

C++

Introdução a STL

Instrutor :
Bruno Masquio
Material elaborado por :
Caio César F. A. Lima,
Bianca Rosa e Giancarlo França



História

Em 1979, Bjarne Stroustrup, do Bell Labs, começou a desenvolver a linguagem "C with Classes" que se tornaria conhecida formalmente como C++ em 1983.



História

Em Outubro de 1985, saiu a primeira versão comercial junto com a primeira edição do livro: "The C++ Programming Language".



C++ é...

- Compilada
- Tipagem "fraca" e estática
- Multi-paradigma
- Multi-plataforma
- Compatível com C
- Case-sensitive.



Hello World!

```
#include <iostream>
using namespace std;

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



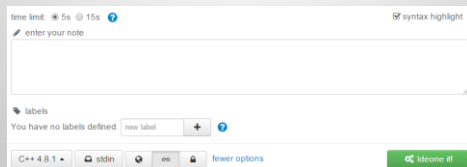
Rodando um programa no prompt

1. Escreva seu programa (Acrescente ou não, não é necessário)
2. Compile-o no seu compilador de preferência
3. Abra o prompt de comando
4. Navegue até a pasta onde o executável está (comando cd)
5. Digite o nome do seu executável e pressione ENTER.

```
C:\Users\UERJ>cd "Documents\C++\STL"
C:\Users\UERJ\Documents\C++\STL>helloworld.exe
Hello World!
C:\Users\UERJ\Documents\C++\STL>
```

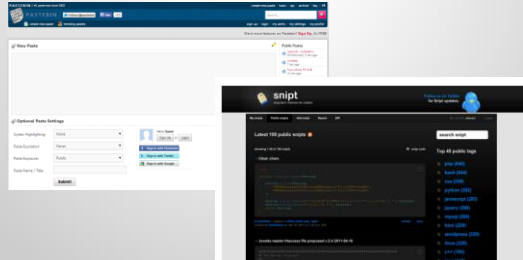
Rodando um programa no Ideone

1. Acesse <http://ideone.com/>
2. Entre em sua conta cadastrada (opcional, recomendado)
3. Selecione a linguagem C++ 5.1
4. Escreva seu programa
5. Clique no botão "stdin" para abrir a caixa de texto em que será especificado o input (opcional)
6. Clique no botão "Ideone it!"



Armazenando códigos on-line

- Pastebin (<http://pastebin.com/>)
- Snipt (<http://snipt.org/>) (Autenticação via Twitter)



Tipos Primitivos

- **int** : inteiros, 4 bytes, de -2.147.483.648 a 2.147.483.647.
- **float** : número de ponto flutuante, 4 bytes, ~ 7 dígitos de precisão.
- **double** : número de ponto flutuante com precisão dupla, 8 bytes, ~15 dígitos de precisão.
- **char** : caractere ou inteiro pequeno, 1 byte.
- **bool** : verdadeiro ou falso, 1 byte.

Modificadores

- **short**
- **long**
- **unsigned**
- **static**
- **register**
- **extern**

Modificadores

Operadores Lógicos

&& e
|| ou
! não

Operadores Relacionais

< menor
<= menor ou igual
> maior
>= maior ou igual
!= diferente

Atribuição =

Outros Atribuidores:

+= **-=** ***=** **/=** **%=**
&= **|=** **^=** **<<=** **>>=**

Operadores Aritméticos

+ **-** ***** **/** **%**
(Soma, subtração, multiplicação, Divisão e resto da divisão inteira)

Operadores de Bits

& bitwise AND
| bitwise OR
^ bitwise XOR
<< left shift
>> right shift

Operadores de Incremento

++ auto incremento
-- auto decremento

Funções

Tipo de retorno

Lista de parâmetros

Valor default (C++)

```
int fatorial(int n, bool iterativo = false) {
    if (iterativo) {
        int i, res = 1;
        for(i=2; i<=n; ++i) res = res * i;
        return res;
    } else {
        if (n == 1) return 1;
        return n*fatorial(n-1);
    }
}
```

↑
Recursão

Estruturas de Repetição

```
for (int i = 0; i < max; i++) {  
    //do amazing stuff here  
}
```

```
while (hungry) {  
    eat();  
}
```

```
do {  
    eat();  
} while (hungry);
```

```
while (hungry) {  
    if (fat) break;  
}
```

```
do {  
    eat();  
    if (onADiet) continue;  
    eatDessert();  
} while (hungry);
```

Arrays e Ponteiros

```
int meuArray[5]; /*Inicializa array vazia*/
```

OBS: Se `meuArray` for `global`, `meuArray = {0, 0, 0, 0, 0}`.
Caso contrário, as posições conterão **lixo de memória**!

```
int meuArray[5] = {0, 7, 7, 3, 4};  
/*Inicializa array com elementos predefinidos*/
```

OBS: A quantidade de elementos **não pode exceder** o tamanho (no caso 5)!

```
int meuArray[] = {0, 7, 7, 3, 4};  
/*Inicializa array com elementos predefinidos*/
```

OBS: É **necessário** definir os elementos caso não seja especificado o tamanho!

Arrays e Ponteiros

```
int meuArray[5];
```



Memória interpretada como `ints`. (números escolhidos ao acaso)

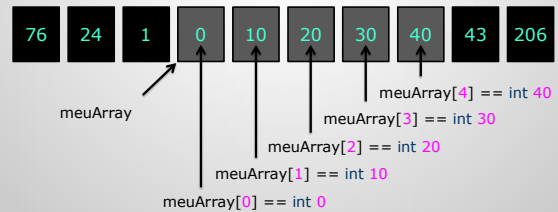
Declarando essa `array` estamos alocando espaço **sequencial** suficiente para armazenar **5 ints**.

Se `meuArray` for `array global`:



Arrays e Ponteiros

```
int meuArray[5];  
for (int i = 0; i < 5; ++i) {  
    meuArray[i] = i*10;  
}
```



Arrays e Ponteiros

```
int meuArray[5];  
for (int i = 0; i < 5; ++i) {  
    meuArray[i] = i*10;  
}
```



`meuArray`

Uma vez alocada a `array`, `meuArray` é um **PONTEIRO CONSTANTE** para o primeiro elemento da mesma.

Arrays e Ponteiros

```
int meuArray[5];  
for (int i = 0; i < 5; ++i) {  
    meuArray[i] = i*10;  
}
```



`meuArray`

`meuArray+1`

`meuArray+4`

Arrays e Ponteiros

```
int* meuArray = (int*) malloc(5*sizeof(int));
for (int i = 0; i < 5; ++i) {
    *(meuArray+i) = i*10;
}
```

Entendendo o código acima...

`meuArray` é um ponteiro para `int` que recebe o endereço retornado pela função `malloc()`, convertido (*typecast*) para um ponteiro para `int`.

A função `malloc(N)` retorna um ponteiro para `void` correspondente ao `começo` do bloco de tamanho `N` alocado. Nesse caso, é um bloco de tamanho $5 \times \text{Tamanho de Int} = 5 \times 4 = 20 \text{ bytes}$.

Arrays e Ponteiros

Um ponteiro para... **VOID?**

`void`, em C e C++, representa a ausência de tipo.

Um ponteiro para `void` é um ponteiro para um endereço **sem um tamanho** associado. Isso significa que o “valor” dele **não** pode ser acessado pelo **operador ***.

Ele é comumente usado como um **ponteiro genérico**, como é o caso da função `malloc()`.

OBS: As operações com ponteiros não são possíveis em ponteiros para void!

OBS 2: Void Pointers \neq Null Pointers !!

Arrays e Ponteiros

```
int* meuArray = (int*) malloc(5*sizeof(int));
for (int i = 0; i < 5; ++i) {
    *(meuArray+i) = i*10;
}
```

Voltando ao código acima...

Alocado o espaço para os 5 ints da array, acessamos cada posição usando o operador * e aritmética de ponteiros.

Lembrando:

```
meuArray[i] = i*10;           *(meuArray+i) = i*10;
```

São equivalentes!

Arrays e Ponteiros

Em C, **strings** são representadas por **arrays** de **char** com o caractere de terminação `'\0'`.

