Bitwise AND
	OR	Bitwise Inclusive OR
^	XOR	Bitwise Exclusive OR
~	NOT	Unary complement (bit inversion)
<<	SHL	Shift Left
>>	SHR	Shift Right

### 1.4.10 Operadores de conversão explícita de tipo

Os operadores de conversão explícita permitem converter um dado de um determinado tipo para outro. Existem várias maneiras de fazer isso em C++. No mais simples, herdado da linguagem C, a expressão a ser convertida deve suceder o novo tipo colocado entre parênteses ( ).

```
1 int i;
2 float f = 3.14;
3 i = (int) f;
```

O código anterior converte o número float 3.14 para um valor inteiro (3), o restante é perdido. Aqui, o operador de conversão é o (int). Outra maneira de fazer a mesma coisa em C++ é usando a notação funcional: precedendo a expressão a ser convertida pelo tipo e colocando a expressão entre parênteses:

```
i = int ( f );
```

Ambas as formas de conversão são válidas em C++.

### 1.4.11 sizeof()

Esse operador aceita um parâmetro que pode ser tanto um tipo ou uma variável em si e retorna o tamanho em bytes desse tipo ou objeto:

```
a = sizeof (char);
```

Este código vai atribuir o valor 1 à *a*, porque *char* é um tipo de variável com tamanho de um byte. O valor retornado por *sizeof* é uma constante, por isso é sempre determinado antes da execução do programa.

### 1.4.12 Outros operadores

Mais tarde, nesta apostila, vamos ver mais alguns operadores, referentes a ponteiros ou específicos à programação orientada a objeto. Cada um é tratado em sua respectiva seção.

### 1.4.13 Precedência dos operadores

Ao escrever expressões complexas com vários operandos, podemos ter algumas dúvidas sobre qual operando é avaliado primeiro e qual posteriormente. Por exemplo, nesta expressão:

```
a = 5 + 7 % 2
```

podemos ficar em dúvida se ele realmente significa:

```
1 a = 5 + (7 % 2)    // with a result of 6, or
2 a = (5 + 7) % 2    // with a result of 0
```

A resposta correta é a primeira das duas expressões, com um resultado de 6. Há uma ordem estabelecida com a prioridade de cada operador, e não apenas os aritméticos (aqueles cuja preferência vem da matemática), mas para todos os operadores que podem aparecer em C++. Da maior para a menor prioridade, a ordem é a seguinte:

Nível	Operador	Descrição	Grouping
1	::	scope	Left-to-right
2	() [] . -> ++ -- dynamic_cast static_cast reinterpret_cast const_cast typeid	postfix	Left-to-right
3	++ -- ~ ! sizeof new delete	unary (prefix)	Right-to-left
	* &	indirection and reference (pointers)	
	+ -	unary sign operator	
4	(type)	type casting	Right-to-left
5	. * -> *	pointer-to-member	Left-to-right
6	* / %	multiplicative	Left-to-right
7	+ -	additive	Left-to-right
8	<< >>	shift	Left-to-right
9	< > <= >=	relational	Left-to-right
10	== !=	equality	Left-to-right
11	&	bitwise AND	Left-to-right
12	^	bitwise XOR	Left-to-right
13		bitwise OR	Left-to-right
14	&&	logical AND	Left-to-right
15		logical OR	Left-to-right
16	?:	conditional	Right-to-left
17	= *= /= %= += -= >>= <<= &= ^=  =	assignment	Right-to-left
18	,	comma	Left-to-right

“Grouping” define a ordem de precedência em situações cujos operadores avaliados possuam o mesmo nível em uma determinada expressão.

Todos esses níveis de precedência dos operadores podem ser manipulados ou se tornar mais legíveis, eliminando possíveis ambiguidades usando sinais entre parênteses ( ), como neste exemplo:

```
a = 5 + 7 % 2;
```

pode ser escrito como:

```
a = 5 + (7 % 2);
```

ou

```
a = (5 + 7) % 2;
```

dependendo da operação que deseja executar.

Então, se você quiser escrever expressões complicadas e você não está completamente certo dos níveis de prioridade, sempre inclua parênteses. Ele também fará seu código mais fácil de ler.

# Capítulo 2: Estruturas de controle

## 2.1 Estruturas de controle

Um programa é geralmente não limitado a uma sequência linear de instruções. Durante o seu processo, podem se bifurcar, repetir o código, ou tomar decisões. Para esse propósito, a linguagem C++ oferece estruturas de controle que servem para especificar o que deve ser feito pelo nosso programa, quando e sob quais circunstâncias.

Com a introdução de estruturas de controle nós vamos ter que introduzir um novo conceito: o bloco de instrução. Um bloco de instruções é um conjunto de instruções que são separados por ponto e vírgula (;) como todas as instruções em C++, mas agrupados entre chaves: {}:

```
{ statement1; statement2; statement3; }
```

A maioria das estruturas de controle que veremos nesta seção requer um enunciado genérico, como parte de sua sintaxe. A declaração pode ser uma simples declaração (uma simples instrução termina com um ponto e vírgula) ou um comando composto (várias instruções agrupadas em um bloco), como a descrita acima. Nos casos onde houver uma instrução simples, não é necessário incluí-la entre chaves ({}). Mas em casos onde houver uma instrução composta esta deve ser colocada entre chaves ({}), formando um bloco.

### 2.1.1 Estruturas condicionais: if e else

A palavra-chave `if` é usada para executar uma instrução (ou bloco) se uma condição é satisfeita. Sua forma é:

```
if (condition) statement
```

Onde `condition` é a expressão que está sendo avaliada. Se esta condição for verdadeira, a declaração `statement` é executada. Se for falsa, sua execução é ignorada (não executada) e o programa continua logo após esta estrutura condicional. Por exemplo, o seguinte fragmento de código imprime `x is 100` somente se o valor armazenado na variável `x` for 100:

```
1 if (x == 100)
2   cout << "x is 100";
```

Se nós queremos mais do que uma simples instrução executada caso a condição seja verdadeira, podemos especificar um bloco de códigos usando chaves {}:

```
1 if (x == 100) {
2   cout << "x is ";
3   cout << x;
4 }
```

Podemos adicionalmente especificar o que queremos que aconteça, se a condição não for cumprida, usando a palavra-chave `else` (senão). Sua forma usada em conjunto com `if` é:

```
if (condition) statement1 else statement2
```

Por exemplo:

```
1 if (x == 100)
2   cout << "x is 100";
3 else
4   cout << "x is not 100";
```

imprime na tela `x is 100` se `x`, de fato, possui valor igual a 100, senão imprime `x is not 100`.

O `if + else` pode ser concatenado com o intuito de verificar um intervalo de valores. O exemplo a seguir mostra a sua utilização imprimindo na tela dizer se o valor atualmente armazenado em `x` é positivo, negativo ou nenhum deles (ou seja, zero):

```
1 if (x > 0)
2   cout << "x is positive";
3 else if (x < 0)
4   cout << "x is negative";
5 else
6   cout << "x is 0";
```

Lembre-se que no caso em que queremos mais do que uma simples instrução executada devemos agrupá-las em um bloco, colocando-as entre chaves `{}`.

## 2.1.2 Estruturas de iterações (loops)

Os "loops" têm como finalidade repetir uma afirmação um certo número de vezes ou enquanto uma condição for satisfeita.

### O loop while (enquanto)

Seu formato é:

```
while (expression) statement
```

e sua funcionalidade é simplesmente repetir um bloco de códigos (`statement`) enquanto uma condição estabelecida na expressão (`expression`) é verdadeira. Por exemplo, vamos fazer um programa para realizar uma contagem regressiva usando um loop `while`:

```
1 // custom countdown using while
2
3 #include <iostream>
```

```
Enter the starting number > 8
8, 7, 6, 5, 4, 3, 2, 1, FIRE!
```

```

4  using namespace std;
5
6  int main () {
7      int n;
8      cout << "Enter the starting
number > ";
9      cin >> n;
10
11     while (n>0) {
12         cout << n << ", ";
13         --n;
14     }
15
16     cout << "FIRE!\n";
17     return 0;
18 }

```

Ao iniciar o programa o usuário é solicitado a inserir um número inicial para a contagem regressiva. Em seguida, o loop while inicia, se o valor digitado pelo usuário preenche as condições  $n > 0$  (que  $n$  é maior que zero), o bloco que segue a condição será executado e repetido enquanto a condição ( $n > 0$ ) continua sendo verdade.

O processo inteiro do programa anterior pode ser interpretado de acordo com o seguinte roteiro (a partir de `main`):

1. Usuário atribui um valor para  $n$
2. A condição enquanto é verificada ( $n > 0$ ). Neste ponto existem duas possibilidades:
  - \* condição é verdadeira: instrução é executada (passo 3)
  - \* condição é falsa: ignorar instrução e continuar depois dela (passo 5)
3. Execute o bloco:

```
cout << n << ", ";
-- n;
```

 (Imprime o valor de  $n$  na tela e diminui  $n$  de 1)
4. Fim do bloco. Voltar automaticamente para a etapa 2
5. Continuar o programa após o bloco: Imprimir FIRE! e encerrar o programa.

Ao criar um laço while devemos sempre considerar que ele deve terminar em algum momento, portanto, temos de fornecer, dentro deste bloco, algum método para forçar a condição ( $n > 0$ ) se tornar falsa em algum momento, caso contrário, o loop continuará sendo executado para sempre. Neste caso, nós incluímos o código `-- n;` que diminui o valor da variável que está sendo avaliado na condição ( $n$ ) em um - isso acabará por tornar a condição ( $n > 0$ ) falsa depois de certo número de iterações, para ser mais específico, quando  $n$  for 0.

## O loop do-while (faça-enquanto)

Seu formato é:

```
do statement while (condition);
```

Sua funcionalidade é exatamente a mesma que a do loop while, exceto que a condição do loop do while é avaliada após a execução do bloco de código, em vez de antes, garantindo, pelo



menos, uma execução das instruções, mesmo se a condição não for cumprida. Por exemplo, o programa *echoes* a seguir imprime qualquer número que você inserir até que você digite 0.

<pre>1 // number echoes 2 3 #include &lt;iostream&gt; 4 using namespace std; 5 6 int main (){ 7     unsigned long n; 8     do { 9         cout &lt;&lt; "Enter number (0 to end): "; 10        cin &gt;&gt; n; 11        cout &lt;&lt; "You entered: " &lt;&lt; n &lt;&lt; "\n"; 12    } while (n != 0); 13    return 0; 14 }</pre>	<pre>Enter number (0 to end): 12345 You entered: 12345 Enter number (0 to end): 160277 You entered: 160277 Enter number (0 to end): 0 You entered: 0</pre>
---	--

O `do-while` normalmente é utilizado quando a condição que tem de determinar o fim do ciclo é determinada dentro das instruções do loop, como no caso anterior, onde a entrada do usuário que é dentro do bloco, é usada para determinar se o loop tem que acabar. De fato, se você não entra o valor 0 no exemplo anterior, você será solicitado para inserir números indefinidamente.

## O loop for (para)

Seu formato é:

```
for (initialization; condition; increase) statement;
```

e sua principal função é repetir o `statement`, enquanto condição for verdadeira, como no loop `while`. Mas, além disso, o `for` fornece localizações específicas para conter uma declaração de inicialização (`initialization`) e uma declaração para o incremento (`increase`). Portanto, este ciclo é especialmente concebido para realizar uma ação repetitiva com um contador que é inicializado e aumentado (ou decrementado) em cada iteração.

Seu funcionamento é da seguinte maneira:

1. `initialization` é executada. Geralmente é um valor inicial de configuração para uma variável de contador. Esta é executada apenas uma vez.
2. `condition` é checada. Se é verdade o ciclo continua, caso contrário, o laço termina e a instrução (`statement`) é ignorada (não executado).
3. `statement` é executada. Como de costume, ela pode ser uma única instrução ou um bloco de instruções fechado entre chaves `{}`
4. finalmente, o que é especificado no campo de aumento (`increase`) é executado e o ciclo volta para a etapa 2.

Aqui está um exemplo de contagem regressiva usando um loop `for`:

<pre>1 // countdown using a for loop 2 #include &lt;iostream&gt;</pre>	<pre>10, 9, 8, 7, 6, 5, 4, 3, 2, 1, FIRE!</pre>
--	---

```

3 using namespace std;
4 int main () {
5     for (int n=10; n>0; n--) {
6         cout << n << ", ";
7     }
8     cout << "FIRE!\n";
9     return 0;
10 }

```

Os campos inicialização (initialization) e incremento (increase) são opcionais. Eles podem ficar vazios, mas em todos os casos, os sinais de ponto e vírgula entre eles devem ser escritos. Por exemplo, poderíamos escrever: `for (;n<10;)` se não quiséssemos especificar nenhuma inicialização e não incrementar, ou `for (;n<10;n++)`, se quiséssemos incluir um campo de incremento, mas não de inicialização (talvez porque a variável já foi inicializada antes). Neste exemplo, a variável `n` foi declarada dentro do laço `for` e existirá somente dentro dele. No entanto, ela pode ser declarada anteriormente ao seu uso.

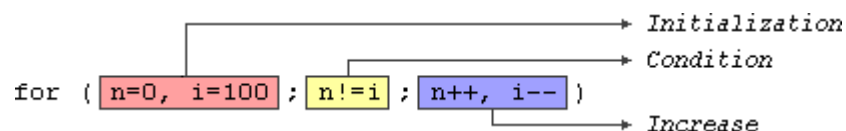
Opcionalmente, usando o operador vírgula (,) nós podemos especificar mais de uma expressão em qualquer um dos campos do loop (initialization, condition ou increase). O operador vírgula (,), neste caso, é um separador de expressão, e serve para separar mais de uma expressão em que apenas uma é normalmente esperada. Por exemplo, suponha que queremos iniciar mais de uma variável no nosso loop:

```

1 for ( n=0, i=100 ; n!=i ; n++, i-- ) {
2     // whatever here...
3 }

```

Este loop será executado 50 vezes se `n` ou `i` são modificadas dentro do laço:



`n` começa com um valor 0 e `i` com 100, a condição é `n!=i` (`n` diferente de `i`). Porque `n` é incrementado de um e `i` decrementado de um, a condição do loop se tornará falsa depois do ciclo 50, quando ambos `N` e `i` serão iguais a 50.

## 2.1.3 Instruções de salto

### O comando break

Usando `break` podemos deixar um loop mesmo que sua condição para o fim não é cumprida. Ele pode ser usado para terminar um loop infinito, ou forçá-lo a terminar antes de seu fim natural. Por exemplo, vamos parar a contagem regressiva antes de seu fim natural:

```

1 // break loop example
2
3 #include <iostream>

```

10, 9, 8, 7, 6, 5, 4, 3, countdown aborted!

```

4 using namespace std;
5
6 int main () {
7     int n;
8     for (n=10; n>0; n--) {
9         cout << n << ", ";
10        if (n==3) {
11            cout << "countdown aborted!";
12            break;
13        }
14    }
15    return 0;
16 }

```

## O comando continue

O comando `continue` faz com que o programa não execute o resto do laço na iteração atual, como se o fim do bloco fosse alcançado, fazendo com que ele salte para o início da iteração seguinte. Por exemplo, vamos pular o número 5 em nossa contagem regressiva:

```

1 // continue loop example
2 #include <iostream>
3 using namespace std;
4
5 int main () {
6     for (int n=10; n>0; n--) {
7         if (n==5) continue;
8         cout << n << ", ";
9     }
10    cout << "FIRE!\n";
11    return 0;
12 }

```

10, 9, 8, 7, 6, 4, 3, 2, 1, FIRE!

## O comando goto

O comando `goto` permite fazer um salto arbitrário para outro ponto do programa. Você deve usar esse recurso com cautela, pois sua execução realiza um salto incondicional ignorando qualquer tipo de limitações de aninhamento.

O ponto de destino é identificado por um rótulo, que depois é usado como um argumento para a instrução `goto`. Um rótulo é feito de um identificador válido seguido por dois pontos (:)

De um modo geral, esta instrução não tem nenhum uso concreto estruturado ou programação orientada a objeto (com exceção daqueles fãs de programação de baixo nível). Por exemplo, aqui é o nosso laço de contagem regressiva usando `goto`:

```

1 // goto loop example
2
3 #include <iostream>
4 using namespace std;
5
6 int main () {

```

10, 9, 8, 7, 6, 5, 4, 3, 2, 1, FIRE!

```

7   int n=10;
8   loop:
9   cout << n << ", ";
10  n--;
11  if (n>0) goto loop;
12  cout << "FIRE!\n";
13  return 0;
14 }

```

## A função exit

`exit` é uma função definida na biblioteca `cstdlib`. O objetivo da função `exit` é terminar o programa com um código de saída específico. Seu protótipo é:

```
void exit (int exitcode);
```

`exitcode` é utilizado por alguns sistemas operacionais e pode ser utilizado para chamar programas. Por convenção, um código de saída de 0 significa que o programa terminou normalmente, e qualquer outro valor significa que alguns resultados inesperados ou erros aconteceram.

### 2.1.4 A estrutura seletiva: switch

A sintaxe da instrução `switch` é um pouco peculiar. Seu objetivo é verificar possíveis diversos valores constantes de uma expressão. Algo semelhante ao que fizemos no início desta seção, com a concatenação de várias instruções de `if` and `else if`. Sua forma é a seguinte:

```

switch (expression){
    case constant1:
        group of statements 1;
        break;
    case constant2:
        group of statements 2;
        break;
    .
    .
    .
    default:
        default group of statements
}

```

Ele funciona da seguinte maneira: `switch` avalia expressão (`expression`) e verifica se ele é equivalente a `constant1`, se for, ele executa um grupo de comandos até encontrar o comando `break`. Quando ele encontra `break` o programa pula para o fim da estrutura `switch`.

Se a expressão não era igual a `constant1` será verificado `constant2`. Se for igual a este, ele irá executar grupo de comandos 2 até uma palavra-chave `break` for encontrada, e em seguida, irá saltar para o final da estrutura `switch`.

Finalmente, se o valor da expressão não corresponde a nenhuma das constantes especificadas anteriormente (você pode incluir muitos valores), o programa irá executar as declarações depois do rótulo `default`: (padrão), se ele existir, uma vez que é opcional.

Analisar os fragmentos de código a seguir. Ambos têm o mesmo comportamento:

switch example	if-else equivalent
<pre>switch (x) {   case 1:     cout &lt;&lt; "x is 1";     break;   case 2:     cout &lt;&lt; "x is 2";     break;   default:     cout &lt;&lt; "value of x unknown"; }</pre>	<pre>if (x == 1) {   cout &lt;&lt; "x is 1"; } else if (x == 2) {   cout &lt;&lt; "x is 2"; } else {   cout &lt;&lt; "value of x unknown"; }</pre>

O `switch` é um pouco peculiar dentro da linguagem C++, pois ele usa etiquetas em vez de blocos. Isto nos obriga a colocar instruções `break` após o grupo de instruções que queremos que seja executada para uma determinada condição. Caso contrário, as declarações restantes, incluindo os correspondentes a outros rótulos, também serão executadas, até o final do bloco `switch`, ou uma instrução `break` seja atingida.

Por exemplo, se não incluir uma declaração `break` após o primeiro grupo `case`, o programa não saltará automaticamente para o final do bloco `switch`, e ele iria continuar a executar o restante das declarações até que ele atinja um `break` ou uma instrução do final do bloco `switch`. Isso torna desnecessário incluir chaves `{}` em torno das declarações de cada um dos casos, e pode também ser útil para executar o mesmo bloco de instruções para diferentes valores possíveis para a expressão ser avaliada. Por exemplo:

```
1 switch (x) {
2   case 1:
3   case 2:
4   case 3:
5     cout << "x is 1, 2 or 3";
6     break;
7   default:
8     cout << "x is not 1, 2 nor 3";
9 }
```

Observe que a estrutura `switch` só pode ser usada para comparar uma expressão e constantes. Portanto, não podemos colocar variáveis como etiquetas (por exemplo, `case n:` onde `n` é uma variável) ou intervalos (`case (1..3:)`) porque eles não são válidos como constantes C++.

Se você precisar verificar um intervalo de valores ou valores que não são constantes use uma concatenação `if` e `else if`.

## 2.2 Funções (Parte I)

Usando funções podemos estruturar nossos programas de uma forma modular utilizando todo o potencial que a programação estruturada em C++ pode nós oferecer.

Uma função é um grupo de instruções que é executada a partir de uma chamada em algum ponto do programa. O seguinte é o seu formato:

```
type name ( parameter1, parameter2, ...) {  
    statements  
}
```

where:

- `type` é o especificador do tipo de dados retornado pela função.
- `name` é o identificador pelo qual será possível chamar a função.
- `parameters` (quantos forem necessários): Cada parâmetro consiste em um tipo de dados seguido por um identificador, como qualquer declaração de variáveis regulares (por exemplo: `int x`) e que atua dentro da função como uma variável local normal. Eles permitem passar argumentos para a função quando é chamada. Os diferentes parâmetros são separados por vírgulas.
- `statements` é o corpo da função, ou seja, um bloco de instruções entre chaves { }.

Aqui nós temos o nosso primeiro exemplo de função:

```
1 // function example  
2 #include <iostream>  
3 using namespace std;  
4  
5 int addition (int a, int b){  
6     int r;  
7     r=a+b;  
8     return (r);  
9 }  
10  
11 int main (){  
12     int z;  
13     z = addition (5,3);  
14     cout << "The result is " << z;  
15     return 0;  
16 }
```

The result is 8

A fim de examinar este código e, antes de tudo, lembrar algo dito no início desta apostila: um programa em C++ sempre inicia sua execução pela função `main`. Então, vamos começar por ela.

Podemos ver que a função `main` inicia pela declaração da variável `z` do tipo `int`. Logo depois, uma chamada a uma função denominada `addition`. Prestando atenção, seremos capazes de ver a semelhança entre a estrutura de chamada da função e a declaração da função:

```
int addition (int a, int b)
      ↑      ↑
z = addition ( 5 , 3 );
```

Os parâmetros e argumentos têm uma correspondência clara. Dentro da função `main` nós chamamos a função `addition` passando dois valores: 5 e 3. Estes valores correspondem aos parâmetros `int a` e `int b` declarados na função.

No momento em que a função é chamada de dentro de `main`, o controle é perdido pela função `main` e passados para a função `addition`. O valor de ambos os argumentos passados na chamada (5 e 3) são copiados para as variáveis locais `int a` e `int b` dentro da função.

A função `addition` declara outra variável local (`int r`) e atribui, por meio da expressão `r = a + b`, o resultado de `a` mais `b` a `r`. Como os parâmetros inteiros passados para `a` e `b` são 5 e 3, respectivamente, o resultado é 8.

A seguinte linha de código:

```
return (r);
```

finaliza a função `addition` e retorna o controle para a função que a chamou (neste caso, a função `main`). Nesse momento, o programa segue curso regular do mesmo ponto em que foi interrompida pela a adição. Mas, além disso, porque a instrução de retorno (`return`) da função `addition` especificou um valor: o conteúdo da variável `r` (`return (r);`), que naquele momento tinha um valor de 8. Este valor torna-se o valor da avaliação da chamada de função.

```
int addition (int a, int b)
      ↓ 8
z = addition ( 5 , 3 );
```

Assim sendo, o valor retornado pela função `addition`, a partir dos valores passados a ela (5, 3), é atribuído à variável `z`, neste caso, o valor que será atribuído a `z` é 8. Para explicar de outra forma, você pode imaginar que a chamada da função (`addition (5,3)`) é literalmente substituído pelo valor que ela retorna (8).

A seguinte linha de código, na função principal, é:

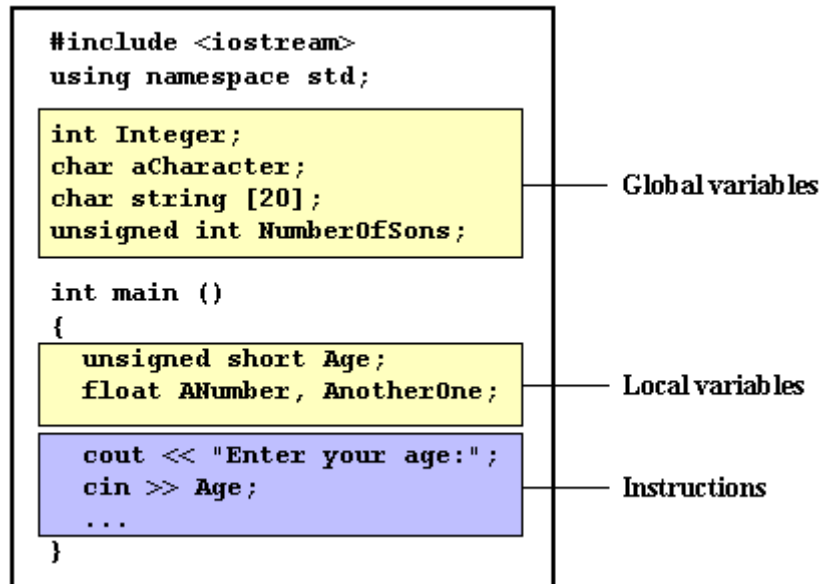
```
cout << "The result is " << z;
```

Que, como você já pode esperar, produz a impressão do resultado na tela.

## 2.2.1 Escopo das variáveis

O escopo das variáveis declaradas dentro de uma função ou qualquer outro bloco é apenas a própria função ou o próprio bloco e não pode ser utilizado fora deles. Por exemplo, no exemplo anterior, teria sido impossível usar as variáveis `a`, `b` ou `r` diretamente na função principal, uma vez que foram declaradas dentro da função `addition`, isto é, são variáveis locais desta função.

Além disso, teria sido impossível usar a variável `z` dentro da função `addition`, uma vez que esta era uma variável local da função principal (`main`).



Portanto, o escopo de variáveis locais é limitado ao nível do mesmo bloco em que elas são declaradas. No entanto, também temos a possibilidade de declarar variáveis globais, que são visíveis e que podem ser usadas em qualquer ponto do código, dentro e fora de todas as funções. Para declarar variáveis globais, você simplesmente tem que declarar a variável fora de qualquer função ou bloco, ou seja, diretamente no corpo do programa.

E aqui está outro exemplo sobre funções:

```
1 // function example
2 #include <iostream>
3 using namespace std;
4
5 int subtraction (int a, int b){
6     int r;
7     r=a-b;
8     return (r);
9 }
10
11 int main (){
12     int x=5, y=3, z;
13     z = subtraction (7,2);
14     cout << "The first result is " <<
15     z << '\n';
16     cout << "The second result is "
17     << subtraction (7,2) << '\n';
18     cout << "The third result is " <<
19     subtraction (x,y) << '\n';
20     z= 4 + subtraction (x,y);
21     cout << "The fourth result is "
22     << z << '\n';
23     return 0;
24 }
```

The first result is 5  
The second result is 5  
The third result is 2  
The fourth result is 6



Neste caso, criamos uma função chamada de `subtraction` (subtração). A única coisa que esta função faz é subtrair os parâmetros passados a ela e retornar o resultado.

No entanto, se examinarmos a função `main`, veremos que temos feito várias chamadas para a função de subtração. Nós usamos diferentes métodos de chamada para que você veja outras formas ou momentos em que uma função pode ser chamada.

A fim de compreender estes exemplos, deve-se considerar mais uma vez que uma chamada para uma função poderia ser substituída pelo valor que a função vai retornar. Por exemplo, o primeiro caso (que você já deve saber, porque é o mesmo padrão que temos usado nos exemplos anteriores):

```
1 z = subtraction (7,2);  
2 cout << "The first result is " << z;
```

Se substituirmos a chamada de função com o valor que retorna (i.e., 5), teríamos:

```
1 z = 5;  
2 cout << "The first result is " << z;
```

Bem como

```
cout << "The second result is " << subtraction (7,2);
```

tem o mesmo resultado que a chamada anterior, mas neste caso nós fizemos a chamada para a função subtração diretamente como um parâmetro de inserção para `cout`. Basta considerar que o resultado é o mesmo como se tivéssemos escrito:

```
cout << "The second result is " << 5;
```

uma vez que 5 é o valor retornado pela função `subtraction (7,2)`.

Neste caso:

```
cout << "The third result is " << subtraction (x,y);
```

A única coisa nova que foi introduzida é que os parâmetros da função de subtração são variáveis, ao invés de constantes. Isso é perfeitamente válido. Neste caso, os valores passados para a função `subtraction` são os valores armazenados nas variáveis `X` e `Y`, que são 5 e 3, respectivamente, obtemos, então, 2 como resultado.

O quarto caso é o mesmo dos anteriores. Simplesmente note que ao invés de:

```
z = 4 + subtraction (x,y);
```

nós poderíamos escrever:

```
z = subtraction (x,y) + 4;
```

com exatamente o mesmo resultado. Eu tenho mudado de lugar e assim você pode ver que o sinal de ponto e vírgula (;) vai ao final de toda a declaração. Não tem que ir, necessariamente, logo após a chamada da função.

## 2.2.2 Funções void (sem tipo)

Se você lembra da sintaxe de uma declaração de função:

```
type name ( argument1, argument2 ...) statement
```

you vai ver que a declaração começa com um tipo, que é o tipo da função em si (ou seja, o tipo do dado que será retornado pela função com a instrução de retorno). Mas e se não queremos retornar nenhum valor?

Imagine que nós queremos fazer uma função só para mostrar uma mensagem na tela. Nós não precisamos que ele retorne algum valor. Neste caso, devemos utilizar o especificador de tipo `void` para a função. Este é um especificador especial que indica ausência de tipo.

<pre>1 // void function example 2 #include &lt;iostream&gt; 3 using namespace std; 4 5 void printmessage () { 6     cout &lt;&lt; "I'm a function!"; 7 } 8 9 int main () { 10    printmessage (); 11    return 0; 12 }</pre>	<pre>I'm a function!</pre>
--	----------------------------

`void` também pode ser usado como parâmetro de uma função quando queremos especificar (explicitamente) que não queremos transferir nenhum parâmetro a ela no momento que é chamada. Por exemplo, a função `printmessage` poderia ter sido declarada como:

```
1 void printmessage (void) {
2     cout << "I'm a function!";
3 }
```

Embora seja opcional para especificar `void` na lista de parâmetros. Em C++, a lista de parâmetros pode simplesmente ser deixada em branco, se queremos uma função sem parâmetros.

O que você deve sempre lembrar é que o formato de chamada de uma função deve iniciar por seu nome e encerrar com os respectivos parâmetros entre parênteses. A inexistência de

parâmetros não nos exime da obrigação de escrever os parênteses. Por esse motivo, a chamada da função `printmessage` é:

```
printmessage ();
```

Os parênteses indicam claramente que esta é uma chamada de uma função e não o nome de uma variável ou alguma outra declaração. A chamada a seguir seria incorreta:

```
printmessage;
```

## 2.3 Funções (Parte II)

### 2.3.1 Argumentos passados por valor e por referência

Até agora, em todas as funções já vimos que os argumentos eram passados para as funções por valor. Isto significa que ao chamar uma função com parâmetros, o que temos passado para a função eram cópias de seus valores, mas nunca as próprias variáveis. Por exemplo, suponha que nós chamamos a nossa primeira função `addition` usando o seguinte código:

```
1 int x=5, y=3, z;  
2 z = addition ( x , y );
```

O que fizemos nesse caso foi a chamada da função `addition` passando os valores de `x` e `y`, ou seja, 5 e 3 respectivamente, mas não as variáveis `x` e `y`.

```
int addition (int a, int b)  
           ↑      ↑  
z = addition ( 5 , 3 );
```

Dessa forma, quando a função adição é chamada, são atribuídos 5 e 3 às suas variáveis locais `a` e `b`, respectivamente, mas qualquer modificação para `a` ou `b` dentro da função não terá nenhum efeito nos valores de `X` e `Y`, isto porque as variáveis `X` e `Y` não eram passadas para a função, mas apenas cópias de seus valores no momento em que a função foi chamada.

Mas pode haver alguns casos em que você precisa manipular de dentro de uma função o valor de uma variável externa. Para esse efeito, podemos usar argumentos passados por referência, como na função `duplicate` (duplicar), no seguinte exemplo:

```
1 // passing parameters by reference  
2 #include <iostream>  
3 using namespace std;  
4  
5 void duplicate (int& a, int& b,  
6               int& c){  
7     a*=2;  
8     b*=2;  
9     c*=2;
```

```
x=2, y=6, z=14
```


```

10 }
11
12 int main () {
13     int x=1, y=3, z=7;
14     duplicate (x, y, z);
15     cout << "x=" << x << ", y=" << y
16     << ", z=" << z;
17     return 0;
18 }

```

A primeira coisa que deve chamar a atenção é que na declaração de `duplicate` o tipo de cada parâmetro foi seguido por um sinal "e" comercial (&). Esse símbolo especifica que os seus argumentos correspondentes devem ser passados *por referência* em vez de *por valor*.