```
char str[] = "Hello world!";
```

	H	e	l	l	o		w	o	r	l	d	!	\0
str	[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]	[11]	[12]

```
char nstr[5] = {'H', 'e', 'l', 'l', 'o'}; /* Não é string! */
char nstr[6] = {'H', 'e', 'l', 'l', 'o', '\0'}; /* É string! */
```

Portanto, também é possível alocá-las **dinamicamente** com a função **malloc()**.

Arrays e Ponteiros

Também é possível criar **arrays** de mais de uma dimensão. Uma **array unidimensional** é chamada de **vetor** e uma **array bidimensional** também é chamada de **matriz**.

```
int tabelao[50][40];
```

O código acima é a declaração de uma matriz de **inteiros**, com **50 linhas** e **40 colunas**. Significa que estamos ocupando o espaço de **50x40 = 2000 ints!**

Preenchendo arrays multidimensionais:

```
for (int i = 0; i < 50; ++i) {
    for (int j = 0; j < 40; ++j) {
        tabelao[i][j] = i-j;
    }
}
```

Arrays e Ponteiros

A inicialização com elementos predefinidos se dá do mesmo jeito, também. Opcionalmente, para facilitar a legibilidade, é possível organizar por linhas:

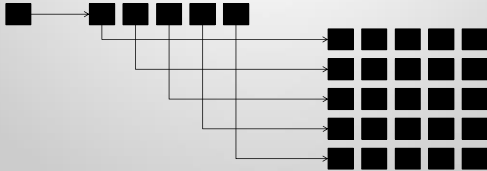
```
int q_magico[3][3] = {4, 9, 2, 3, 5, 7, 8, 1, 6};  
/*Inicializa matriz com elementos predefinidos*/
```

```
int q_magico[3][3] = {    {4, 9, 2},
                          {3, 5, 7},
                          {8, 1, 6}  };
/*Inicializa matriz com os mesmos elementos predefinidos*
```

Arrays e Ponteiros

Podemos fazer arrays multidimensionais usando **ponteiros para ponteiros**.

Considere n ponteiros apontando para o **começo** de n arrays com m elementos cada. Uma array de **duas dimensões** seria uma array com esses n ponteiros, isto é, uma **array de arrays**.



Arrays e Ponteiros

Observe o código abaixo que aloca dinamicamente e preenche o mesmo array de duas dimensões de antes (50x40):

```
int** tabelao = (int**) malloc(50*sizeof(int*));
for (int i = 0; i < 50; ++i) {
    *(tabelao+i) = (int*) malloc(40*sizeof(int));
    for (int j = 0; j < 40; ++j) {
        *(*tabelao+i)+j = i-j;
    }
}
```

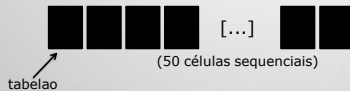
Não se assuste!
Vamos entender isso já já.



Arrays e Ponteiros

```
int** tabelao = (int**) malloc(50*sizeof(int*)); ←
for (int i = 0; i < 50; ++i) {
    *(tabelao+i) = (int*) malloc(40*sizeof(int));
    for (int j = 0; j < 40; ++j) {
        *(*tabelao+i)+j = i-j;
    }
}
```

A **primeira linha** do código declara o ponteiro **tabelao**, que apontará para o **começo** de um **segmento de memória** com espaço suficiente para **50 ponteiros** para **int**.

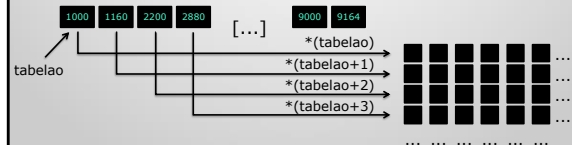


O espaço alocado **ainda não contém** dados relevantes, então o próximo **loop** tratará disso...

Arrays e Ponteiros

```
int** tabelao = (int**) malloc(50*sizeof(int*));
for (int i = 0; i < 50; ++i) {
    *(tabelao+i) = (int*) malloc(40*sizeof(int)); ←
    for (int j = 0; j < 40; ++j) {
        *(*tabelao+i)+j = i-j;
    }
}
```

Em seguida, no **loop for**, **cada um dos ponteiros (linhas)** recebe um **endereço**, que é o **começo** do espaço alocado, suficiente para **40 ints. (colunas)**

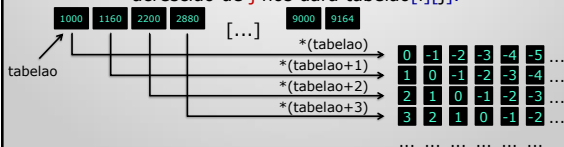


Arrays e Ponteiros

```
int** tabelao = (int**) malloc(50*sizeof(int*));
for (int i = 0; i < 50; ++i) {
    *(tabelao+i) = (int*) malloc(40*sizeof(int));
    for (int j = 0; j < 40; ++j) {
        *(*tabelao+i)+j = i-j; ←
    }
}
```

Finalmente, acessamos as **células** para alterar o **valor** de cada uma para $i-j$ (**linha-coluna**).

$*(*tabelao+i)$ é o **endereço base** da **linha i** , então isso acrescido de j nos dará **tabelao[i][j]**.



Referências em C++

A **passagem de parâmetros** padrão para **funções** é a **passagem por valor**.

Passagem por valor

A **função** recebe **cópias** dos objetos passados. Isso demanda **memória e tempo adicionais**, que podem ser **desprezíveis** para **tipos primitivos** mas se tornam mais **expressivos** para **tipos compostos**.

```
void swap(int a, int b) {
    int aux = a;
    a = b; b = aux;
}

int main() {
    int x = 3, y = 4;
    swap(x,y);
    cout << x << endl;
    cout << y << endl;
    return 0;
}
```

Output:
3
4

Referências em C++

Para **acessar** os **objetos passados**, podemos passar **ponteiros** para eles.

Passagem por ponteiro

A **função** recebe **endereços**, podendo então **manipular livremente** os dados localizados neles. Mais **liberdade e eficiência** mas menos **legibilidade e segurança** que a **passagem por valor**.

```
void swap(int *a, int *b) {  
    int aux = *a;  
    *a = *b; *b = aux;  
}  
  
int main() {  
    int x = 3, y = 4;  
    swap(&x, &y);  
    cout << x << endl;  
    cout << y << endl;  
    return 0;  
}
```

Output:
4
3

Referências em C++

C++ permite também a **passagem de parâmetros** por **referência** com o **operador &**.
(é diferente do operador & de endereços!)

Passagem por referência

A **função** recebe **alias**. Operações feitas com eles utilizarão os **objetos originais, implicitamente**. É **similar** à abordagem com **ponteiros**, mais prática mas menos flexível.