Quando uma variável é passada por referência, não estamos passando uma cópia do seu valor, na realidade as variáveis `a`, `b` e `c` e suas correspondentes, `x`, `y` e `z`, pertencentes à função `duplicate` ocupam a mesma posição de memória do computador e, conseqüentemente, qualquer alteração `a`, `b` ou `c` implica, diretamente, em uma alteração em `x`, `y` ou `z`.

**void duplicate (int& a, int& b, int& c)**  
  
**duplicate ( x , y , z );**

Se, durante a declaração da função, ao invés de declararmos:

```
void duplicate (int& a, int& b, int& c)
```

nós tivéssemos declarado desta maneira:

```
void duplicate (int a, int b, int c)
```

ou seja, sem os sinais e comercial (&), não teríamos passado por referência as variáveis, mas uma cópia de seus valores, e, portanto, a saída na tela do nosso programa teria sido os valores iniciais de `X`, `Y` e `Z` (sem ter sido modificado).

Passagem por referência é também uma forma eficaz de permitir uma função retornar mais de um valor. Por exemplo, a seguir tem-se uma função que retorna o número anterior e o próximo do primeiro parâmetro passado.

<pre> 1 // more than one returning value 2 #include &lt;iostream&gt; 3 using namespace std; 4 5 void prevnext (int x, int&amp; prev, 6   int&amp; next) { 7     prev = x-1; 8     next = x+1; 9 } </pre>	<p>Previous=99, Next=101</p>
--	------------------------------

```

9
10 int main (){
11     int x=100, y, z;
12     prevnext (x, y, z);
13     cout << "Previous=" << y << ",
Next=" << z;
14     return 0;
15 }

```

### 2.3.2 Parâmetros com valores padrões

Ao declarar uma função nós podemos especificar valores a cada um dos últimos parâmetros. Esse valor será usado caso o argumento correspondente seja deixado em branco quando a função é chamada. Para atribuir valores padrão, nós simplesmente temos que usar o operador de atribuição e um valor para os argumentos na declaração da função. Se nenhum valor para esse parâmetro é passado, no momento em que a função é chamada, o valor padrão é usado. No entanto, se um valor é especificado, o valor padrão é ignorado e o valor passado é utilizado. Por exemplo:

```

1 // default values in functions
2 #include <iostream>
3 using namespace std;
4
5 int divide (int a, int b=2){
6     int r;
7     r=a/b;
8     return (r);
9 }
10
11 int main (){
12     cout << divide (12);
13     cout << endl;
14     cout << divide (20,4);
15     return 0;
16 }

```

```

6
5

```

Como podemos ver no corpo do programa há duas chamadas para a função `divide`. Na primeira:

```
divide (12)
```

temos apenas um argumento especificado (a função `divide` permite até dois argumentos). Assim, a função assumiu que o segundo parâmetro é 2, uma vez que especificamos em sua declaração que, caso este parâmetro fosse omitido, `b` seria igual a 2 (`int b=2`, e não `int b`). Portanto, o resultado dessa chamada de função é 6 ( $12/2$ ).

Na segunda chamada:

```
divide (20,4)
```

Aqui existem parâmetros, então o valor padrão para `b` (`int b=2`) é ignorado e `b` assume o valor passado por argumento, isto é 4, fazendo que o resultado retornado seja igual a 5 ( $20/4$ ).

### 2.3.3 Sobrecarga de funções

Em C++ duas diferentes funções podem ter o mesmo nome. Isso significa que você pode dar o mesmo nome para mais de uma função caso estas tenham um número (quantidade) diferente de parâmetros ou tipos diferentes de seus parâmetros. Por exemplo:

```
1 // overloaded function
2 #include <iostream>
3 using namespace std;
4
5 int operate (int a, int b){
6     return (a*b);
7 }
8
9 float operate (float a, float b){
10    return (a/b);
11 }
12
13 int main (){
14     int x=5,y=2;
15     float n=5.0,m=2.0;
16     cout << operate (x,y);
17     cout << "\n";
18     cout << operate (n,m);
19     cout << "\n";
20     return 0;
21 }
```

```
10
2.5
```

Neste caso, nós definimos duas funções com o mesmo nome, `operate`, mas um deles aceita dois parâmetros do tipo `int` e o outro aceita do tipo `float`. O compilador sabe qual função deve ser chamada em cada caso, examinando os tipos passados como argumentos, no momento em que a função é chamada. Se for chamada com dois inteiros como argumentos, será chamada a função que tem dois parâmetros `int` em sua declaração. E se for chamada com dois argumentos do tipo `float`, será chamada àquela que possui dois parâmetros `float`.

Na primeira chamada de `operate` os dois argumentos passados são do tipo `int`, portanto, a primeira função declarada é chamada; essa função retorna o resultado da multiplicação de ambos os parâmetros. Enquanto a segunda chamada passa dois argumentos do tipo `float`, então a segunda função é chamada. Esta possui um comportamento diferente: ela divide um parâmetro pelo outro. Assim, o comportamento de uma chamada depende do tipo dos argumentos passados, isto é, a função `operate` foi sobrecarregada.

Observe que uma função não pode ser sobrecarregada apenas pelo seu tipo de retorno. Pelo menos um dos seus parâmetros deve ter um tipo diferente.

### 2.3.4 Recursividade

Recursividade é a propriedade que as funções têm de chamar a si mesmas. É útil para muitas tarefas, como classificação ou calcular o fatorial de números. Por exemplo, para obter o fatorial de um número ( $n!$ ) a fórmula matemática seria:

$$n! = n * (n-1) * (n-2) * (n-3) \dots * 1$$

mais concretamente, 5! (Fatorial de 5) seria:

$$5! = 5 * 4 * 3 * 2 * 1 = 120$$

e uma função recursiva para calcular isso em C++ pode ser:

```
1 // factorial calculator
2 #include <iostream>
3 using namespace std;
4
5 long factorial (long a){
6     if (a > 1)
7         return (a * factorial (a-1));
8     else
9         return (1);
10 }
11
12 int main (){
13     long number;
14     cout << "Please type a number: ";
15     cin >> number;
16     cout << number << "! = " <<
17     factorial (number);
18     return 0;
19 }
```

```
Please type a number: 9
9! = 362880
```

Repare como na função `factorial` incluímos uma chamada para si mesma, no entanto, somente se o argumento passado for maior que 1, pois, caso contrário, a função irá executar um loop infinito (provavelmente provocando um erro de estouro de pilha durante a execução).

Esta função tem uma limitação, devido ao tipo de dados foram utilizados (`long`). Os resultados obtidos não serão válidos para valores muito maiores do que 10! ou 15!, dependendo do sistema operacional que compilá-lo.

## 2.3.5 Declarando funções

Até agora, temos declarado todas as funções antes de sua primeira chamada no código-fonte. Estas chamadas foram, em geral, dentro da função `main`. Se você tentar repetir alguns dos exemplos anteriores, mas colocando a função `main` antes de quaisquer outras funções, você provavelmente irá obter alguns erros de compilação. A razão é que uma chamada somente pode ser realizada para uma função conhecida, isto é, declarada em algum ponto antes do código, como fizemos em todos os nossos exemplos anteriores.

Mas há uma maneira alternativa para evitar escrever todo o código de uma função antes de sua chamada, na função `main` ou em alguma outra função. Isto pode ser conseguido, declarando apenas um protótipo da função antes de ser usado, em vez de toda a definição. Esta declaração é menor, mas suficiente para o compilador para determinar o seu tipo de retorno e os tipos de seus parâmetros.

Sua forma é:

```
type name ( argument_type1, argument_type2, ...);
```

É idêntica a uma definição de função, exceto que ela não inclui o corpo da função em si (ou seja, as declarações de função normais são delimitados por chaves {}), e, ao invés, a declaração final do protótipo de uma função encerra-se necessariamente com um ponto e vírgula (;).

A enumeração dos parâmetros não precisa incluir os identificadores, mas apenas os especificadores de tipo. A inclusão de um nome para cada parâmetro, na definição da função é opcional na declaração do protótipo. Por exemplo, podemos declarar uma função chamada `protofunction` com alguma das seguintes declarações:

```
1 int protofunction (int first, int second);
2 int protofunction (int, int);
```

De qualquer forma, incluindo um nome para cada variável faz com que o protótipo fique mais inteligível.

<pre>1 // declaring functions prototypes 2 #include &lt;iostream&gt; 3 using namespace std; 4 5 void odd (int a); 6 void even (int a); 7 8 int main (){ 9     int i; 10    do { 11        cout &lt;&lt; "Type a number (0 to exit): "; 12        cin &gt;&gt; i; 13        odd (i); 14    } while (i!=0); 15    return 0; 16 } 17 18 void odd (int a){ 19     if ((a%2)!=0) cout &lt;&lt; "Number is odd.\n"; 20     else even (a); 21 } 22 23 void even (int a){ 24     if ((a%2)==0) cout &lt;&lt; "Number is even.\n"; 25     else odd (a); 26 }</pre>	<pre>Type a number (0 to exit): 9 Number is odd. Type a number (0 to exit): 6 Number is even. Type a number (0 to exit): 1030 Number is even. Type a number (0 to exit): 0 Number is even.</pre>
---	--

Este exemplo não é de fato um exemplo de eficiência. Estou certo de que, neste ponto você já pode fazer um programa com o mesmo resultado, mas utilizando apenas metade das linhas de código que foram usados neste exemplo. De qualquer forma este exemplo ilustra como funciona a prototipagem. Além disso, neste exemplo, a prototipagem de pelo menos uma das duas funções é necessária para compilar o código sem erros.

As primeiras coisas que nós vemos é a declaração das funções `odd` e `even`:



```
1 void odd (int a);  
2 void even (int a);
```

Isso permite que essas funções possam ser utilizadas antes de serem definidas, por exemplo, em `main`, que agora está localizado onde algumas pessoas acham que seja um lugar mais lógico: o início do código-fonte.

Enfim, a razão pela qual este programa precisa que pelo menos uma das funções deve ser declarada antes de ser definido é porque em `odd` existe uma chamada para `even` e em `even` existe uma chamada para `odd`. Se nenhuma das duas funções tivesse sido declarada anteriormente, um erro de compilação aconteceria, uma vez que `even` não seria visível para `odd` ou `odd` não seria visível para `even`.

Incluir o protótipo de todas as funções no início do programa é uma solução prática encontrada por alguns programadores para evitar erros de compilação.

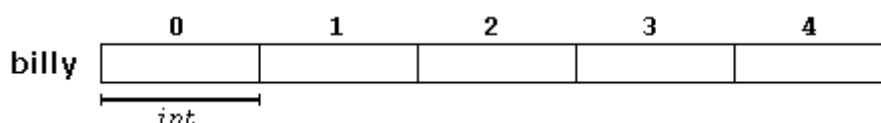
# Capítulo 3: Tipos de dados compostos

## 3.1 Matrizes

Uma matriz é um conjunto de elementos do mesmo tipo, colocados em locais contíguos de memória, para que possam ser individualmente referenciados pela adição de um índice para um identificador exclusivo.

Isso significa que, por exemplo, podemos armazenar 5 valores do tipo `int` em uma matriz sem ter que declarar 5 variáveis diferentes, cada uma com um identificador diferente. Em vez disso, usando uma matriz, podemos armazenar 5 valores diferentes do mesmo tipo, `int` por exemplo, com um identificador exclusivo.

Por exemplo, uma matriz que contém 5 valores inteiros do tipo `int` chamada `billy` poderia ser representada assim:



onde cada painel em branco representa um elemento da matriz, que neste caso são valores inteiros do tipo `int`. Estes elementos são numerados de 0 a 4, pois em C++ as matrizes possuem o primeiro índice sempre igual a 0, independentemente do seu comprimento.

Como uma variável normal, uma matriz deve ser declarada antes de ser usada. Uma declaração típica para uma matriz em C++ é a seguinte:

```
type name [elements];
```

onde `type` é um tipo válido (como `int`, `float` ...), `name` é um identificador válido e o campo `elements` especifica quantos desses elementos a matriz contém.

Portanto, a fim de declarar uma matriz chamada `billy`, como o mostrado na figura acima, é tão simples como:

```
int billy [5];
```

**NOTA:** O campo `elements` dentro de colchetes `[]` que representa o número de elementos da matriz deve ser um valor **constante**, já que matrizes são blocos de memória não-dinâmicos cujo tamanho deve ser determinado antes da execução. A fim de criar matrizes com comprimento variável, conceitos de alocação dinâmica de memória são necessários, o que é explicado mais adiante.

### 3.1.1 Inicializando matrizes

Ao declarar uma matriz regular de âmbito local (dentro de uma função, por exemplo), se não especificarmos os seus elementos, estes não serão inicializados, por padrão, e seu conteúdo será indeterminado até que nós guardamos algum valor neles. Os elementos de matrizes estáticas e de âmbito global, por outro lado, são automaticamente inicializados com seus valores padrão, que, para todos os tipos fundamentais, significa que são preenchidos com zeros.

Em ambos os casos, seja âmbito local ou global, quando se declarar uma matriz temos a possibilidade de atribuir valores iniciais para cada um dos seus elementos, colocando os valores em chaves {}. Por exemplo:

```
int billy [5] = { 16, 2, 77, 40, 12071 };
```

Esta declaração teria criado uma matriz assim:

	0	1	2	3	4
billy	16	2	77	40	12071

A quantidade de valores entre chaves {} não deve ser maior que o número de elementos que declaramos para a matriz entre colchetes []. Por exemplo, no exemplo da matriz `billy` foi declarado 5 elementos e na lista de valores iniciais dentro de chaves {} temos que especificar 5 valores, um para cada elemento.

Quando a inicialização de valores está prevista em uma matriz, a linguagem C++ permite a possibilidade de deixar os colchetes [] vazios. Neste caso, o compilador irá assumir uma dimensão para a matriz que corresponde ao número de valores compreendidos entre chaves {}:

```
int billy [] = { 16, 2, 77, 40, 12071 };
```

Após esta declaração, a matriz `billy` possuiria 5 números inteiros, uma vez que inserimos 5 valores durante sua inicialização.

### 3.1.2 Acessando valores de uma matriz

Em qualquer ponto de um programa em que uma matriz é visível, podemos acessar o valor de qualquer um dos seus elementos individualmente como se fosse uma variável normal, sendo capaz de ler e modificar seu valor. O formato é tão simples como:

`name[index]`

Seguindo os exemplos anteriores em que `billy` possuía 5 elementos e cada um desses elementos era do tipo `int`, devemos chamar cada elemento da seguinte maneira:

	billy[0]	billy[1]	billy[2]	billy[3]	billy[4]
billy					

Por exemplo, para armazenar o valor 75 no terceiro elemento de `billy`, nós poderíamos escrever a seguinte instrução:

```
billy[2] = 75;
```

e, por exemplo, para passar o valor do terceiro elemento de `billy` para uma variável chamada `a`, poderíamos escrever:

```
a = billy[2];
```

Portanto, a expressão `billy[2]` é para todos os efeitos como uma variável do tipo `int`.

Observe que o terceiro elemento de `billy` é especificado `billy[2]`, uma vez que o primeiro é `billy[0]`, o segundo é `billy[1]` e, portanto, o terceiro é `billy[2]`. Por esta mesma razão, o seu último elemento é `billy[4]`. Desta forma, se escrevessemos `billy[5]` estaríamos acessando o sexto elemento de `billy` e, portanto, superior ao tamanho da matriz.

**NOTA:** Em C++ é sintaticamente correto exceder o intervalo válido de índices para uma matriz. Isso pode criar problemas, uma vez que o acesso de elementos fora da faixa não causa erros de compilação, mas pode causar erros de execução. A razão para que isso seja permitido será vista mais adiante quando começarmos a usar ponteiros.

Neste ponto é importante ser capaz de distinguir claramente entre os dois usos que `[]` colchetes têm relacionado a matrizes. Eles executam duas tarefas diferentes: uma é para especificar o tamanho das matrizes que tenham sido declaradas; e a segunda é especificar os índices para os elementos de matrizes. Não confundir estes dois usos possíveis de colchetes `[]`.

```
1 int billy[5];           // declaration of a new array
2 billy[2] = 75;         // access to an element of the array.
```

Se você ler cuidadosamente, você verá que um especificador de tipo sempre precede uma declaração de variável ou matriz, enquanto ela nunca precede um acesso.