```
void swap(int &a, int &b) {  
    int aux = a;  
    a = b; b = aux;  
}  
  
int main() {  
    int x = 3, y = 4;  
    swap(x, y);  
    cout << x << endl;  
    cout << y << endl;  
    return 0;  
}
```

Output:
4
3

Referências em C++

Usamos **referências constantes** quando queremos acessar os dados originais (sem copiá-los) **sem permitir a alteração** dos mesmos:

↓

```
int soma(const int &a, const int &b) {  
    return a+b;  
}
```

```
int main() {  
    int x = 3, y = 4;  
    x = soma(x, y);  
    cout << x << endl;  
    cout << y << endl;  
    return 0;  
}
```

Output:
7
4

Orientado a objetos



Esses objetos...

```
#include <iostream>  
using namespace std;
```

```
class Aluno  
{  
    string nome;  
    int idade;  
    double nota;  
};
```

O que é uma classe?

Uma **classe** define um objeto. Diz quais métodos e atributos ele terá. Mas...

UMA CLASSE NÃO É UM OBJETO.

Um objeto só existe a partir do momento que é criado(ou instanciado, o que significa a mesma coisa, embora seja um termo mais utilizado).

Cada objeto é instância de uma classe.



O que é uma classe?



Um objeto pode ter **métodos** e **atributos**.

Objeto do tipo **Aluno**:

Métodos: estudar();
fazerProva(); etc.

Atributos:
nome, idade, nota.

Herança

Uma classe pode ser filha (ou mãe) de outra classe.

Se todo aluno é uma pessoa, porque não fazer a classe aluno herdar de pessoa?

```
class Pessoa {  
    string nome;  
    int idade;  
}  
  
class Aluno: public Pessoa  
{  
    double nota;  
}
```

C++ suporta herança múltipla.



Polimorfismo

É a capacidade de um método de uma classe se comportar diferente do método de mesmo nome da classe mãe.

```
class Pessoa {  
    //atributos  
    void dormir() { /* dorme por 8 horas */ }  
}
```

```
class AlunoDeHumanas: public Pessoa, public Aluno {  
    //atributos  
    void dormir() { /* dorme por 6 horas */ }  
}
```

```
class AlunoDeExatas: public Pessoa, public Aluno {  
    //atributos  
    void dormir() { /* dorme por 3 horas */ }  
}
```



class == struct

Exceto pelo fato que, em **struct**, os membros são públicos por padrão

Por exemplo...

```
class Aluno  
{  
    int membroPrivado;  
  
public:  
    string nome;  
    int idade;  
    float nota;  
    void estudar()  
    {  
        /* ... */  
    }  
};
```

```
struct Aluno  
{  
    string nome;  
    int idade;  
    float nota;  
    void estudar()  
    {  
        /* ... */  
    }  
  
private:  
    int membroPrivado;  
};
```

Como usar uma classe

Padrão:

```
Aluno a;  
a.estudar();
```

Com parâmetros:

```
Aluno a("Pedro");  
a.estudar();
```

Ponteiro:

```
Aluno *a = new Aluno("Ana");  
a->estudar();
```

Atribuindo:

```
Aluno a;  
a = Aluno();  
//a.assistirAula(); TODO: compilation error
```

OBS:

A definição do **construtor** da classe pode ser customizada ao criar um **método público** com o mesmo nome da classe!

Templates

são importantes!

São o T da STL

(Standard Template Library)

Mas... o que são templates?

São modelos.

Com elas, funções e classes podem operar com tipos genéricos!

Não entendeu? Vamos lá...

Soma de inteiros

```
int soma(int a, int b)
{
    return a + b;
}
```

Soma genérica

```
template <typename T>
T soma(T a, T b)
{
    return a + b;
}
```

Então...

```
int a = soma(2, 2);
```

```
int b = soma<int>(2, 2);
```

```
long c = soma<long>(2, 2);
```

```
double d = soma<double>(2.0, 2.0);
```

Templates e Classes

```
class Array
{
    int A[10000];
};
```

→

```
template <typename T>
class Array
{
    T A[10000];
};
```

Instanciando...

```
int main()
{
    Array<int> V;
}
```

Templates e Classes

```
class Array
{
    int A[10000];
};
```

→

```
template <typename T, int N>
class Array
{
    T A[N];
};
```

Instanciando...

```
int main()
{
    Array<int, 10000> V;
}
```

Templates e Classes

```
template <typename T, int N = 10000>
class Array
{
    T A[N];
};
```

Valor default

Instanciando...

```
int main()
{
    Array<int> V;           //Tem 10000 elementos
    Array<int, 5000> W;    //Tem 5000 elementos
}
```

Input / Output

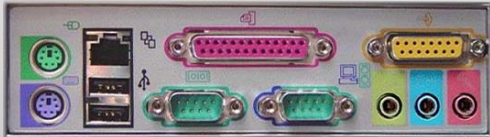
Manipulando entrada e saída com C++



Bibliotecas

iostream fstream

sstream iomanip



Lendo e exibindo valores? (cin, cout)

```
#include <iostream>
using namespace std;

int main()
{
    int a, b;
    while(cin >> a >> b)
    {
        cout << a+b << endl;
    }
}
```

Streams em C++

Fluxo >> Dados (>> Dados ...)

Extrai informação do fluxo, delimitada por espaços ou quebras de linha

e.g.: **cin** (dispositivo de input)

cin >> a >> b >> c;

Fluxo << Dados (<< Dados ...)

Inserir informação no fluxo, obtida pelos dados

e.g.: **cout** (dispositivo de output)

cout << a << endl;

Streams em C++

cin: istream

cout: ostream

cerr: ostream

Além deles:

<fstream>: file streams

<sstream>: string streams

<iomanip>: stream manipulations

Streams em C++

Aceitam dados de **qualquer tipo** que suporte os operadores << e >>!

```
#include <iostream>
#include <sstream>
```

```
using namespace std;
```

```
int main()
{
    stringstream ss; ss.str(""); ss.clear();
    double a; int b;
    ss << "3.1415" << endl << "3.1415";
    ss >> a >> b;
    cout << a << endl << b << endl;
    return 0;
}
```

```
3.1415
3
```

Usando <sstream>

```
#include <iostream>
#include <sstream>
```

```
using namespace std;
```

```
int main()
{
    string linha;
    int n;
    stringstream ss;

    getline(cin, linha);
    ss.str(linha);
    while(ss >> n) { cout << n << endl; }
    return 0;
}
```

Converte string
em sequência de
inteiros!



Usando <sstream>

str(), str(stringnova);

Retorna ou altera a sequência de caracteres armazenada na stream. Use **str("")** para limpar o conteúdo!

clear();

Limpa as flags de erro da stream.

operador >>

Extraí caracteres sequencialmente, atribuindo ao operando à direita.

operador <<

Inserir caracteres sequencialmente, gerados a partir do operando à direita.

Usando <iomanip>

```
#include <iostream>
#include <iomanip>
using namespace std;
```

```
int main()
{
    cout << setw(8) << setfill('0') << 42 << endl;
    cout << setw(8) << left << 42 << endl;
    cout << fixed << setprecision(3) << 42.32 << endl;
    return 0;
}
```