Algumas válidas outras operações com matrizes:

```
1 billy[0] = a;
2 billy[a] = 75;
3 b = billy [a+2];
4 billy[billy[a]] = billy[2] + 5;
```

```
1 // arrays example
2 #include <iostream>
3 using namespace std;
4
5 int billy [] = {16, 2, 77, 40,
6 12071};
7 int n, result=0;
8
9 int main (){
10     for ( n=0 ; n<5 ; n++ ){
11         result += billy[n];
12     }
13     cout << result;
```

12206

```

13 return 0;
14 }

```

### 3.1.3 Matrizes multidimensionais

Matrizes multidimensionais podem ser descritas como "matrizes de matrizes". Por exemplo, uma matriz bidimensional pode ser imaginada como uma tabela bidimensional feita de elementos, todos eles de um mesmo tipo de dados.

		0	1	2	3	4
jimmy	0					
	1					
	2					

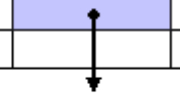
`jimmy` representa uma matriz bidimensional de 3 por 5 elementos do tipo `int`. A maneira de declarar esta matriz em C++ é:

```
int jimmy [3][5];
```

e, por exemplo, a forma para fazer referência ao segundo elemento verticalmente e o quarto horizontalmente em uma expressão seria:

```
jimmy[1][3]
```

		0	1	2	3	4
jimmy	0					
	1					
	2					


  
`jimmy[1][3]`

(Lembre-se que os índices de uma matriz sempre começam em zero)

Matrizes multidimensionais não são limitadas a dois índices (ou seja, duas dimensões). Elas podem conter quantos índices que forem necessários. **Mas cuidado!** A quantidade de memória necessária para uma matriz aumenta rapidamente com cada dimensão. Por exemplo:

```
char century [100][365][24][60][60];
```

declara uma matriz com elementos do tipo `char` para cada segundo de um século, ou seja mais de 3 bilhões de caracteres. Então, essa declaração iria consumir mais de 3 GB de memória!

As matrizes multidimensionais são apenas uma abstração para os programadores, uma vez que podemos obter os mesmos resultados com uma matriz unidimensional simplesmente colocando um fator entre os índices:

```
1 int jimmy [3][5]; // is equivalent to
```

```
2 int jimmy [15]; // (3 * 5 = 15)
```

Com a única diferença de que com matrizes multidimensionais o compilador lembra a profundidade de cada dimensão imaginária para nós. Tomemos como exemplo estes dois trechos de código, com ambos produzindo exatamente o mesmo resultado. Um usa uma matriz bidimensional e outra usa uma matriz simples:

multidimensional array	pseudo-multidimensional array
<pre>#define WIDTH 5 #define HEIGHT 3  int jimmy [HEIGHT][WIDTH]; int n,m;  int main () {     for (n=0;n&lt;HEIGHT;n++)         for (m=0;m&lt;WIDTH;m++)         {             jimmy[n][m]=(n+1)*(m+1);         }     return 0; }</pre>	<pre>#define WIDTH 5 #define HEIGHT 3  int jimmy [HEIGHT * WIDTH]; int n,m;  int main () {     for (n=0;n&lt;HEIGHT;n++)         for (m=0;m&lt;WIDTH;m++)         {             jimmy[n*WIDTH+m]=(n+1)*(m+1);         }     return 0; }</pre>

Nenhum dos dois códigos-fontes acima produz saída na tela, mas ambos atribuem valores para o bloco de memória chamado Jimmy, da seguinte forma:

jimmy {		<b>0</b>	<b>1</b>	<b>2</b>	<b>3</b>	<b>4</b>
	<b>0</b>	1	2	3	4	5
	<b>1</b>	2	4	6	8	10
	<b>2</b>	3	6	9	12	15

Temos usado "constantes definidas" (# define) para simplificar as possíveis modificações futuras do programa. Por exemplo, no caso em que decidimos ampliar a matriz para uma altura de 4 em vez dos 3 que poderia ser feito simplesmente mudando a linha:

```
#define HEIGHT 3
```

para:

```
#define HEIGHT 4
```

sem a necessidade de fazer outras modificações no programa.

### 3.1.4 Matrizes como parâmetros

Em algum momento podemos precisar passar uma matriz para uma função como um parâmetro. Em C++ não é possível passar um bloco inteiro de memória *por valor* como um parâmetro para

uma função, mas podemos passar o seu endereço. Na prática, isso tem quase o mesmo efeito e é muito mais rápida além de ser uma operação mais eficiente.

Para utilizar matrizes como parâmetros a única coisa que temos que fazer é, quando se declara a função, especificar em seus parâmetros o tipo dos elementos da matriz, um identificador e um par de colchetes vazio []. Por exemplo, a seguinte função:

```
void procedure (int arg[])
```

aceita um parâmetro do tipo "array de int" chamado arg. Para passar para esta função uma matriz declarada como:

```
int myarray [40];
```

seria suficiente escrever uma chamada como esta:

```
procedure (myarray);
```

Aqui você tem um exemplo completo:

```
1 // arrays as parameters
2 #include <iostream>
3 using namespace std;
4
5 void printarray (int arg[], int
length) {
6     for (int n=0; n<length; n++)
7         cout << arg[n] << " ";
8     cout << "\n";
9 }
10
11 int main () {
12     int firstarray[] = {5, 10, 15};
13     int secondarray[] = {2, 4, 6, 8,
10};
14     printarray (firstarray,3);
15     printarray (secondarray,5);
16     return 0;
17 }
```

```
5 10 15
2 4 6 8 10
```

Como você pode ver, o primeiro parâmetro (int arg[]) aceita qualquer matriz cujos elementos são do tipo int, independentemente do seu comprimento. Por essa razão, nós incluímos um segundo parâmetro que diz a função do comprimento de cada array que passamos. Isso permite que o loop que imprime a matriz saiba o intervalo que deve iterar sem ultrapassar o intervalo de dados.

Em uma declaração de função também é possível incluir matrizes multidimensionais. O formato de um parâmetro de matriz tridimensional é:

```
base_type[][depth][depth]
```

por exemplo, uma função com uma matriz multidimensional como argumento poderia ser:

```
void procedure (int myarray[][3][4])
```

Observe que os primeiros colchetes [] são deixados em branco, enquanto os seguintes não são. Isto é assim porque o compilador deve ser capaz de determinar dentro da função qual é a profundidade de cada dimensão adicional.

Matrizes, simples ou multidimensionais, passadas como parâmetros da função são uma fonte muito comum de erros para programadores iniciantes. Eu recomendo a leitura do capítulo sobre ponteiros para uma melhor compreensão sobre como operar com matrizes.

## 3.2 Sequência de caracteres

Como você já sabe, a biblioteca padrão do C++ implementa a poderosa classe `string`, que é muito útil para manipular cadeias de caracteres. No entanto, as cadeias de caracteres são, de fato, uma sequência de caracteres, e podemos representá-las também como matrizes simples de elementos do tipo `char`.

Por exemplo, a seguinte matriz:

```
char jenny [20];
```

pode armazenar até 20 elementos do tipo `char`. Ele pode ser representado como:

**jenny**

--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--

Nesta matriz, em teoria, podemos armazenar sequências de até 20 caracteres. Mas também podemos armazenar sequências mais curtas. Por exemplo, `jenny` poderia armazenar em algum ponto em um programa a sequência "Hello" ou a sequência "Merry christmas", já que ambos são menores do que 20 caracteres.

Portanto, uma vez que a matriz de caracteres pode armazenar sequências mais curtas do que o seu comprimento total, um caractere especial é usado para sinalizar o fim da sequência válida: o caractere nulo, cuja constante literal pode ser escrito como `'\0'` (barra invertida, zero).

Nossa matriz de 20 elementos do tipo `char`, chamada `jenny`, armazenando as sequências de caracteres "Hello" and "Merry Christmas" pode ser representada como:

**jenny**

H	e	l	l	o	\0														
M	e	r	r	y		C	h	r	i	s	t	m	a	s	\0				



Note como depois do conteúdo válido um caractere nulo ('\0') foi incluído a fim de indicar o fim da sequência. Os painéis em cor cinza representam elementos char com valores indeterminados.

### 3.2.1 Inicialização de sequências de caracteres terminada por caractere nulo

Como arrays de caracteres são vetores comuns como os demais, estes seguem as mesmas regras dos demais. Por exemplo, se quisermos inicializar um array de caracteres com uma sequência pré-determinada podemos fazê-lo tal como qualquer outro:

```
char myword[] = { 'H', 'e', 'l', 'l', 'o', '\0' };
```

Neste caso, foi declarada uma matriz de 6 elementos, do tipo `char`, e inicializado com os caracteres que formam a palavra "Hello" mais o caractere nulo '\0'. No entanto, vetores `char` têm um método adicional para inicializar seus valores: usando *literais string*.

Em exemplos descritos nos capítulos anteriores, nós já usamos constantes que representam sequências de caracteres várias vezes. Estas sequências são especificadas incluindo o texto (que queremos que se torne uma sequência de caracteres) entre aspas ("). Por exemplo:

```
"the result is: "
```

é uma constante string literal que temos, provavelmente, já utilizado anteriormente.

Os caracteres inseridos entre aspas duplas (") são constantes literais que contém, na verdade, um caractere nulo encerrando o array. Estes literais sempre têm um caractere nulo ('\0') anexado (automaticamente) ao seu final.

Portanto, podemos inicializar uma array de elementos do tipo `char` chamado `myword`, terminada com um caractere nulo, utilizando uma das seguintes maneiras:

```
1 char myword [] = { 'H', 'e', 'l', 'l', 'o', '\0' };
2 char myword [] = "Hello";
```

Em ambos os casos a matriz de caracteres `myword` é declarada com um tamanho de 6 elementos do tipo `char`: os 5 caracteres que compõem a palavra "Hello" mais um caractere nulo final ('\0'), que especifica o fim da sequência, e que, no segundo caso, quando se utilizam aspas (") o caractere nulo é anexado automaticamente.

Note que estamos falando de inicializar um array de caracteres no momento em que está sendo declarado e não sobre a atribuição de valores a eles uma vez que já tinha sido declarado. De fato, porque estes tipos de arrays (terminadas por caracteres nulos) são matrizes como as demais e têm as mesmas restrições que qualquer outra matriz, de modo que não são capazes de copiar blocos de dados com uma operação de atribuição.

Assumindo que `mystext` é uma variável `char[]`, expressões dentro de um código fonte, como:

```
1 mystext = "Hello";
2 mystext[] = "Hello";
```

Não seriam válidas, nem como seria:

```
mystext = { 'H', 'e', 'l', 'l', 'o', '\0' };
```

A razão para isso pode se tornar mais compreensível uma vez que você sabe um pouco mais sobre ponteiros. Desde então ficará claro que uma matriz é, na verdade, um ponteiro constante que aponta para um bloco de memória.

### 3.2.2 Usando uma sequência de caracteres terminada por caractere nulo

Sequências de caracteres é a maneira natural de tratar strings em C++ e estas podem ser usadas em muitos procedimentos. Na verdade, os literais strings possuem este tipo (`char []`) e também podem ser usados na maioria dos casos.

Por exemplo, `cin` e `cout` suportam sequências terminadas por caractere nulo tal que estes comandos podem ser usados para extrair diretamente sequências de caracteres usando `cin` ou para inseri-los através de `cout`. Por exemplo:

<pre>1 // null-terminated sequences of 2 characters 3 #include &lt;iostream&gt; 4 using namespace std; 5 6 int main (){ 7     char question[] = "Please, enter 8     your first name: "; 9     char greeting[] = "Hello, "; 10    char yourname [80]; 11    cout &lt;&lt; question; 12    cin &gt;&gt; yourname; 13    cout &lt;&lt; greeting &lt;&lt; yourname &lt;&lt; 14    "!"; 15    return 0; 16 }</pre>	<pre>Please, enter your first name: John Hello, John!</pre>
--	---

Como você pode ver, foi declarado três arrays de elementos `char`. Os dois primeiros foram inicializados com constantes literais enquanto o terceiro não foi inicializado. Em qualquer caso, temos que especificar o tamanho do vetor: nos dois primeiros (`question` e `greeting`) o tamanho foi definido implicitamente pelo comprimento da constante literal inicializada. Enquanto que para `yourname` foi explicitamente especificado o tamanho de 80 caracteres.

Finalmente, as sequências de caracteres armazenados em arrays `char` podem ser facilmente convertidas em objetos `string` usando apenas o operador de atribuição:

```
1 string mystring;
2 char myntcs[]="some text";
3 mystring = myntcs;
```

## 3.3 Ponteiros

Anteriormente vimos que as variáveis são vistas como células de memória e podem ser acessadas usando seus identificadores. Desta forma, não precisa se preocupar com a localização física dos dados dentro da memória, nós simplesmente usamos sua identificação sempre que for necessário referir-se à variável.

A memória do seu computador pode ser imaginada como uma sucessão de células de memória, cada uma do tamanho mínimo que computadores podem administrar (um byte). Estas células de memória (byte) são numeradas de forma consecutiva, de modo que, dentro de qualquer bloco de memória, cada célula tem o mesmo número que a anterior mais um.

Desta forma, cada célula pode ser facilmente localizada na memória, porque ela tem um endereço único e todas as células de memória seguem um padrão, sucessivamente. Por exemplo, se nós estamos olhando para a célula 1776, sabemos que deve estar entre as células de 1775 e 1777, exatamente mil células após 776 e exatamente mil células antes 2776.

### 3.3.1 O operador referência (&)

Assim que nós declaramos uma variável, a quantidade de memória necessária é atribuída por ela em uma posição específica na memória (o endereço de memória). Nós geralmente não *decidimos o local exato da variável dentro das células disponíveis da memória* - Felizmente, essa é uma tarefa executada automaticamente pelo sistema operacional durante a execução do programa. No entanto, em alguns casos, podemos estar interessados em conhecer o endereço onde a variável está sendo armazenada durante a execução do programa, a fim de operar com posições relativas a ela.

O endereço que localiza uma variável na memória é o que chamamos uma referência para essa variável. A referência de qualquer variável pode ser obtida precedendo o identificador da variável com um sinal de "e" comercial (&), conhecido como operador de referência, e que pode ser traduzido literalmente como "o endereço de". Por exemplo:

```
ted = &andy;
```

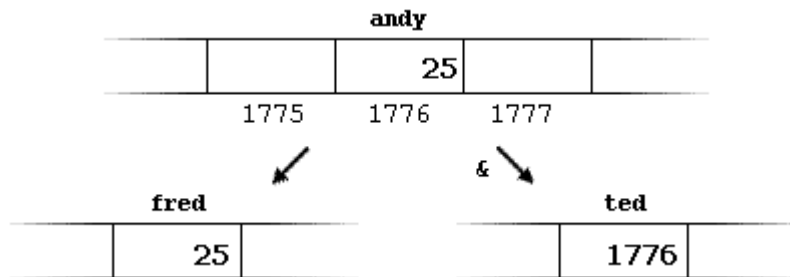
Isso atribuiria a `ted` o endereço da variável `andy`. Ao inserir o operador de referência (&) à variável `andy` não estamos mais falando sobre o conteúdo da variável em si, mas sobre a sua referência (ou seja, o seu endereço na memória).

De agora em diante vamos supor que `andy` é colocado (durante a execução) no endereço de memória 1776. Este número (1776) é apenas uma suposição arbitrária que estamos inventando agora, a fim de ajudar a esclarecer alguns conceitos neste material, mas na realidade, não podemos saber antes da execução o valor real do endereço que uma variável terá na memória.

Considere as seguintes linhas de código:

```
1 andy = 25;  
2 fred = andy;  
3 ted = &andy;
```

Os valores contidos em cada variável, após sua execução, são mostrados no diagrama abaixo:



Primeiro, foi atribuído o valor de 25 a `andy` (uma variável cujo endereço na memória que temos assumido como sendo 1776).

Na segunda declaração foi copiado para `fred` o conteúdo da variável `andy` (que é 25). Esta é uma operação de atribuição padrão.

Finalmente, a terceira declaração não copia para `ted` o valor contido em `andy`, mas uma referência a ela (isto é, seu endereço, que foi assumido igual a 1776). Isto ocorreu, pois na terceira linha de código foi precedido ao identificador `andy` o operador de referência(&).

A variável que armazena a referência a outra variável (como `Ted`, no exemplo anterior) é o que chamamos de um *ponteiro*. Os ponteiros são um poderoso recurso da linguagem C++ e tem muita utilidade na programação avançada. Mais à frente, vamos ver como esse tipo de variável é utilizada e declarada.