```
00000042
42000000
42.320
```

Saída no console:

String

Sequência ordenada de caracteres



o Construtor string()

string();

Construtor default. Cria uma string vazia.

string(str);

Cria uma string como cópia de str.
(aceita tipo string ou tipo char*)

string(str, pos, n);

Cria uma string como substring de str, contando n caracteres a partir da posição pos (inclusive).
(str é do tipo string)

o Construtor string()

`string(S, n);`

Cria uma `string` com os `n` primeiros elementos do array de caracteres `S`.

`string(n, c);`

Cria uma `string` com o caractere `c` repetido `n` vezes.

`string(begin, end);`

Cria uma `string` com o conteúdo no intervalo definido pelos iterators `begin` (inclusive) e `end` (exclusive), mantendo a ordem.

Strings

```
#include <iostream>
using namespace std;
```

```
int main()
{
    string s = "Hello World, World";
    cout << s.find("World");
    return 0;
}
```

6

Strings

```
#include <iostream>
using namespace std;
```

```
int main()
{
    string s = "Hello World, World";
    int pos = s.find("World");
    bool contemWorld = (pos != string::npos);
    cout << (contemWorld?("Sim")?("Nao"));

    return 0;
}
```

Sim

Strings

```
#include <iostream>
using namespace std;
```

```
int main()
{
    string s = "Hello World, World";
    int pos = s.find("World");
    int pos2 = s.find("World", pos+1);
    cout << pos2;

    return 0;
}
```

13

Strings

```
#include <iostream>
using namespace std;
```

```
int main()
{
    string s = "Hello World, World";
    int pos = s.find_first_of("aeiou");
    cout << pos << endl;
    int pos2 = s.find_first_of("aeiou", pos+1);
    cout << pos2 << endl;

    return 0;
}
```

1
4

Strings

```
#include <iostream>
using namespace std;
```

```
int main()
{
    string s = "Hello World, World";
    cout << s.substr(1, 4);

    return 0;
}
```

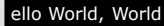
ello

Strings

```
#include <iostream>
using namespace std;
```

```
int main()
{
    string s = "Hello World, World";
    cout << s.substr(1);

    return 0;
}
```



Strings

```
#include <iostream>
using namespace std;
```

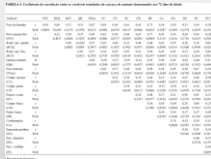
```
int main()
{
    string s = "Hello World, World";
    cout << s.length() << endl;
    cout << s.size() << endl;

    return 0;
}
```



Vector

Arrays de tamanho **variável**



Nome	Descrição	Exemplo
vector	Contêiner de dados de tamanho variável	vector<int> v;
vector::begin	Retorna um iterador para o primeiro elemento	vector<int> v; v.begin();
vector::end	Retorna um iterador para o elemento após o último	vector<int> v; v.end();
vector::size	Retorna o número de elementos	vector<int> v; v.size();
vector::push_back	Adiciona um elemento no final	vector<int> v; v.push_back(1);
vector::pop_back	Remove o elemento do final	vector<int> v; v.pop_back();
vector::clear	Remove todos os elementos	vector<int> v; v.clear();
vector::empty	Verifica se o vetor está vazio	vector<int> v; v.empty();
vector::operator[]	Retorna referência ao elemento no índice	vector<int> v; v[0];
vector::at	Retorna referência ao elemento no índice, com verificação de erro	vector<int> v; v.at(0);
vector::front	Retorna referência ao primeiro elemento	vector<int> v; v.front();
vector::back	Retorna referência ao último elemento	vector<int> v; v.back();
vector::front	Retorna const referência ao primeiro elemento	vector<int> v; v.front();
vector::back	Retorna const referência ao último elemento	vector<int> v; v.back();
vector::data	Retorna o endereço de memória dos elementos	vector<int> v; v.data();
vector::operator[]	Retorna const referência ao elemento no índice	vector<int> v; v[0];
vector::at	Retorna const referência ao elemento no índice, com verificação de erro	vector<int> v; v.at(0);
vector::front	Retorna const referência ao primeiro elemento	vector<int> v; v.front();
vector::back	Retorna const referência ao último elemento	vector<int> v; v.back();
vector::front	Retorna const referência ao primeiro elemento	vector<int> v; v.front();
vector::back	Retorna const referência ao último elemento	vector<int> v; v.back();
vector::data	Retorna o endereço de memória dos elementos	vector<int> v; v.data();

Biblioteca
vector

Como usá-la:

```
#include <vector>
```

Construtores:

```
vector<tipo> nome;
```

```
vector<tipo> nome(parâmetro);
```

Exemplo:

```
vector<int> v;
```

```
vector<int> v1(v);
```

```
/* inicia v1 com copia de v */
```

Iterators

Iterator é um objeto que **permite** ao **programador** percorrer **todos** os **elementos** de uma **coleção**



Uso de iterator em **vector**

`it = v.begin(); it = v.end();`
Retorna um iterator para o primeiro elemento ou para o final do vetor.

`it = v.rbegin(); it = v.rend();`
Retorna um reverse iterator para o primeiro elemento reverso ou para o final reverso do vetor.
(operam em ordem **inversa**)

Funções de acesso ao **vector**

`front(), back();`
Retorna o primeiro/último elemento do vetor.

`push_back(elemento);`
Adiciona um elemento no **final** do vetor.

`pop_back();`
Remove o elemento do **final** do vetor.

Exemplo

```
#include <iostream>
#include <vector>
using namespace std;

int main()
{
    vector<int> v;

    for (int i=0; i<= 10; ++i) v.push_back(i);
    cout << "Primeiro elemento..."<< v.front() << endl;
    cout << "Ultimo elemento..."<< v.back() << endl;

    return 0;
}
```

Primeiro elemento...0
Ultimo elemento...10

Exemplo

```
#include <iostream>
#include <vector>
using namespace std;

int main()
{
    vector<int> v;
    for (int i=1; i<= 10; ++i) v.push_back(i);

    vector<int>::iterator it = v.begin();
    while(it != v.end()) cout << *(it++) << " "; cout << endl;

    vector<int>::reverse_iterator rit = v.rbegin();
    while(rit != v.rend()) cout << *(rit++) << " "; cout << endl;

    return 0;
}
```

1 2 3 4 5 6 7 8 9 10
10 9 8 7 6 5 4 3 2 1

Funções de informação sobre o estado do **vetor**

`empty();`
Retorna true se o vetor estiver **vazio**, caso contrário **retorna false**.

`size();`
Retorna o **tamanho** do vetor em uma **unsigned int**.

Função erase

```
#include <iostream>
#include <vector>
using namespace std;

int main()
{
    int i;
    vector<int> v;
    for(i = 1; i<=10; i++) { v.push_back(i); }

    v.erase(v.begin()+4);
    for(i = 0; i < v.size(); i++) cout << v[i] << " "; cout << endl;

    v.erase( v.begin() , v.begin()+3);
    for(i = 0; i < v.size(); i++) cout << v[i] << " "; cout << endl;

    return 0;
}
```

1 2 3 4 6 7 8 9 10
4 6 7 8 9 10

Função resize

```
#include <iostream>
#include <vector>
using namespace std;

int main()
{
    int i;
    vector<int> v;

    for(i = 1; i <= 10; i++) { v.push_back(i); }

    v.resize(4);
    for(i = 0; i < v.size(); i++) cout << v[i] << " "; cout << endl;

    v.resize(7,10);
    for(i = 0; i < v.size(); i++) cout << v[i] << " "; cout << endl;

    return 0;
}
```

```
1 2 3 4
1 2 3 4 10 10 10
```

Sort – versão padrão

```
#include <vector>
#include <algorithm>
using namespace std;

int main()
{
    vector<int> v;
    v.push_back(3);
    v.push_back(2);
    v.push_back(10);

    sort(v.begin(), v.end());
    return 0;
}
```

Sort – versão inversa

```
#include <vector>
#include <algorithm>
#include <functional>
using namespace std;

int main()
{
    vector<int> v;
    v.push_back(3);
    v.push_back(2);
    v.push_back(10);

    sort(v.begin(), v.end(), greater<int>());
    return 0;
}
```

Sort – versão customizada

```
#include <vector>
#include <algorithm>
using namespace std;

bool menor(int a, int b) { return a < b; }

int main()
{
    vector<int> v;
    v.push_back(3);
    v.push_back(2);
    v.push_back(10);

    sort(v.begin(), v.end(), menor);
    return 0;
}
```

Busca(s) Binária(s)

- Demonstraremos as **três** funções
 - lower_bound
 - upper_bound
 - binary_search
- Recebem **três ou quatro** parâmetros
 - iterator begin
 - iterator end
 - valor
 - comparador (opcional)
- **Requerem** um container **ordenado**



Busca(s) Binária(s) lower_bound

Retorna um iterator para o **primeiro** elemento **não-menor** que o **valor** especificado.

Sendo:

v = {10, 10, 12, 15, 20, 20, 20, 25, 50}

int n = *lower_bound(v.begin(), v.end(), 20);

O valor em **negrito** será **retornado**, o **ponteiro** para o primeiro valor 20 do vetor.

Busca(s) Binária(s) upper_bound

Retorna um **iterator** para o **primeiro elemento maior** que o **valor** especificado.

Sendo:

`v = {10, 10, 12, 15, 20, 20, 20, 25, 50}`

```
int n = *upper_bound(v.begin(), v.end(), 20);
```

Será **retornado** o **ponteiro** para o valor em **negrito**, justamente o primeiro valor maior que 20.

Busca(s) Binária(s) - *_bound

Quando o valor não puder ser encontrado, o mesmo valor será **retornado** pelo `upper_bound` e pelo `lower_bound`.

Sendo:

`v = {10, 10, 12, 15, 20, 20, 20, 25, 50}`

```
int n = *lower_bound(v.begin(), v.end(), 21);
```

```
int n = *upper_bound(v.begin(), v.end(), 21);
```

Será **retornado** o **ponteiro** para o valor em **negrito**, justamente o valor acima do valor desejado.

Busca(s) Binária(s) Binary Search

O `binary_search` simplesmente diz se o valor existe ou não, **retornando** uma **bool**.

Sendo:

`v = {10, 10, 12, 15, 20, 20, 20, 25, 50}`

```
bool a = binary_search(v.begin(), v.end(), 21);
```

```
bool b = binary_search(v.begin(), v.end(), 20);
```

`a = false`

`b = true`

Outras utilidades da <algorithm>

```
for_each(comeco, fim, fn);
```

Aplica a função **fn** a todos os elementos em **[comeco, fim)**.

```
find(comeco, fim, val);
```

Retorna um **iterator** para a primeira ocorrência de **val** em **[comeco, fim)**.

```
count(comeco, fim, val);
```

Retorna a quantidade de ocorrências de **val** em **[comeco, fim)**.

Outras utilidades da <algorithm>

```
min_element(comeco, fim, comp);
```

Retorna **iterator** para o **menor** elemento em **[comeco, fim)** de acordo com a função de comparação **comp**. Especificar **comp** é opcional, por default é o **operador <**.

```
max_element(comeco, fim, comp);
```

Análogo ao `min_element`, mas retorna o **iterator** para o **maior** elemento.

Várias outras facilidades que se aplicam aos containers da STL (busca e substituição, particionamento, permutações, preenchimento de intervalos e mais) estão documentadas em

<http://www.cplusplus.com/reference/algorithm/>

C++

Pilhas



Pilha (Stack)

É um tipo de coleção onde o último elemento a entrar é o primeiro a sair, LIFO (Last in, First out)



Biblioteca
stack

Como usá-la:

```
#include <stack>
```



Construtores:

```
stack<tipo> nome;
```

```
stack<tipo> nome(parâmetro);
```

Exemplo:

```
stack<int> p;
```

```
stack<int> p1(p);
```

```
/* inicia p1 com copia de p */
```

Funções de acesso a pilha

```
top();
```

Retorna um iterator do elemento do topo da pilha.

```
push(elemento);
```

Adiciona um elemento ao topo da pilha.

```
pop();
```

Remove o elemento do topo da pilha.

Funções de informação sobre o estado da pilha

```
empty();
```

Retorna true se a pilha estiver vazia, caso contrário retorna false.

```
size();
```

Retorna o tamanho da pilha em uma unsigned int.

Exemplo

```
#include <iostream>
#include <stack>
using namespace std;
```

Tirando os elementos... 4 3 2 1 0

```
int main()
{
    stack<int> pilha;
    for (int i=0; i<5; ++i) pilha.push(i);
    cout << "Tirando os elementos...";
    while (!pilha.empty())
    {
        cout << " " << pilha.top();
        pilha.pop();
    }
    cout << endl;
    return 0;
}
```

Pilha customizada

Podemos escolher qual contêiner usar para armazenar os elementos (padrão: `deque`)

Ex: `stack<int, vector<int>> pilha;`

Podemos iniciar a pilha com uma cópia de outro container

Mesmo template da declaração

Padrão também é `deque`!

Ex: `stack<int, vector<int>> pilha(meuVector);`

Pilha contém os elementos de `meuVector`

Problemas no Valladolid

442 - Matrix Chain Multiplication

514 - Rails

673 - Parentheses Balance

10152 - ShellSort

C++

Filas



Fila (Queue)

É um tipo de coleção onde o primeiro elemento a entrar é o primeiro a sair, FIFO (First in, First out)



Biblioteca
queue

Como usá-la:

#include <queue>



Construtores:

`queue<tipo> nome;`

`queue<tipo> nome(parâmetro);`

Exemplo:

`queue<int> f;`

`queue<int> f1(f);`

/ inicia f1 com copia de f */*

Funções

`back();`

Retorna um iterator do
último elemento da fila.

`push(elemento);`

Adiciona um elemento ao fim da fila.

`pop();`

Remove o primeiro elemento da fila.

Funções

`front();`

Retorna um iterator do
primeiro elemento da fila.

`size();`

Retorna o tamanho da fila em uma `unsigned int`.

`empty();`

Retorna `true` se a fila estiver vazia,
caso contrário retorna `false`.

Exemplo

```
#include <iostream>
#include <queue>
using namespace std;
```

```
int main()
```

```
{
    int i;
    queue<int> f1;
    for(i = 1; i <= 10; i++) { f1.push(i); }
    cout << "Primeiro elemento: " << f1.front() << endl;
    cout << "Ultimo elemento: " << f1.back() << endl;
    f1.pop(); cout << "Removendo Primeiro elemento" << endl;
    cout << "Primeiro elemento: " << f1.front() << endl;
    cout << "Tamanho de fila elemento: " << f1.size() << endl;
    return 0;
}
```