### 3.3.2 Operador dereferência (dereference)(\*)

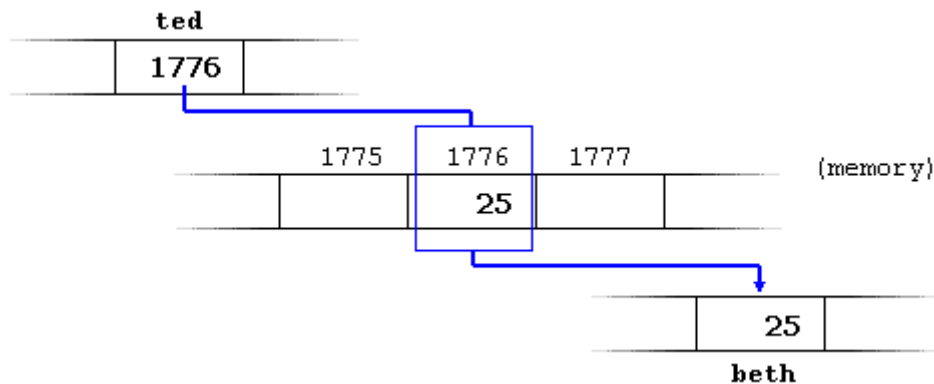
Acabamos de ver que uma variável que armazena uma referência à outra variável é chamada de ponteiro. Ponteiros "apontam para" a variável cuja referência é armazenada por eles.

Usando um ponteiro, podemos acessar diretamente o valor armazenado na variável que ele aponta. Para fazer isso, nós simplesmente temos que preceder o identificador do ponteiro com um asterisco (\*), que atua como operador de referência e que pode ser traduzido literalmente como "valor apontado por".

Assim, seguindo com os valores do exemplo anterior, se escrevermos:

```
beth = *ted;
```

(que poderíamos ler como: " `beth` é igual ao valor apontado por `ted` ") `beth` levaria o valor 25, uma vez que `ted` é 1776, e o valor apontado pelo 1776 é 25.



Você deve notado claramente que a expressão de `ted` se refere ao valor de 1776, enquanto `*ted` (com um asterisco `*` antes do identificador) refere-se ao valor armazenado no endereço 1776, que neste caso é 25. Observe a diferença ao incluir ou não o operador dereferência (Eu incluí um comentário explicativo de como cada uma dessas duas expressões pode ser lido):

```
1 beth = ted;    // beth equal to ted ( 1776 )
2 beth = *ted;   // beth equal to value pointed by ted ( 25 )
```

Observe a diferença entre os operadores de referência e dereferência:

- `&` é o operador de referência e pode ser lido como "endereço de"
- `*` é o operador de dereferência e pode ser lido como "valor apontado por"

Assim, eles têm significados complementares (ou opostos). Uma variável de referência pode ser dereferenciada com `*`.

Anteriormente foram realizadas as seguintes operações de atribuição:

```
1 andy = 25;
2 ted = &andy;
```

Logo após essas duas afirmações, as seguintes expressões são verdadeiras:

```
1 andy == 25
2 &andy == 1776
3 ted == 1776
4 *ted == 25
```

A primeira expressão é bastante clara, considerando que a operação de atribuição foi realizada em `andy` (`andy=25`). A segunda usa o operador de referência (`&`), que retorna o endereço da variável `andy`, que foi assumido o valor de 1776. A terceira é um tanto óbvia, já que a segunda expressão era verdade e da operação de atribuição realizada foi `ted=&andy`. A quarta expressão usa o operador de dereferência (`*`) que, como acabamos de ver, pode ser lido como "valor apontado por", e, o valor apontado por `ted` é realmente 25.

Então, depois de tudo isso, você também pode concluir que, enquanto o endereço apontado por `ted` permanece inalterado a seguinte expressão também será verdadeira:

```
*ted == andy
```

### 3.3.3 Declarando variáveis do tipo ponteiro

Devido à capacidade de um ponteiro referir-se diretamente para o valor que ele aponta, torna-se necessário especificar em sua declaração que tipo de dados um ponteiro vai apontar. Não é a mesma coisa apontar para um `char` como para apontar para um `int` ou `float`.

A declaração de ponteiros segue este formato:

```
type * name;
```

onde `type` é o tipo de dados que o ponteiro irá apontar. Este tipo não é o tipo do ponteiro em si, mas o tipo de dados que o ponteiro aponta. Por exemplo:

```
1 int * number;
2 char * character;
3 float * greatnumber;
```

Estes são três declarações de ponteiros. Cada um deles destina-se a apontar para um tipo de dado diferente, mas na verdade todos eles são ponteiros e todos eles vão ocupar o mesmo espaço na memória (o tamanho da memória de um ponteiro depende da plataforma onde o código está sendo executado). No entanto, os dados que eles apontam não ocupam o mesmo espaço na memória nem são do mesmo tipo: o primeiro aponta para um `int`, o segundo para um `char` e o último para um `float`. Portanto, embora estas três variáveis são ponteiros que ocupam o mesmo tamanho em memória, é dito que eles têm tipos diferentes: `int*`, `char*` e `float*`, respectivamente, dependendo do tipo que eles apontam.

**NOTA:** O sinal asterisco (\*) que usamos quando declaramos um ponteiro significa apenas que ele é um ponteiro (faz parte do seu especificador composto tipo), e não deve ser confundido com o operador dereferência que vimos anteriormente, mas também é escrito com um asterisco (\*). Eles são duas coisas diferentes representados com o mesmo sinal.

Agora dê uma olhada neste código:

```
1 // my first pointer
2 #include <iostream>
3 using namespace std;
4
5 int main (){
6     int firstvalue, secondvalue;
7     int * mypointer;
8
9     mypointer = &firstvalue;
10    *mypointer = 10;
11    mypointer = &secondvalue;
12    *mypointer = 20;
13    cout << "firstvalue is " <<
firstvalue << endl;
14    cout << "secondvalue is " <<
```

```
firstvalue is 10
secondvalue is 20
```

```

15     secondvalue << endl;
16     return 0;

```

Observe que, embora não foi definido diretamente um valor para as variáveis `firstvalue` ou `secondvalue`, estes valores foram definidos indiretamente através do uso de `mypointer`. Este é o procedimento:

Primeiro, foi atribuído como valor de `mypointer` uma referência para `firstvalue` usando o operador de referência (&). E então foi atribuído o valor 10 para a localização de memória apontada por `mypointer`. Neste momento `mypointer` está apontando para o local da memória do `firstvalue` e, conseqüentemente, modifica o valor de `firstvalue`.

Para demonstrar que um ponteiro pode tomar vários valores no mesmo programa, foi repetido o processo com `secondvalue` com mesmo ponteiro, `mypointer`.

Aqui está um exemplo um pouco mais elaborado:

```

1  // more pointers
2  #include <iostream>
3  using namespace std;
4
5  int main () {
6      int firstvalue = 5, secondvalue = 15;
7      int * p1, * p2;
8
9      p1 = &firstvalue; // p1 = address of
firstvalue
10     p2 = &secondvalue; // p2 = address of
secondvalue
11     *p1 = 10;          // value pointed by
p1 = 10
12     *p2 = *p1;         // value pointed by
p2 = value pointed by p1
13     p1 = p2;           // p1 = p2 (value of
pointer is copied)
14     *p1 = 20;          // value pointed by
p1 = 20
15     cout << "firstvalue is " << firstvalue
<< endl;
16     cout << "secondvalue is " <<
secondvalue << endl;
17     return 0;
18 }

```

```

firstvalue is 10
secondvalue is 20

```

Foi incluído como um comentário em cada linha do código: comercial (&) como "endereço de" e um asterisco (\*) como "valor apontado por".

Observe que há expressões com os ponteiros `p1` e `p2` com e sem operador dereferência (\*). O significado de uma expressão usando o operador de dereferência (\*) é muito diferente de um que não o utiliza: quando este operador precede o nome do ponteiro, a expressão se refere ao valor que está sendo apontado, ao mesmo tempo quando um nome de ponteiro aparece sem

este operador, refere-se para o valor do ponteiro em si (ou seja, o endereço ao qual o ponteiro está apontando).

Outra coisa que pode chamar sua atenção é a linha:

```
int * p1, * p2;
```

Esta linha de código declara os dois ponteiros usados no exemplo anterior. Mas repare que há um asterisco (\*) para cada ponteiro, de modo que ambos têm tipo `int*` (ponteiro para `int`).

Caso contrário, o tipo para a segunda variável declarada teria sido `int` (e não `int *`), devido às relações de precedência. Se tivéssemos escrito:

```
int * p1, p2;
```

`p1` seria certamente do tipo `int*`, mas `p2` seria tipo `int`. Isto é devido às regras de precedência do operador. Mas de qualquer maneira, basta lembrar que você tem que colocar um asterisco por ponteiro que é suficiente para a maioria dos usuários de ponteiro.

### 3.3.4 Ponteiros e matrizes

O conceito de matriz é muito ligado ao do ponteiro. Na verdade, o identificador de uma matriz é equivalente ao endereço do seu primeiro elemento, como um ponteiro também é equivalente ao endereço do primeiro elemento, então, na verdade, eles têm o mesmo conceito. Por exemplo, supondo que estas duas declarações:

```
1 int numbers [20];  
2 int * p;
```

A seguinte operação de atribuição seria válida:

```
p = numbers;
```

Depois disso, `p` e `numbers` seriam equivalentes e teriam as mesmas propriedades. A única diferença é que nós podemos mudar o valor do ponteiro `p` por outro, ao passo que `numbers` apontaria sempre para o primeiro dos 20 elementos do tipo `int`, com o qual ele foi definido. Portanto, ao contrário de `p`, que é um ponteiro comum, `numbers` é uma matriz e pode ser considerada como um *ponteiro constante*. Portanto, a atribuição a seguir não seria válida:

```
numbers = p;
```

Como `numbers` é uma matriz, este funciona como um ponteiro constante, e, portanto, não podemos atribuir valores para constantes.

Devido às características das variáveis, todas as expressões que incluem ponteiros no exemplo a seguir são perfeitamente válidas:



```

1 // more pointers
2 #include <iostream>
3 using namespace std;
4
5 int main () {
6     int numbers[5];
7     int * p;
8     p = numbers; *p = 10;
9     p++; *p = 20;
10    p = &numbers[2]; *p = 30;
11    p = numbers + 3; *p = 40;
12    p = numbers; *(p+4) = 50;
13    for (int n=0; n<5; n++)
14        cout << numbers[n] << ", ";
15    return 0;
16 }

```

```
10, 20, 30, 40, 50,
```

No capítulo sobre matrizes usamos colchetes ([]) várias vezes a fim de especificar o índice de um elemento da matriz a que queríamos nos referir. Bem, estes operadores (sinal de colchetes []) também são um operador **dereferência** conhecido como *operador de offset*. Eles dereferenciam a variável como faz o operador asterisco (\*), mas também adicionam o número entre parênteses para o endereço a ser dereferenciado. Por exemplo:

```

1 a[5] = 0;           // a [offset of 5] = 0
2 *(a+5) = 0;         // pointed by (a+5) = 0

```

Estas duas expressões são equivalentes e ambas válidas, independentemente se `a` é um ponteiro ou uma matriz.

### 3.3.5 Inicialização de ponteiros

Ao declarar ponteiros, nós podemos especificar explicitamente qual a variável que queremos que eles apontem:

```

1 int number;
2 int *tommy = &number;

```

O comportamento deste código é equivalente a:

```

1 int number;
2 int *tommy;
3 tommy = &number;

```

Ao inicializar um ponteiro, nós sempre estamos atribuindo o valor de referência para onde o ponteiro aponta (`tommy`), nunca o valor que está sendo apontado (`* tommy`). Você deve considerar que, no momento de declarar um ponteiro, o asterisco (\*) indica apenas que é um ponteiro, ele não é o operador de dereferência (embora ambos usam o mesmo sinal: \*). Lembre-se, são duas funções diferentes de um sinal. Assim, devemos tomar cuidado para não confundir com o código anterior:

```

1 int number;
2 int *tommy;
3 *tommy = &number;

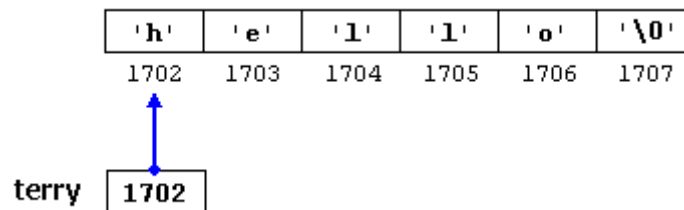
```

que está incorreto, e, mesmo assim, não teria muito sentido neste caso.

Como no caso de matrizes, o compilador permite o caso especial que queremos inicializar o conteúdo ao qual o ponteiro aponta com constantes no mesmo instante que o ponteiro é declarado:

```
char * terry = "hello";
```

Neste caso, o espaço de memória é reservado para conter a palavra "hello" e, em seguida, um ponteiro para o primeiro caractere do bloco de memória é atribuído a `terry`. Se imaginarmos que "hello" é armazenado em locais da memória que começam a partir do endereço 1702, podemos representar a declaração anterior como:



É importante indicar que `terry` contém o valor 1702, e não 'h', nem "hello", embora, na verdade, 1702 é o endereço de ambos.

O ponteiro `terry` aponta para uma sequência de caracteres e pode ser lido como se fosse uma matriz comum (lembre-se que uma matriz é igual a um ponteiro constante). Por exemplo, podemos acessar o quinto elemento da matriz com qualquer uma dessas duas expressões:

```

1 *(terry+4)
2 terry[4]

```

Ambas as expressões têm um valor de 'o' (o quinto elemento da matriz).

### 3.3.6 Aritmética de ponteiros

As operações aritméticas com ponteiros é um pouco diferente do que as operações com dados inteiros. Para começar, as únicas operações que podem ser realizadas são adição e subtração, as outras não fazem sentido no mundo dos ponteiros. Mas a adição e subtração têm um comportamento diferente de acordo com o tamanho do tipo de dados para o qual eles apontam.

Quando vimos os diferentes tipos de dados fundamentais, vimos que alguns ocupam menos ou mais espaço que outros na memória. Por exemplo, vamos supor que em um dado compilador, para uma máquina específica, `char` tem 1 byte, `short` ocupa 2 bytes e `long` 4.

Suponha que nós definimos três ponteiros neste compilador:

```

1 char *mychar;
2 short *myshort;
3 long *mylong;

```

e que nós sabemos que eles apontam para posições de memória 1000, 2000 e 3000, respectivamente.

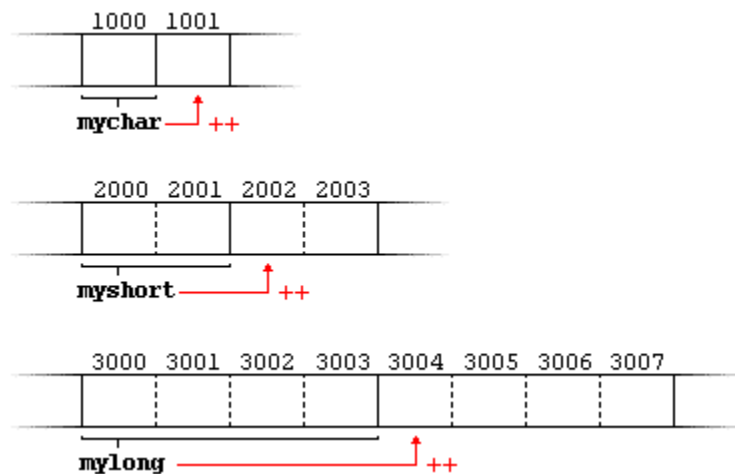
Então se nós escrevemos:

```

1 mychar++;
2 myshort++;
3 mylong++;

```

`mychar`, como você pode esperar, deve conter o valor de 1001. Mas não tão obviamente, `myshort` deverá conter o valor de 2002, e `mylong` deverá conter 3004, apesar de terem sido aumentados uma só vez. A razão é que, ao adicionar uma unidade a um ponteiro estamos fazendo com que ele aponte para o elemento seguinte do mesmo tipo com o qual ele foi definido e, portanto, o tamanho em bytes do tipo apontado é adicionado ao ponteiro.