```
Primeiro elemento: 1
Ultimo elemento: 10
Removendo primeiro elemento
Primeiro elemento: 2
Tamanho da fila: 9
```

Deque

Deque (Double-ended queue) é um
tipo de coleção onde os elementos
podem ser inseridos e
removidos tanto
no início quanto no fim.



Como usá-la:

```
#include <deque>
```

Biblioteca

```
deque
```

Os principais subtipos

Input-restricted: deleção pode ser feita
de ambos os lados, mas a inserção somente
de um.

Output-restricted: inserção pode ser feita
de ambos os lados, mas a deleção somente
de um.

A maioria das filas e pilhas podem ser
consideradas especializações das deque.

Propriedades

- Elementos individuais podem ser acessados pelo índice.
 - Iteração dos elementos pode ser feita em qualquer ordem.
- Elementos podem ser eficientemente adicionados ou removidos de qualquer ponta.
- Deques não garantem armazenamento contínuo de seus elementos, logo pode ocorrer erro em acessos com cálculos de ponteiros.
- São similares aos vetores em sua interface (mas bem diferentes na implementação interna).

Construtores:

deque<tipo> nome;

deque<tipo> nome(parâmetro);

Exemplos:

deque<int> d;

deque<int> d1(d);

deque<int> d2(10);

deque<double> d3(10, 0.7);

Funções de acesso

front();

Acessa o primeiro elemento da deque.

back();

Acessa o último elemento da deque.

[índice];

Acessa o elemento da posição passada.

at(índice);

Acessa o elemento da posição passada, mas caso o índice não exista na deque é jogada uma exceção de out_of_range.

```
#include <iostream>
#include <stdexcept>
#include <deque>
using namespace std;

int main()
{
    int array[] = {0,1,2,3,4,5};
    deque<int> dq(array, array+5);

    cout << "front " << dq.front() << ", back " << dq.back() << endl;
    for(int i = 0; i <= 5; ++i) cout << "dq[" << i << "] = " << dq[i] << endl;

    try { cout << "dq.at(5) = " << dq.at(5) << endl; }
    catch(out_of_range)
    { cout << "*** dq.at(5) out_of_range exception" << endl; }

    dq.at(0) = 23;
    dq.back() = 32;
    cout << "front " << dq.front() << ", back " << dq.back() << endl;
    return 0;
}
```

front 0, back 4
dq[0] = 0
dq[1] = 1
dq[2] = 2
dq[3] = 3
dq[4] = 4
dq[5] = 0
** dq.at(5) out_of_range exception
front 23, back 32

Funções de informações

empty();

Retorna true se a deque estiver vazia, caso contrário retorna false.

size();

Retorna o tamanho da deque em uma unsigned int.

max_size();

Retorna o tamanho máximo permitido para a deque usado.

Exemplo

```
#include <iostream>
#include <deque>
using namespace std;

int main()
{
    deque<double> dq(5, 7.5);

    cout << "Deque tamanho: " << dq.size() << endl;
    cout << "Deque vazio? " << ((dq.empty())?("Sim")):("Não")) << endl;
    cout << "Tamanho maximo: " << dq.max_size() << endl;

    return 0;
}
```

Deque tamanho: 5
Deque vazio? Não
Tamanho maximo: 4294967295

Funções push() e pop()

`push_back(elemento);`

Adiciona um elemento no **fim** da deque.

`push_front(elemento);`

Adiciona um elemento no **início** da deque.

`pop_back();`

Remove o **último** elemento da deque.

`pop_front();`

Remove o **primeiro** elemento da deque.

Funções dos iterators

`begin();`

Retorna um iterator para o **primeiro** elemento.

`end();`

Retorna um iterator para o **final** da deque.

`rbegin();`

Retorna um iterator para o **primeiro** elemento reverso da deque, ou seja, o **último**.

`rend();`

Retorna um iterator para o **final** reverso da deque, ou seja, para o **começo** da deque.

Função erase();

`It = erase(iterator location);`

Deleta o elemento em **location** e **retorna** o iterator que **aponta** para o elemento que ocupou o **lugar** do item **excluído**.

`It = erase(iterator start, iterator end);`

Deleta os elementos entre **start** e **end**, e **retorna** o iterator que **aponta** para o elemento que ocupou o **lugar** do item **excluído**.

C++

Lista



Lista (List)



List é uma sequência de elementos guardados numa **lista** duplamente encadeada.

Comparado com **vectors**, são mais rápidas na inserção e deleção de elementos, porém **lentas** no acesso aleatório.

Biblioteca
list

Como usá-la:

#include <list>



Funções dos iterators

`begin();`

Retorna um iterator para o primeiro elemento.

`end();`

Retorna um iterator para o final da lista.

`rbegin();`

Retorna um iterator para o primeiro elemento reverso da lista, ou seja, o último.

`rend();`

Retorna um iterator para o final reverso da lista, ou seja, para o começo da lista.

Funções de informações

`empty();`

Retorna true se a lista estiver vazia, caso contrário retorna false.

`size();`

Retorna o tamanho da lista em uma unsigned int.

`max_size();`

Retorna o tamanho máximo permitido para a lista usada.

`resize(Novo Tamanho);`

Altera o tamanho da lista.

Exemplo(max_size):

```
#include <list>
#include <iostream>
using namespace std;
```

```
int main()
{
    list<int> lista;
    cout << "Tamanho permitido pelo sistema: ";
    cout << lista.max_size() << endl;
    return 0;
}
```

Tamanho permitido pelo sistema: 4294967295

Exemplo(resize):

```
#include <list>
#include <iostream>
using namespace std;
```

```
int main()
{
    long long tamanho;
    list<int> lista;
    cout << "Digite o tamanho: ";
    cin >> tamanho;
    lista.resize(tamanho, 8);
    lista.resize(tamanho*2);
    list<int>::iterator iter = lista.begin();
    for(; iter != lista.end(); ++iter) cout << *iter << " ";
    cout << endl;
    lista.resize(tamanho);
    iter = lista.begin();
    for(; iter != lista.end(); ++iter) cout << *iter << " ";
    return 0;
}
```

Digite o tamanho: 4
8 8 8 8 0 0 0 0
8 8 8 8

Funções de acesso

`front();`

Acessa o primeiro elemento da lista.

`back();`

Acessa o último elemento da lista.

Funções push() e pop()

`push_back(elemento);`

Adiciona um elemento no fim da lista.

`push_front(elemento);`

Adiciona um elemento no início da lista.

`pop_back();`

Remove o último elemento da lista.

`pop_front();`

Remove o primeiro elemento da lista.

Funções de manipulação

`assign(elemento);`
Atribui novo conteúdo a lista.

`erase(elemento);`
Deleta o elemento da lista.

`insert(elemento);`
Insere o elemento a lista.

Função `assign();`

`assign(quantidade, elemento);`
Atribui à lista uma nova lista formada apenas por elemento na quantidade determinada.

`assign(inicio, final);`
Atribui à lista uma cópia do segmento formado pelos elementos de inicio até final.

Exemplo(assign) 1:

```
#include <list>
#include <iostream>
using namespace std;
```

42 42 42 42 42 42 42

```
int main()
{
    list<int> l;
    l.assign(7, 42);
    list<int>::iterator it = l.begin();
    for(; it != l.end(); ++it) cout << *it << " ";
    cout << endl;
    return 0;
}
```

Exemplo(assign) 2:

```
#include <list>
#include <iostream>
using namespace std;
```

0 1 2 3 4 5 6 7 8 9

```
int main()
{
    list<int> l1, l2;
    for(int i = 0; i < 10; ++i) l1.push_back(i);
    l2.assign(7, 42);
    l2.assign(l1.begin(), l1.end());
    list<int>::iterator it = l2.begin();
    for(; it != l2.end(); ++it) cout << *it << " ";
    cout << endl;
    return 0;
}
```

Função `insert();`

`It = insert(posição, elemento);`
Insere o elemento antes da posição e retorna o iterator que aponta para o elemento inserido.

`insert(posição, quantidade, elemento);`
Insere o elemento na quantidade determinada antes da posição determinada.

`insert(posição, inicio, fim);`
Insere o segmento formado pelos elementos de inicio até final, antes da posição determinada.

Exemplo(insert) 1:

```
#include <list>
#include <iostream>
using namespace std;
```

```
int main()
{
    list<char> lc;
    for(int i = 0; i < 10; ++i) lc.push_back(i+65);
    list<char>::iterator it = lc.begin();
    it = lc.insert(it, 'T');
    for(; it != lc.end(); ++it)
        cout << *it << ", ";
    cout << endl;
    return 0;
}
```

T, A, B, C, D, E, F, G, H, I, J,

`#include <list>` **Exemplo(insert) 2:**
`#include <iostream>`
`using namespace std;`

```
int main()
{
    list<char> lc;
    for(int i = 0; i < 10; ++i) lc.push_back(i+65);
    list<char>::iterator it = lc.begin();
    lc.insert(it, 4, 'K');
    for(it = lc.begin(); it != lc.end(); ++it)
        cout << *it << " ";
    cout << endl;
    return 0;
}
```

K, K, K, K, K, A, B, C, D, E, F, G, H, I, J,

`#include <list>` **Exemplo(insert) 3:**
`#include <iostream>`
`using namespace std;`

```
int main()
{
    list<char> l1, l2;
    for(int i = 0; i < 10; ++i) l1.push_back(i+65);
    for(int i = 11; i < 16; ++i) l2.push_back(i+65);
    list<char>::iterator it = l1.begin();
    l1.insert(it, l2.begin(), l2.end());
    for(it = l1.begin(); it != l1.end(); ++it)
        cout << *it << " ";
    cout << endl;
    return 0;
}
```

L, M, N, O, P, A, B, C, D, E, F, G, H, I, J,

Função erase();

`It = erase(posição);`
Deleta o elemento na **posição** determinada e **retorna** o **iterator** que aponta para o elemento que ocupou o **lugar** do item **excluído**.

`It = erase(início, fim);`
Deleta os elementos entre **início** e **fim**, e **retorna** o **iterator** que aponta para o elemento que ocupou o **lugar** dos **itens** **excluídos**.

`#include <list>` **Exemplo(erase) 1:**
`#include <iostream>`
`using namespace std;`

```
int main()
{
    list<char> lc;
    for( int i = 0; i < 10; i++ )
        lc.push_back( i + 65 );
    int size = lc.size();
    list<char>::iterator stit, it;
    for( int i=0; i < size; i++ )
    {
        stit = lc.begin();
        it = lc.erase(stit);
        for( ; it != lc.end(); it++ ) cout << *it << " ";
        cout << endl;
    }
    return 0;
}
```

A B C D E F G H I J
 B C D E F G H I J
 C D E F G H I J
 D E F G H I J
 E F G H I J
 F G H I J
 G H I J
 H I J
 I J
 J

`#include <list>` **Exemplo(erase) 2:**
`#include <iostream>`
`using namespace std;`

```
int main()
{
    list<char> lc;
    for( int i = 0; i < 10; ++i) lc.push_back(i+65);
    list<char>::iterator it = lc.begin();
    for( ; it != lc.end(); ++it) cout << *it << " ";
    cout << endl;
    it = lc.begin();
    lc.erase( ++it, lc.end() );
    for(it = lc.begin(); it != lc.end(); it++)
        cout << *it << " ";
    cout << endl;
    return 0;
}
```

A B C D E F G H I J
 A

Outras funções de manipulação

`clear();`
Deleta todos os elementos de uma **lista**.

`swap(outra lista);`
Troca o conteúdo da lista com o de outra lista.

`splice(elementos);`
Move elementos de uma lista para outra lista.

Exemplo(swap):

```
#include <list>
#include <iostream>
using namespace std;
```

```
int main()
{
    list<string> l1, l2;
    l1.push_back("Filho de l1!");
    l2.push_back("Filho de l2!");
    l1.swap(l2);
    cout << "O 1º elemento de l1 e: " << l1.front() << endl;
    cout << "O 1º elemento de l2 e: " << l2.front() << endl;

    return 0;
}
```

O 1º elemento de l1 e: Filho de l2!
O 1º elemento de l2 e: Filho de l1!

Função splice();

`splice(posição, outra lista);`

Move todos os elementos de outra lista para antes da posição na lista atual.

`splice(posição, outra lista, posição2);`

Move um elemento de outra lista (localizado em posição2) para antes da posição na lista atual.

`splice(posição, outra lista, inicio, fim);`

Move o segmento de outra lista formado pelos elementos de inicio até fim, para antes da posição determinada.

Exemplo(splice) 1:

```
#include <list>
#include <iostream>
using namespace std;
```

```
int main()
{
    list<int> l1, l2;
    list<int>::iterator it;
    for(int i = 1; i < 5; ++i) l1.push_back(i);
    for(int i = 1; i < 4; ++i) l2.push_back(i*10);
    it = l1.begin();
    l1.splice(++it, l2);
    cout << "Lista 1: ";
    for(it=l1.begin(); it != l1.end(); ++it) cout << *it << " ";
    cout << endl << "Lista 2: ";
    for(it = l2.begin(); it != l2.end(); ++it) cout << *it << " ";
    cout << endl;
    return 0;
}
```

Lista 1: 1 10 20 30 2 3 4
Lista 2:

Exemplo(splice) 2:

```
#include <list>
#include <iostream>
using namespace std;
```

```
int main()
{
    list<int> l1, l2;
    list<int>::iterator it;
    for(int i = 1; i < 5; ++i) l1.push_back(i);
    for(int i = 1; i < 4; ++i) l2.push_back(i*10);
    it = l1.begin();
    l2.splice(l2.begin(), l1, ++it);
    cout << "Lista 1: ";
    for(it=l1.begin(); it != l1.end(); ++it) cout << *it << " ";
    cout << endl << "Lista 2: ";
    for(it = l2.begin(); it != l2.end(); ++it) cout << *it << " ";
    cout << endl;
    return 0;
}
```

Lista 1: 1 3 4
Lista 2: 2 10 20 30

Exemplo(splice) 3:

```
#include <list>
#include <iostream>
using namespace std;
```

```
int main()
{
    list<int> l1, l2;
    list<int>::iterator it;
    for(int i = 1; i < 5; ++i) l1.push_back(i);
    for(int i = 1; i < 4; ++i) l2.push_back(i*10);
    it = l1.begin();
    l2.splice(l2.begin(), l1, ++it, l1.end());
    cout << "Lista 1: ";
    for(it=l1.begin(); it != l1.end(); ++it) cout << *it << " ";
    cout << endl << "Lista 2: ";
    for(it = l2.begin(); it != l2.end(); ++it) cout << *it << " ";
    cout << endl;
    return 0;
}
```

Lista 1: 1
Lista 2: 2 3 4 10 20 30

Mais funções de manipulação

`remove(valor);`

Remove elementos de uma lista que possuam um determinado valor.

`remove_if(condição);`

Remove todos elementos de uma lista que satisfaçam uma determinada condição.

`unique();`

Remove todos elementos duplicados de uma lista ordenada.