Isto é aplicável tanto ao somar e subtrair qualquer número para um ponteiro. Iria acontecer exatamente o mesmo se escrevermos:

```

1 mychar = mychar + 1;
2 myshort = myshort + 1;
3 mylong = mylong + 1;

```

Ambos os operadores, incrementar (`++`) ou decrementar (`--`), têm precedência sobre o operador de referência (`*`), mas ambos têm um comportamento especial quando usados como sufixo (a expressão é avaliada com o valor que tinha antes de ser incrementada).

Portanto, a seguinte expressão pode levar à confusão:

```
*p++
```

essa expressão é equivalente a `*(p++)`. Portanto, o que esta expressão faz é incrementar o valor de `p` (de modo que agora ele aponta para o elemento seguinte), isto acontece porque `++` é usado como sufixo e toda a expressão é avaliada como o valor apontado pela referência original (o endereço do ponteiro apontado para antes de ser aumentada).

Observe a diferença com:

```
(*p)++
```

Aqui, a expressão foi avaliada como o valor apontado por `p` incrementado em uma unidade. O valor de `p` (o próprio ponteiro) não seria alterado (o que está sendo modificado é o que está sendo apontado por este ponteiro).

Se escrevermos:

```
*p++ = *q++;
```

Porque `++` tem uma precedência maior do que `*`, `p` e `q` são incrementados, mas como os operadores de incremento (`++`) são utilizados como sufixo e não como prefixo, o valor atribuído à `*p` é `*q` **antes** de `p` e `q` serem incrementados. E então, ambos serão aumentados. Este código seria equivalente a:

```
1 *p = *q;  
2 ++p;  
3 ++q;
```

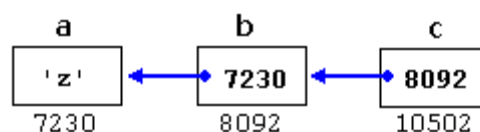
Como sempre, eu recomendo que você use parênteses `()`, a fim de evitar resultados inesperados e para dar maior legibilidade ao código.

### 3.3.7 Ponteiros para ponteiros

C++ permite o uso de ponteiros que apontam para ponteiros, e estes, por sua vez, apontam para dados (ou até mesmo para outros ponteiros). Para fazer isto só precisamos adicionar um asterisco (`*`) para cada nível de referência em suas declarações:

```
1 char a;  
2 char * b;  
3 char ** c;  
4 a = 'z';  
5 b = &a;  
6 c = &b;
```

Supondo que os locais de memória escolhidos aleatoriamente para cada variável são 7230, 8092 e 10502, respectivamente, estes códigos podem ser representados graficamente como:



O valor de cada variável é escrito dentro de cada célula, as células estão sob seus respectivos endereços na memória.

A novidade nesse exemplo é a variável `c`, que pode ser utilizada em três diferentes níveis de indireção, cada uma delas corresponderia a um valor diferente:

- `c` é do tipo `char**` e valor igual a 8092
- `*c` é do tipo `char*` e valor igual a 7230
- `**c` é do tipo `char` e valor igual a 'z'

### 3.3.8 Ponteiros void

Os ponteiros `void` são um tipo especial de ponteiro. Em C++, `void` representa a ausência do tipo, então os ponteiros deste tipo são ponteiros que apontam para um valor que não tem tipo (e, portanto, comprimento e propriedades de referência indeterminada).

Isso permite que os ponteiros do tipo `void` apontem para qualquer tipo de dados, a partir de um valor inteiro ou um float para uma sequência de caracteres. Mas em troca, eles têm uma grande limitação: os dados apontados por eles não podem ser diretamente dereferenciado (que é lógico, pois não temos nenhum tipo de referência), e por isso sempre teremos que atribuir o endereço do ponteiro nulo para algum outro ponteiro que aponta para um tipo de dado concreto antes de dereferenciá-lo.

Um de seus usos pode ser a passagem de parâmetros genéricos para uma função:

```
1 // increaser
2 #include <iostream>
3 using namespace std;
4
5 void increase (void* data, int psize){
6     if ( psize == sizeof(char) ){
7         char* pchar;
8         pchar=(char*)data;
9         ++(*pchar);
10    }
11    else if (psize == sizeof(int) ){
12        int* pint;
13        pint=(int*)data;
14        ++(*pint); }
15 }
16
17 int main (){
18     char a = 'x';
19     int b = 1602;
20     increase (&a, sizeof(a));
21     increase (&b, sizeof(b));
22     cout << a << ", " << b << endl;
23     return 0;
24 }
```

y, 1603

sizeof é um operador integrado na linguagem C++ que retorna o tamanho em bytes do seu parâmetro. Para tipos de dados não dinâmicos (char, int, float, etc) esse valor é uma constante. Assim, por exemplo, sizeof(char) é 1, porque o tipo char possui um byte.

### 3.3.9 Ponteiro nulo

Um ponteiro nulo é um ponteiro comum de qualquer tipo que tem um valor especial e indica que ele não está apontando para qualquer referência válida ou endereço de memória. Este valor especial é o zero para qualquer tipo de ponteiro.

```
1 int * p;  
2 p = 0; // p has a null pointer value
```

**NOTA:** Não confundir com ponteiros nulos ponteiros void. Um ponteiro nulo é um valor que pode assumir um ponteiro de um determinado tipo para indicar que ele não está apontando para "nenhum lugar". Um ponteiro void é um tipo especial de ponteiro que pode apontar para qualquer lugar sem um tipo específico. Um se refere ao valor armazenado no ponteiro e outro para o tipo de dados que ele aponta.

### 3.3.10 Ponteiros para funções

A linguagem C++ permite operações com ponteiros para funções. O uso típico deste é para passar uma função como um argumento para outra função, uma vez que estes ponteiros não podem ser passados dereferenciados. Para declarar um ponteiro para uma função temos que declará-la como uma função comum, exceto que o nome da função é colocado entre parênteses () e um asterisco (\*) é inserido antes do nome:

```
1 // pointer to functions  
2 #include <iostream>  
3 using namespace std;  
4  
5 int addition (int a, int b){  
6     return (a+b);  
7 }  
8  
9 int subtraction (int a, int b){  
10     return (a-b);  
11 }  
12  
13 int operation (int x, int y, int  
14 (*functocall) (int,int)){  
15     int g;  
16     g = (*functocall) (x,y);  
17     return (g);  
18 }  
19  
20 int main (){  
21     int m,n;  
22     int (*minus) (int,int) =  
    subtraction;
```

8

```

23     m = operation (7, 5, addition);
24     n = operation (20, m, minus);
25     cout <<n;
26     return 0;
27 }

```

No exemplo, `minus` é um ponteiro para uma função que tem dois parâmetros do tipo `int`. Ele é imediatamente atribuído para apontar para a função `subtraction`, tudo em uma única linha:

```
int (* minus)(int,int) = subtraction;
```

## 3.4 Alocação dinâmica de memória

Até agora, em todos os nossos programas, nós tínhamos apenas memória disponível à nossas variáveis, como nós as declaramos. A quantidade de memória reservada para o nosso programa era determinada no código fonte, antes da execução do programa, isto é, uma variável `char` utiliza 1 byte, uma variável `int` 4 bytes, etc. Mas, e se precisamos de uma quantidade variável de memória que só pode ser determinada durante a execução? Por exemplo, no caso em que uma entrada do usuário determine a quantidade de espaço de memória necessária.

A resposta é *alocação dinâmica de memória*, para isto, a linguagem C++ integra os operadores `new` e `delete`.

### 3.4.1 Operadores `new` e `new[]`

Para alocar memória dinamicamente devemos usar operador `new`. `new` é seguido por um tipo de dados e, se uma sequência de mais de um elemento é necessária, o número de elementos deve ser inserido entre colchetes `[]`. Ele retorna um ponteiro para o início do novo bloco de memória alocada. Sua forma é:

```

pointer = new type
pointer = new type [number_of_elements]

```

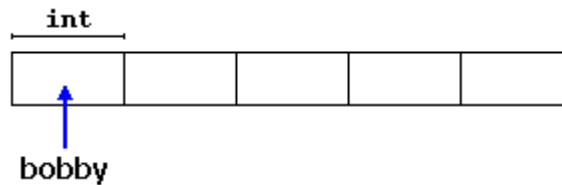
A primeira expressão é usada para alocar memória para um único elemento do tipo `type`. O segundo é usado para atribuir um bloco (uma matriz) de elementos de tipo `type`, onde `number_of_elements` é um valor inteiro que representa a quantidade destes. Por exemplo:

```

1 int * bobby;
2 bobby = new int [5];

```

Neste caso, o sistema atribui dinamicamente espaço para cinco elementos do tipo `int` e retorna um ponteiro para o primeiro elemento da sequência, que é atribuído à `bobby`. Portanto, agora `bobby` aponta para um bloco válido de memória com espaço para cinco elementos do tipo `int`.



O primeiro elemento apontado por `bobby` pode ser acessado tanto com a expressão `bobby[0]` ou a expressão `*bobby`. Ambos são equivalentes, como foi explicado na seção sobre ponteiros. O segundo elemento pode ser acessado tanto com `bobby[1]` ou `*(bobby+1)` e assim por diante.

Você pode estar se perguntando a diferença entre declarar uma matriz normal e atribuição dinâmica de memória para um ponteiro, como acabamos de fazer. A diferença mais importante é que o tamanho de uma matriz é um valor constante, limitado no momento da concepção do programa, antes de sua execução, enquanto ao utilizar a alocação dinâmica de memória permite a atribuição de memória durante a execução do programa (runtime) usando qualquer variável ou um valor constante para determinar seu tamanho.

A memória solicitada pelo nosso programa durante a execução é atribuída pelo sistema operacional a partir da pilha de memória. No entanto, a pilha é um recurso limitado e pode se esgotar. Portanto, é importante ter algum mecanismo para verificar se o nosso pedido de alocação foi bem sucedido ou não.

C++ fornece dois métodos padrão para verificar se a alocação foi bem sucedida:

Uma delas é o tratamento de exceções. Usando este método uma exceção do tipo `bad_alloc` é acionada quando a alocação falhar. As exceções são um poderoso recurso do C++ explicado mais tarde nessa apostila. Mas, por agora você deve saber que, se essa exceção é lançada e não é tratada por um manipulador específico, a execução do programa será encerrada.

Este método de exceção é o método padrão usado pelo `new` e é o utilizado em uma declaração como:

```
bobby = new int [5]; // if it fails an exception is thrown
```

O outro método é conhecido como `nothrow`, e o que acontece quando ele é usado é que, quando uma alocação de memória falha, em vez de executar uma exceção `bad_alloc` ou encerrar o programa, o ponteiro retornado por `new` é um ponteiro nulo, e o programa continua sua execução normalmente.

Este método pode ser especificado usando um objeto especial chamado `nothrow`, declarado no cabeçalho `<new>`, como argumento para o `new`:

```
bobby = new (nothrow) int [5];
```

Neste caso, se a atribuição deste bloco de memória falha, a falha pode ser detectada verificando se `bobby` teve um valor de ponteiro nulo:

```
1 int * bobby;
2 bobby = new (nothrow) int [5];
3 if (bobby == 0) {
4     // error assigning memory. Take measures.
```



```
5 };
```

Este método `nothrow` requer mais trabalho do que o método de exceção, uma vez que o valor retornado deve ser verificado após cada alocação de memória, mas vou usá-lo nos nossos exemplos, devido à sua simplicidade. De qualquer forma esse método pode se tornar tedioso para projetos maiores, onde o método de exceção é geralmente preferido. O método de exceção será explicado em detalhe mais tarde neste material.

### 3.4.2 Operadores `delete` e `delete[]`

Uma vez que a quantidade de memória dinâmica é geralmente limitada dentro de um programa, após ser utilizada ela deve ser liberada, para que a memória se torne disponível novamente para outras solicitações. Este é o objetivo do operador `delete`, seu formato é:

```
1 delete pointer;  
2 delete [] pointer;
```

A primeira expressão deve ser usada para apagar a memória alocada de um único elemento, e a segunda para a memória alocada para vários elementos (matrizes).

O valor passado como argumento para excluir deve ser um ponteiro para um bloco de memória previamente alocado com o operador `new`, ou um ponteiro nulo (no caso de um ponteiro nulo, `delete` não produz nenhum efeito).

<pre>1 // rememb-o-matic 2 #include &lt;iostream&gt; 3 #include &lt;new&gt; 4 using namespace std; 5 6 int main () { 7     int i,n; 8     int * p; 9     cout &lt;&lt; "How many numbers would you like to type? "; 10    cin &gt;&gt; i; 11    p= new (nothrow) int[i]; 12    if (p == 0) 13        cout &lt;&lt; "Error: memory could not be allocated"; 14    else { 15        for (n=0; n&lt;i; n++){ 16            cout &lt;&lt; "Enter number: "; 17            cin &gt;&gt; p[n]; 18        } 19        cout &lt;&lt; "You have entered: "; 20        for (n=0; n&lt;i; n++) 21            cout &lt;&lt; p[n] &lt;&lt; ", "; 22        delete[] p; 23    } 24    return 0; 25 }</pre>	<pre>How many numbers would you like to type? 5 Enter number : 75 Enter number : 436 Enter number : 1067 Enter number : 8 Enter number : 32 You have entered: 75, 436, 1067, 8, 32,</pre>
--	---

Observe como o valor entre parêntesis na declaração de `new` é um valor variável digitado pelo usuário (`i`), não um valor constante:

```
p= new (nothrow) int[i];
```

Mas o usuário poderia ter entrado um valor para `i` tão grande que nosso sistema não poderia lidar. Por exemplo, quando eu tentei dar um valor de 1000 milhões para a pergunta "Quantos números", meu sistema não pôde alocar a memória e recebi a mensagem de texto para este caso (*Error: memory could not be allocated*). Lembre-se que se tentássemos alocar a memória, sem especificar o parâmetro `nothrow` na expressão `new`, uma exceção seria iniciada, que se não for tratada, termina o programa.

É uma boa prática sempre verificar se um bloco de memória dinâmica com êxito foi atribuído. Portanto, se você usar o método `nothrow`, você deve sempre verificar o valor do ponteiro retornado. Caso contrário, use o método de exceção, mesmo se você não tratar a exceção. Desta forma, o programa será encerrado neste ponto, sem causar resultados inesperados ao continuar a execução de um código que pressupõe um bloco de memória que tenha sido atribuído, quando na verdade ele não foi.

## 3.5 Estruturas de dados

Nós já aprendemos como grupos de dados sequenciais (matrizes) podem ser usados em C++. No entanto, isso é um pouco restritivo, pois em muitas ocasiões o que queremos armazenar não são sequências simples do mesmo tipo de dados, mas sim conjuntos de elementos diferentes, contendo tipos de dados diferentes.

### 3.5.1 Estruturas de dados

Uma estrutura de dados é um grupo de elementos de dados agrupados sob um único nome. Esses elementos de dados, conhecidos como membros, podem ter diferentes tipos e tamanhos. As estruturas de dados são declaradas em C++ usando a seguinte sintaxe:

```
struct structure_name {  
    member_type1 member_name1;  
    member_type2 member_name2;  
    member_type3 member_name3;  
    .  
    .  
} object_names;
```

onde `structure_name` é um nome para o tipo de estrutura, `object_name` pode ser um conjunto de identificadores válidos para objetos que tenham o tipo desta estrutura. Dentro das chaves `{ }` existe uma lista com os membros de dados, cada um é especificado com um tipo e um identificador válido como o seu nome.

A primeira coisa que temos que saber é que uma estrutura de dados cria um novo tipo: quando uma estrutura de dados é declarada, um novo tipo de identificador especificado como

`structure_name` é criado e pode ser usado no resto do programa como se fosse qualquer outro tipo. Por exemplo:

```
1 struct product {  
2     int weight;  
3     float price;  
4 } ;  
5  
6 product apple;  
7 product banana, melon;
```

Temos primeiro declarado uma estrutura chamada `product` com dois membros: `weight` (peso) e `price` (preço), cada um de um tipo de dado diferente. Usamos, em seguida, este nome de estrutura (`product`) para declarar três objetos: `apple`, `banana` e `melon` como teria feito com qualquer tipo de dados fundamental.

Uma vez declarado, `product` tornou-se um novo tipo de dado válido como os fundamentais `int`, `char` ou `short` e, a partir desse ponto, somos capazes de declarar objetos (variáveis) desse novo tipo de composto, tal como temos feito com, `banana`, `maçã` e `melão`.

Logo no final da declaração `struct`, e antes do ponto e vírgula final, podemos usar o campo opcional `object_name` para declarar objetos diretamente. Por exemplo, podemos também declarar a estrutura de objetos de `apple`, `banana` e `melon` no momento em que definimos o tipo de estrutura de dados:

```
1 struct product {  
2     int weight;  
3     float price;  
4 } apple, banana, melon;
```

É importante diferenciar claramente entre o que é o nome do tipo de estrutura e o que é um objeto (variável) que tem esse tipo de estrutura. Podemos instanciar muitos objetos (variáveis, ou seja, como, `banana`, `maçã` e `melão`) de um tipo de estrutura simples (produto).

Uma vez que tenhamos declarado nossos três objetos de um tipo de estrutura determinada (`banana`, `maçã` e `melão`), podemos atuar diretamente com os seus membros. Para isso, usamos um ponto (.) inserido entre o nome do objeto e o nome do membro. Por exemplo, poderíamos operar com qualquer um desses elementos como se fossem variáveis de seus respectivos tipos:

```
1 apple.weight  
2 apple.price  
3 banana.weight  
4 banana.price  
5 melon.weight  
6 melon.price
```

Cada um deles contém o tipo de dados correspondente ao membro que se referem: `apple.weight`, `banana.weight` e `melon.weight` são do tipo `int`, enquanto, `banana.price`, `apple.price` e `melon.price` são do tipo `float`.

Vamos ver um exemplo real, onde você pode ver como um tipo de estrutura pode ser usado da mesma forma como tipos fundamentais:

```

1  // example about structures
2  #include <iostream>
3  #include <string>
4  #include <sstream>
5  using namespace std;
6
7  struct movies_t {
8      string title;
9      int year;
10 } mine, yours;
11
12 void printmovie (movies_t movie);
13
14 int main (){
15     string mystr;
16
17     mine.title = "2001 A Space
18 Odyssey";
19     mine.year = 1968;
20
21     cout << "Enter title: ";
22     getline (cin,yours.title);
23     cout << "Enter year: ";
24     getline (cin,mystr);
25     stringstream(mystr) >>
26     yours.year;
27
28     cout << "My favorite movie is:\n
29 ";
30     printmovie (mine);
31     cout << "And yours is:\n ";
32     printmovie (yours);
33     return 0;
34 }
35
36 void printmovie (movies_t movie){
37     cout << movie.title;
38     cout << " (" << movie.year <<
39 ") \n";
40 }

```

```

Enter title: Alien
Enter year: 1979

My favorite movie is:
  2001 A Space Odyssey (1968)
And yours is:
  Alien (1979)

```

O exemplo mostra como podemos usar os membros de um objeto como variáveis normais. Por exemplo, o `yours.year` membro é uma variável válida do tipo `int`, e `mine.title` é uma variável válida do tipo `string`.

Os objetos `mine` e `yours` também podem ser tratados como variáveis do tipo `movies_t`, por exemplo, nós os passamos para a função `printmovie` como teríamos feito com variáveis comuns. Portanto, uma das vantagens de estruturas é que podemos fazer referência aos seus membros individualmente ou para toda a estrutura como um bloco com apenas um identificador.

Estruturas de dados é um recurso que pode ser usado para representar bancos de dados, especialmente se considerarmos a possibilidade de construção de matrizes:

```

1  // array of structures
2  #include <iostream>

```

```

Enter title: Blade Runner
Enter year: 1982

```

<pre> 3  #include &lt;string&gt; 4  #include &lt;sstream&gt; 5  using namespace std; 6 7  #define N_MOVIES 3 8 9  struct movies_t { 10     string title; 11     int year; 12 } films [N_MOVIES]; 13 14 void printmovie (movies_t movie); 15 16 int main () { 17     string mystr; 18     int n; 19 20     for (n=0; n&lt;N_MOVIES; n++) 21     { 22         cout &lt;&lt; "Enter title: "; 23         getline (cin,films[n].title); 24         cout &lt;&lt; "Enter year: "; 25         getline (cin,mystr); 26         stringstream(mystr)      &gt;&gt; 27         films[n].year; 28     } 29 30     cout &lt;&lt; "\nYou have entered these movies:\n"; 31     for (n=0; n&lt;N_MOVIES; n++) 32         printmovie (films[n]); 33     return 0; 34 } 35 36 void printmovie (movies_t movie) { 37     cout &lt;&lt; movie.title; 38     cout &lt;&lt; " (" &lt;&lt; movie.year &lt;&lt; ")\n"; 39 } </pre>	<pre> Enter title: Matrix Enter year: 1999 Enter title: Taxi Driver Enter year: 1976  You have entered these movies: Blade Runner (1982) Matrix (1999) Taxi Driver (1976) </pre>
--	--

### 3.5.2 Ponteiros para estruturas

Como qualquer outro tipo de dados, as estruturas também podem ser apontadas pelo seu próprio tipo de ponteiro:

```

1 struct movies_t {
2     string title;
3     int year;
4 };
5
6 movies_t amovie;
7 movies_t * pmovie;

```

Aqui `amovie` é um objeto de `movies_t`, uma estrutura, e `pmovie` é um ponteiro para apontar objetos do tipo estrutura `movies_t`. Então, o seguinte código também seria válido:

```
pmovie = &amovie;
```

ao valor do ponteiro `pmovie` é atribuído uma referência ao objeto `amovie` (seu endereço de memória).

Agora vamos analisar outro exemplo que inclui ponteiros, e também serve para introduzir um novo operador: o operador seta (`->`):

<pre>1 // pointers to structures 2 #include &lt;iostream&gt; 3 #include &lt;string&gt; 4 #include &lt;sstream&gt; 5 using namespace std; 6 7 struct movies_t { 8     string title; 9     int year; 10 }; 11 12 int main () { 13     string mystr; 14 15     movies_t amovie; 16     movies_t * pmovie; 17     pmovie = &amp;amovie; 18 19     cout &lt;&lt; "Enter title: "; 20     getline (cin, pmovie-&gt;title); 21     cout &lt;&lt; "Enter year: "; 22     getline (cin, mystr); 23     (stringstream) mystr &gt;&gt; pmovie- &gt;year; 24 25     cout &lt;&lt; "\nYou have entered:\n"; 26     cout &lt;&lt; pmovie-&gt;title; 27     cout &lt;&lt; " (" &lt;&lt; pmovie-&gt;year &lt;&lt; ")\n"; 28 29     return 0; 30 }</pre>	<pre>Enter title: Invasion of the body snatchers Enter year: 1978  You have entered: Invasion of the body snatchers (1978)</pre>
---	--

O código anterior inclui uma introdução importante: o operador seta (`->`). Este é um operador dereferência que é usado exclusivamente com ponteiros para objetos com membros. Este operador serve para acessar um membro de um objeto para o qual temos uma referência. No exemplo que usamos:

```
pmovie->title
```

que, para todos os efeitos, é equivalente a:

```
(*pmovie).title
```

Ambas as expressões `pmovie->title` e `(*pmovie).title` são válidas e significam que estamos avaliando o membro `title` da estrutura de dados **apontados por** um ponteiro chamado `pmovie`. Deve ser claramente diferenciado de:

```
*pmovie.title
```

que é equivalente a:

```
*(pmovie.title)
```

e isso poderia acessar o valor apontado por um ponteiro membro hipotético chamado `title` do objeto estrutura `pmovie` (que neste caso não seria um ponteiro e sim um valor). O quadro a seguir resume as combinações possíveis de ponteiros e os membros da estrutura:

Expressão	O que é avaliado	Equivalente
<code>a.b</code>	Membro <code>b</code> do objeto <code>a</code>	
<code>a-&gt;b</code>	Membro <code>b</code> do objeto apontado por <code>a</code>	<code>(*a).b</code>
<code>*a.b</code>	Valor apontado pelo membro <code>b</code> do objeto <code>a</code>	<code>*(a.b)</code>

### 3.5.3 Estruturas aninhadas

Estruturas também podem ser aninhadas de modo que um elemento de uma estrutura também pode ser outra estrutura.

```
1 struct movies_t {  
2     string title;  
3     int year;  
4 };  
5  
6 struct friends_t {  
7     string name;  
8     string email;  
9     movies_t favorite_movie;  
10    } charlie, maria;  
11  
12 friends_t * pfriends = &charlie;
```

Após a declaração anterior, podemos usar qualquer uma das seguintes expressões:

```
1 charlie.name  
2 maria.favorite_movie.title  
3 charlie.favorite_movie.year  
4 pfriends->favorite_movie.year
```

(onde, aliás, as duas últimas expressões se referem a um mesmo membro).

## 3.6 Outros tipos de dados

### 3.6.1 Tipo de dados definido (typedef)

O C++ permite a definição de nossos próprios tipos de dados baseados em tipos existentes (int, float, etc). Podemos fazer isto utilizando a palavra-chave `typedef`, cujo formato é:

```
typedef existing_type new_type_name ;
```

onde `existing_type` é um tipo fundamental ou composto e `new_type_name` é o nome para o novo tipo que estamos definindo. Por exemplo:

```
1 typedef char C;
2 typedef unsigned int WORD;
3 typedef char * pChar;
4 typedef char field [50];
```

Neste caso, foi definido quatro tipos de dados: `C`, `WORD`, `pChar` e `field` como `unsigned`, `int`, `char *`, `char` e `char [50]`, respectivamente, que poderíamos perfeitamente usar em declarações posteriores como qualquer outro tipo de dados:

```
1 C mychar, anotherchar, *ptc1;
2 WORD myword;
3 pChar ptc2;
4 field name;
```

`typedef` não cria outros tipos de dados. Ela só cria sinônimos dos tipos existentes. Isso significa que o objeto `myword` pode ser considerado um `unsigned int` ou `WORD`, já que ambos são, de fato, do mesmo tipo.

`typedef` pode ser útil para definir um *alias* (apelido) para um tipo que é usado frequentemente dentro do programa. Também é útil para definir os tipos quando tivermos que alterar o tipo de dado de um conjunto de variáveis em versões posteriores do nosso programa, ou se um tipo de dado cujo nome é muito longo ou confuso.

### 3.6.2 Uniões (union)

Uniões permitem uma mesma porção de memória a ser acessada como diferentes tipos de dados, uma vez que todos eles estão, de fato, no mesmo local da memória. Sua declaração e utilização são semelhantes às estruturas, mas sua funcionalidade é totalmente diferente:

```
union union_name {
    member_type1 member_name1;
    member_type2 member_name2;
    member_type3 member_name3;
    .
    .
} object_names;
```



Todos os elementos da declaração `union` ocupam o mesmo espaço físico na memória. Seu tamanho é assumido como sendo o do maior elemento da declaração. Por exemplo:

```
1 union mytypes_t {
2     char c;
3     int i;
4     float f;
5 } mytypes;
```

define três elementos:

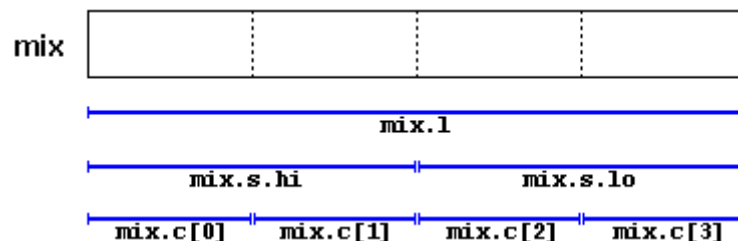
```
1 mytypes.c
2 mytypes.i
3 mytypes.f
```

cada um com um tipo de dado diferente. Uma vez que todos eles estão se referindo ao mesmo local na memória, a modificação de um dos elementos afetará o valor de todos eles. **Não podemos armazenar valores diferentes em cada uma das variáveis ao mesmo tempo.**

Um dos usos das uniões é a possibilidade de unir um tipo elementar de dado com um conjunto de outros elementos ou estruturas de menor dimensão. Por exemplo:

```
1 union mix_t {
2     long l;
3     struct {
4         short hi;
5         short lo;
6     } s;
7     char c[4];
8 } mix;
```

define três nomes que nos permitem acessar o mesmo grupo de 4 bytes: `mix.l`, `mix.s` e `mix.c`. Podemos usá-lo, como se fossem um único tipo de dados, como se fosse um `long`, como se fossem dois `short` ou como uma matriz com 4 elementos `char`, respectivamente. Esta união, para um sistema *little-endian* (maioria das plataformas PC), pode ser representada como:



O alinhamento exato e ordem dos membros de uma `union` na memória dependem da plataforma. Portanto, esteja ciente das possíveis questões de portabilidade com este tipo de uso.

### 3.6.3 Uniões anônimas

Em C++, temos a opção de declarar uniões anônimas. Se declarar uma união sem qualquer nome, a união será anônima e seremos capazes de acessar seus membros diretamente pelo nome. Por exemplo, olhe a diferença entre essas duas declarações de estrutura:

estrutura com união regular	estrutura com união anônima
<pre>struct {     char title[50];     char author[50];     union {         float dollars;         int yens;     } price; } book;</pre>	<pre>struct {     char title[50];     char author[50];     union {         float dollars;         int yens;     }; } book;</pre>

A única diferença entre os dois códigos é que no primeiro foi dado um nome para a união (`price`) e na segunda não foi inserido um nome. A diferença é vista quando acessamos os membros `dollars` e `yens` desse objeto. Para o primeiro código seria:

```
1 book.price.dollars
2 book.price.yens
```

Enquanto para o segundo código, seria:

```
1 book.dollars
2 book.yens
```

Mais uma vez devo lembrar que, porque é uma união e não uma estrutura, os membros `dollars` e `yens` ocupam o mesmo espaço físico na memória de modo que não pode ser usado para armazenar dois valores diferentes simultaneamente. Você pode definir um valor para o preço `dollars` ou em `yens`, mas não em ambos ao mesmo tempo.

### 3.6.4 Enumerações (enum)

Enumerações criam novos tipos de dados para conter algo diferente (não limitado aos valores fundamentais de dados). Sua forma é a seguinte:

```
enum enumeration_name {
    value1,
    value2,
    value3,
    .
    .
} object_names;
```

Por exemplo, poderíamos criar um novo tipo de variável chamada `colors_t` para armazenar as cores, com a seguinte declaração:

```
enum colors_t {black, blue, green, cyan, red, purple, yellow, white};
```

Observe que não incluem qualquer tipo de dado fundamental na declaração. Para dizê-lo de outra forma, criamos um novo tipo de dados a partir do zero, sem baseá-lo em qualquer outro tipo existente. Os possíveis valores que as variáveis deste novo tipo *color\_t* podem tomar é a constante de novos valores incluídos dentro de chaves. Por exemplo, uma vez que a enumeração *colors\_t* é declarada as seguintes expressões são válidas:

```
1 colors_t mycolor;  
2  
3 mycolor = blue;  
4 if (mycolor == green) mycolor = red;
```

Enumerações são tipos compatíveis com as variáveis numéricas, então, às suas constantes é sempre atribuído um valor numérico inteiro por padrão. Se não for explicitamente especificado, o equivalente ao valor inteiro possível, o primeiro valor é equivalente a 0 e as seguintes seguem uma progressão 1. Assim, em nossa enumeração *colors\_t* que definimos acima, *black* seria equivalente a 0, o *blue* seria equivalente a 1, *green* a 2 e assim por diante.

Podemos especificar explicitamente um valor inteiro para qualquer uma das constantes que a nossa enumeração assumirá. Se ao valor da constante que se segue não é dado um valor inteiro, ele é automaticamente assumido o mesmo valor do anterior mais um. Por exemplo:

```
1 enum months_t { january=1, february, march, april,  
2               may, june, july, august,  
3               september, october, november, december} y2k;
```

Neste caso, a variável *y2k* de *months\_t* pode conter qualquer um dos 12 possíveis valores que vão de janeiro até dezembro, e, neste caso, são equivalentes aos valores entre 1 e 12 (não entre 0 e 11, já fizemos *january=1*, no início da declaração).

# Anexo A (alguns cabeçalhos e bibliotecas)

## A.1 Algorithms

library `<algorithm>`

### Standard Template Library: Algorithms

The header `<algorithm>` defines a collection of functions especially designed to be used on ranges of elements.

A range is any sequence of objects that can be accessed through iterators or pointers, such as an array or an instance of some of the [STL containers](#). Notice though, that algorithms operate through iterators directly on the values, not affecting in any way the structure of any possible container (it never affects the size or storage allocation of the container).

### Functions in `<algorithm>`

#### Non-modifying sequence operations:

<b>for_each</b>	Apply function to range (function template)
<b>find</b>	Find value in range (function template)
<b>find_if</b>	Find element in range (function template)
<b>find_end</b>	Find last subsequence in range (function template)
<b>find_first_of</b>	Find element from set in range (function template)
<b>adjacent_find</b>	Find equal adjacent elements in range (function template)
<b>count</b>	Count appearances of value in range (function template)
<b>count_if</b>	Return number of elements in range satisfying condition (function template)
<b>mismatch</b>	Return first position where two ranges differ (function template)
<b>equal</b>	Test whether the elements in two ranges are equal (function template)
<b>search</b>	Find subsequence in range (function template)
<b>search_n</b>	Find succession of equal values in range (function template)

#### Modifying sequence operations:

<b>copy</b>	Copy range of elements (function template)
<b>copy_backward</b>	Copy range of elements backwards (function template)
<b>swap</b>	Exchange values of two objects (function template)
<b>swap_ranges</b>	Exchange values of two ranges (function template)
<b>iter_swap</b>	Exchange values of objects pointed by two iterators (function template)
<b>transform</b>	Apply function to range (function template)
<b>replace</b>	Replace value in range (function template)
<b>replace_if</b>	Replace values in range (function template)
<b>replace_copy</b>	Copy range replacing value (function template)

<b>replace_copy_if</b>	Copy range replacing value (function template)
<b>fill</b>	Fill range with value (function template)
<b>fill_n</b>	Fill sequence with value (function template)
<b>generate</b>	Generate values for range with function (function template)
<b>generate_n</b>	Generate values for sequence with function (function template)
<b>remove</b>	Remove value from range (function template)
<b>remove_if</b>	Remove elements from range (function template)
<b>remove_copy</b>	Copy range removing value (function template)
<b>remove_copy_if</b>	Copy range removing values (function template)
<b>unique</b>	Remove consecutive duplicates in range (function template)
<b>unique_copy</b>	Copy range removing duplicates (function template)
<b>reverse</b>	Reverse range (function template)
<b>reverse_copy</b>	Copy range reversed (function template)
<b>rotate</b>	Rotate elements in range (function template)
<b>rotate_copy</b>	Copy rotated range (function template)
<b>random_shuffle</b>	Rearrange elements in range randomly (function template)
<b>partition</b>	Partition range in two (function template)
<b>stable_partition</b>	Partition range in two - stable ordering (function template)

#### Sorting:

<b>sort</b>	Sort elements in range (function template)
<b>stable_sort</b>	Sort elements preserving order of equivalents (function template)
<b>partial_sort</b>	Partially Sort elements in range (function template)
<b>partial_sort_copy</b>	Copy and partially sort range (function template)
<b>nth_element</b>	Sort element in range (function template)

#### Binary search (operating on sorted ranges):

<b>lower_bound</b>	Return iterator to lower bound (function template)
<b>upper_bound</b>	Return iterator to upper bound (function template)
<b>equal_range</b>	Get subrange of equal elements (function template)
<b>binary_search</b>	Test if value exists in sorted array (function template)

#### Merge (operating on sorted ranges):

<b>merge</b>	Merge sorted ranges (function template)
<b>inplace_merge</b>	Merge consecutive sorted ranges (function template)
<b>includes</b>	Test whether sorted range includes another sorted range (function template)
<b>set_union</b>	Union of two sorted ranges (function template)
<b>set_intersection</b>	Intersection of two sorted ranges (function template)

<b>set_difference</b>	Difference of two sorted ranges (function template)
<b>set_symmetric_difference</b>	Symmetric difference of two sorted ranges (function template)

#### Heap:

<b>push_heap</b>	Push element into heap range (function template)
<b>pop_heap</b>	Pop element from heap range (function template)
<b>make_heap</b>	Make heap from range (function template)
<b>sort_heap</b>	Sort elements of heap (function template)

#### Min/max:

<b>min</b>	Return the lesser of two arguments (function template)
<b>max</b>	Return the greater of two arguments (function template)
<b>min_element</b>	Return smallest element in range (function template)
<b>max_element</b>	Return largest element in range (function template)
<b>lexicographical_compare</b>	Lexicographical less-than comparison (function template)
<b>next_permutation</b>	Transform range to next permutation (function template)
<b>prev_permutation</b>	Transform range to previous permutation (function template)

## A.2 complex

header

### Complex numbers library

The complex library implements the `complex` class to contain complex numbers in cartesian form and several functions and overloads to operate with them:

### Classes

<b>complex</b>	Complex number class (class template)
----------------	---------------------------------------

### Functions

#### Complex values:

<b>real</b>	Return real part of complex (function template)
<b>imag</b>	Return imaginary part of complex (function template)
<b>abs</b>	Return absolute value of complex (function template)
<b>arg</b>	Return phase angle of complex (function template)
<b>norm</b>	Return norm of complex number (function template)
<b>conj</b>	Return complex conjugate (function template)
<b>polar</b>	Return complex from polar components (function template)

#### Transcendentals overloads:

<b>cos</b>	Return cosine of complex (function template)
<b>cosh</b>	Return hyperbolic cosine of complex (function template)
<b>exp</b>	Return exponential of complex (function template)
<b>log</b>	Return natural logarithm of complex (function template)
<b>log10</b>	Return common logarithm of complex (function template)
<b>pow</b>	Return complex power (function template)
<b>sin</b>	Return sine of complex (function template)
<b>sinh</b>	Return hyperbolic sine of complex (function template)
<b>sqrt</b>	Return square root of complex (function template)
<b>tan</b>	Return tangent of complex (function template)
<b>tanh</b>	Return hyperbolic tangent of complex (function template)

#### Operator overloads:

<b>complex operators</b>	Complex number operators (functions)
--------------------------	--------------------------------------

## A.3 cmath (math.h)

header

### C numerics library

`cmath` declares a set of functions to compute common mathematical operations and transformations:

#### Trigonometric functions:

<b>cos</b>	Compute cosine (function)
<b>sin</b>	Compute sine (function)
<b>tan</b>	Compute tangent (function)
<b>acos</b>	Compute arc cosine (function)
<b>asin</b>	Compute arc sine (function)
<b>atan</b>	Compute arc tangent (function)
<b>atan2</b>	Compute arc tangent with two parameters (function)

#### Hyperbolic functions:

<b>cosh</b>	Compute hyperbolic cosine (function)
<b>sinh</b>	Compute hyperbolic sine (function)
<b>tanh</b>	Compute hyperbolic tangent (function)

#### Exponential and logarithmic functions:

<b>exp</b>	Compute exponential function (function)
<b>frexp</b>	Get significand and exponent (function)
<b>ldexp</b>	Generate number from significand and exponent (function)

<b>log</b>	Compute natural logarithm (function)
<b>log10</b>	Compute common logarithm (function)
<b>modf</b>	Break into fractional and integral parts (function)

#### Power functions

<b>pow</b>	Raise to power (function)
<b>sqrt</b>	Compute square root (function)

#### Rounding, absolute value and remainder functions:

<b>ceil</b>	Round up value (function)
<b>fabs</b>	Compute absolute value (function)
<b>floor</b>	Round down value (function)
<b>fmod</b>	Compute remainder of division (function)

## A.4 cstdio (stdio.h)

header

### C library to perform Input/Output operations

Input and Output operations can also be performed in C++ using the **C Standard Input and Output Library** (**cstdio**, known as `stdio.h` in the C language). This library uses what are called *streams* to operate with physical devices such as keyboards, printers, terminals or with any other type of files supported by the system. Streams are an abstraction to interact with these in an uniform way; All streams have similar properties independently of the individual characteristics of the physical media they are associated with.

Streams are handled in the `cstdio` library as pointers to **FILE** objects. A pointer to a **FILE** object uniquely identifies a stream, and is used as a parameter in the operations involving that stream.

There also exist three standard streams: `stdin`, `stdout` and `stderr`, which are automatically created and opened for all programs using the library.

### Stream properties

Streams have some properties that define which functions can be used on them and how these will treat the data input or output through them. Most of these properties are defined at the moment the stream is associated with a file (opened) using the <http://www.cplusplus.com/fopen> function:

#### Read/Write Access

Specifies whether the stream has read or write access (or both) to the physical media they are associated with.

#### Text / Binary

Text streams are thought to represent a set of text lines, each one ending with a new-line character. Depending on the environment where the application is run some character translation may occur with text streams to adapt some special characters to the text file specifications of the environment. A binary stream, on the other hand, is a sequence of characters written or read from the physical media with no translation, having a one-to-one correspondence with the characters read or written to the stream.

#### Buffer

A buffer is a block of memory where data is accumulated before being physically read or written to the associated file or device. Streams can be either *fully buffered*, *line*



*buffered* or *unbuffered*. On fully buffered streams, data is read/written when the buffer is filled, on line buffered streams this happens when a new-line character is encountered, and on unbuffered streams characters are intended to be read/written as soon as possible.

## Indicators

Streams have certain internal indicators that specify their current state and which affect the behavior of some input and output operations performed on them:

Error indicator

This indicator is set when an error has occurred in an operation related to the stream.

This indicator can be checked with the [ferror](#) function, and can be reset by calling either to [clearerr](#) or to any repositioning function ([rewind](#), [fseek](#) and [fsetpos](#)).

End-Of-File indicator

When set, indicates that the last reading or writing operation performed with the stream reached the *End of File*. It can be checked with the [feof](#) function, and can be reset by calling either to [clearerr](#) or to any repositioning function ([rewind](#), [fseek](#) and [fsetpos](#)).

Position indicator

It is an internal pointer of each stream which points to the next character to be read or written in the next I/O operation. Its value can be obtained by the [ftell](#) and [fgetpos](#) functions, and can be changed using the repositioning functions [rewind](#), [fseek](#) and [fsetpos](#).

## Functions

Operations on files:

<a href="#">remove</a>	Remove file ( <a href="#">function</a> )
<a href="#">rename</a>	Rename file ( <a href="#">function</a> )
<a href="#">tmpfile</a>	Open a temporary file ( <a href="#">function</a> )
<a href="#">tmpnam</a>	Generate temporary filename ( <a href="#">function</a> )

File access:

<a href="#">fclose</a>	Close file ( <a href="#">function</a> )
<a href="#">fflush</a>	Flush stream ( <a href="#">function</a> )
<a href="#">fopen</a>	Open file ( <a href="#">function</a> )
<a href="#">freopen</a>	Reopen stream with different file or mode ( <a href="#">function</a> )
<a href="#">setbuf</a>	Set stream buffer ( <a href="#">function</a> )
<a href="#">setvbuf</a>	Change stream buffering ( <a href="#">function</a> )

Formatted input/output:

<a href="#">fprintf</a>	Write formatted output to stream ( <a href="#">function</a> )
<a href="#">fscanf</a>	Read formatted data from stream ( <a href="#">function</a> )
<a href="#">printf</a>	Print formatted data to stdout ( <a href="#">function</a> )
<a href="#">scanf</a>	Read formatted data from stdin ( <a href="#">function</a> )
<a href="#">sprintf</a>	Write formatted data to string ( <a href="#">function</a> )
<a href="#">sscanf</a>	Read formatted data from string ( <a href="#">function</a> )
<a href="#">vfprintf</a>	Write formatted variable argument list to stream ( <a href="#">function</a> )

<b>vprintf</b>	Print formatted variable argument list to stdout (function)
<b>vsprintf</b>	Print formatted variable argument list to string (function)

#### Character input/output:

<b>fgetc</b>	Get character from stream (function)
<b>fgets</b>	Get string from stream (function)
<b>fputc</b>	Write character to stream (function)
<b>fputs</b>	Write string to stream (function)
<b>getc</b>	Get character from stream (function)
<b>getchar</b>	Get character from stdin (function)
<b>gets</b>	Get string from stdin (function)
<b>putc</b>	Write character to stream (function)
<b>putchar</b>	Write character to stdout (function)
<b>puts</b>	Write string to stdout (function)
<b>ungetc</b>	Unget character from stream (function)

#### Direct input/output:

<b>fread</b>	Read block of data from stream (function)
<b>fwrite</b>	Write block of data to stream (function)

#### File positioning:

<b>fgetpos</b>	Get current position in stream (function)
<b>fseek</b>	Reposition stream position indicator (function)
<b>fsetpos</b>	Set position indicator of stream (function)
<b>ftell</b>	Get current position in stream (function)
<b>rewind</b>	Set position indicator to the beginning (function)

#### Error-handling:

<b>clearerr</b>	Clear error indicators (function)
<b>feof</b>	Check End-of-File indicator (function)
<b>ferror</b>	Check error indicator (function)
<b>perror</b>	Print error message (function)

## Macros

<b>EOF</b>	End-of-File (constant)
<b>FILENAME_MAX</b>	Maximum length of file names (constant)
<b>NULL</b>	Null pointer (constant)
<b>TMP_MAX</b>	Number of temporary files (constant)

And also `_IOBF`, `_IOLBF`, `_IONBF`, `BUFSIZ`, `FOPEN_MAX`, `L_tmpnam`, `SEEK_CUR`, `SEEK_END` and `SEEK_SET`, each described with its corresponding function.

## Types

<b>FILE</b>	Object containing information to control a stream (type)
<b>fpos_t</b>	Object containing information to specify a position within a file (type)
<b>size_t</b>	Unsigned integral type (type)

## A.5 cstdlib (stdlib.h)

header

### C Standard General Utilities Library

This header defines several general purpose functions, including dynamic memory management, random number generation, communication with the environment, integer arithmetics, searching, sorting and converting.

## Functions

### String conversion:

<b>atof</b>	Convert string to double (function)
<b>atoi</b>	Convert string to integer (function)
<b>atol</b>	Convert string to long integer (function)
<b>strtod</b>	Convert string to double (function)
<b>strtol</b>	Convert string to long integer (function)
<b>strtoul</b>	Convert string to unsigned long integer (function)

### Pseudo-random sequence generation:

<b>rand</b>	Generate random number (function)
<b>srand</b>	Initialize random number generator (functions)

### Dynamic memory management:

<b>calloc</b>	Allocate space for array in memory (function)
<b>free</b>	Deallocate space in memory (function)
<b>malloc</b>	Allocate memory block (function)
<b>realloc</b>	Reallocate memory block (function)

### Environment:

<b>abort</b>	Abort current process (function)
<b>atexit</b>	Set function to be executed on exit (function)
<b>exit</b>	Terminate calling process (function)
<b>getenv</b>	Get environment string (function)
<b>system</b>	Execute system command (function)

### Searching and sorting:

<b>bsearch</b>	Binary search in array (function)
----------------	-----------------------------------

<b>qsort</b>	Sort elements of array (function)
--------------	-----------------------------------

#### Integer arithmethics:

<b>abs</b>	Absolute value (function)
<b>div</b>	Integral division (function)
<b>labs</b>	Absolute value (function)
<b>ldiv</b>	Integral division (function)

#### Multibyte characters:

<b>mblen</b>	Get length of multibyte character (function)
<b>mbtowc</b>	Convert multibyte character to wide character (function)
<b>wctomb</b>	Convert wide character to multibyte character (function)

#### Multibyte strings:

<b>mbstowcs</b>	Convert multibyte string to wide-character string (function)
<b>wcstombs</b>	Convert wide-character string to multibyte string (function)

## Macros

<b>EXIT_FAILURE</b>	Failure termination code (macro)
<b>EXIT_SUCCESS</b>	Success termination code (macro)
<b>MB_CUR_MAX</b>	Maximum size of multibyte characters (macro)
<b>NULL</b>	Null pointer (macro)
<b>RAND_MAX</b>	Maximum value returned by rand (macro)

## Types

<b>div_t</b>	Structure returned by div (type)
<b>ldiv_t</b>	Structure returned by div and ldiv (type)
<b>size_t</b>	Unsigned integral type (type)

## A.6 ctime (time.h)

header

### C Time Library

This header file contains definitions of functions to get and manipulate date and time information.

## Functions

### Time manipulation

<b>clock</b>	Clock program (function)
<b>difftime</b>	Return difference between two times (function)

<b>mktime</b>	Convert tm structure to time_t (function)
<b>time</b>	Get current time (function)

#### Conversion:

<b>asctime</b>	Convert tm structure to string (function)
<b>ctime</b>	Convert time_t value to string (function)
<b>gmtime</b>	Convert time_t to tm as UTC time (function)
<b>localtime</b>	Convert time_t to tm as local time (function)
<b>strftime</b>	Format time to string (function)

## Macros

<b>CLOCKS_PER_SEC</b>	Clock ticks per second (macro)
<b>NULL</b>	Null pointer (macro)

## types

<b>clock_t</b>	Clock type (type)
<b>size_t</b>	Unsigned integral type (type)
<b>time_t</b>	Time type (type)
<b>struct tm</b>	Time structure (type)