```
#include <list>
#include <iostream>
using namespace std;
```

Exemplo(remove):

```
list<char> l;
```

```
Lista: A B C D E F G H I J
Lista: A B C D F G H I J
```

```
void imprime() {
    list<char>::iterator it = l.begin(); cout << "Lista: ";
    for(; it != l.end(); ++it) { cout << *it << " "; }
    cout << endl;
}
```

```
int main()
{
    for(int i = 0 ; i < 10 ; ++i) l.push_back( i+65);
    imprime();
    l.remove('E');
    imprime();
    return 0;
}
```

```
#include <list>
#include <iostream>
using namespace std;
```

Exemplo(remove_if):

```
list<int> l;
class Compara {
public: bool operator()(int& a)
{ if(a<5) return true; else return false; };
```

```
Lista: 0 1 2 3 4 5 6 7 8 9
Lista: 5 6 7 8 9
```

```
void imprime() {
    list<int>::iterator it = l.begin(); cout << "Lista: ";
    for(; it != l.end(); ++it) { cout << *it << " "; }
    cout << endl; }
```

```
int main()
{
    for( int i = 0 ; i < 10 ; i++ ) l.push_back(i);
    imprime();
    l.remove_if(Compara());
    imprime();
    return 0;
}
```

```
#include <list>
#include <iostream>
using namespace std;
```

Exemplo(unique):

```
list<char> l;
```

```
Lista: A A B B C C D D E E
Lista: A B C D E
```

```
void imprime() {
    list<char>::iterator it = l.begin(); cout << " Lista: ";
    for(; it != l.end(); ++it) { cout << *it << " "; }
    cout << endl; }
```

```
int main()
{
    for(int i = 0 ; i < 5 ; ++i)
    { l.push_back(i+65); l.push_back(i+65); }
    imprime();
    l.sort();
    l.unique();
    imprime();
    return 0;
}
```

Mais funções de manipulação

merge(outra lista);
Faz o merge de uma lista ordenada.

sort();
Ordena todos elementos em uma lista.

reverse();
Coloca todos elementos em ordem inversa.

```
#include <list>
#include <iostream>
using namespace std;
```

Exemplo(merge):

```
void imprime(list<char> l, string nome) {
    list<char>::iterator it = l.begin(); cout << nome << " Lista: ";
    for(; it != l.end(); ++it) { cout << *it << " "; }
    cout << endl; }
```

```
int main()
{
    list<char> l1, l2;
    for(int i = 0 ; i < 10 ; ++i)
    { if(i%2==0) l1.push_back(i+65); else l2.push_back(i+65); }
    imprime(l1, "1ª");
    imprime(l2, "2ª");
    l2.merge(l1);
    imprime(l1, "1ª");
    imprime(l2, "2ª");
    return 0;
}
```

```
1ª Lista: A C E G I
2ª Lista: B D F H J
1ª Lista:
2ª Lista: A B C D E F G H I J
```

Problemas no Valladolid

10107 - What is the Median?
10420 - List of Conquests

C++

Lista de Prioridades



Lista de Prioridades (Priority Queues)

Priority Queues são **semelhantes** às Queues, mas os elementos **contidos** na fila são **ordenados** de **acordo** com um determinado **predicado**.



Biblioteca **queue**

Como usá-la:

#include <queue>

Funções

top();

Retorna o elemento de **maior** prioridade.

push(elemento);

Adiciona o elemento na priority queue.

pop();

Remove o elemento de **maior** prioridade da priority queue.

Funções

size();

Retorna o **número** de elementos da priority queue em uma **unsigned int**.

empty();

Retorna **true** se a priority queue estiver **vazia**, caso contrário **retorna false**.

Exemplo 1

```
#include <queue>
#include <iostream>
using namespace std;
```

```
int main()
{
    priority_queue<int> pq;
    pq.push(2);
    pq.push(5);
    pq.push(1);
    pq.push(3);
    pq.push(4);
    for(;;!pq.empty(); pq.pop()) cout<< pq.top() << endl;
    return 0;
}
```

5
4
3
2
1

Exemplo(greater)

```
#include <string>
#include <queue>
#include <iostream>
using namespace std;

int main()
{
    priority_queue<int, vector<int>, greater<int> > pq;
    pq.push(2);
    pq.push(5);
    pq.push(1);
    pq.push(3);
    pq.push(4);
    for(;!pq.empty(); pq.pop())
        cout<< pq.top() << endl;
    return 0;
}
```

1
2
3
4
5

Versão customizada

```
#include <queue>
#include <iostream>
#include <iomanip>
using namespace std;

struct Aluno
{
    float media;
    string nome;
};

class Compara
{
public:
    bool operator()(Aluno& a1, Aluno& a2)
    {
        if(a1.media < a2.media) return true;
        if(a1.media == a2.media && a1.nome > a2.nome) return true;
        return false;
    }
};
```

```
int main()
{
    priority_queue<Aluno, vector<Aluno>, Compara> pq;
    Aluno novo_aluno;
    for(int i = 0; i < 4; ++i)
    {
        cout<< "Nome: ";
        cin>> novo_aluno.nome;
        cout<< "Media: ";
        cin>> novo_aluno.media;
        pq.push(novo_aluno);
    }
    cout<< setw(12) << left << "Nome" << " ";
    cout << setw(5) << "Media" << endl;
    for(;!pq.empty(); pq.pop())
    {
        Aluno aluno_atual = pq.top();
        cout<< setw(12) << left << aluno_atual.nome << " ";
        cout<< setw(5) << aluno_atual.media << endl;
    }
    return 0;
}
```

Entrada e Saída:

Nome: Ronaldo
Media: 4.5
Nome: Kaka
Media: 8
Nome: Adriano
Media: 4.5
Nome: Robinho
Media: 8
Nome Media
Kaka 8
Robinho 8
Adriano 4.5
Ronaldo 4.5

C++

Map



Map

Map é um conjunto
associativo
ordenado que
mapeia objetos do
tipo Key (a chave)
em objetos do tipo
Data (o valor).



Biblioteca **map**

Como usá-la:
#include <map>

Construtor:

```
map<tipo da chave, tipo> nome;
```

Exemplo:

```
map<string, int> mapa;
```

Atribuição:

```
nome[tipo da chave] = tipo;
```

Exemplo:

```
mapa["Ganso"] = 10;
```

```
#include <map>
#include <iostream>
using namespace std;
```

Exemplo

```
int main()
{
    map<string, int> digitos;
    string s;
    digitos["zero"] = 0;
    digitos["um"] = 1;
    digitos["dois"] = 2;
    digitos["tres"] = 3;
    digitos["quatro"] = 4;

    while (cin >> s) { cout << digitos[s] << endl; }

    return 0;
}
```

```
zero
quatro
dois
0
4
2
```

Funções

begin();

Retorna um iterator para o primeiro elemento do map.

end();

Retorna um iterator para o final do map.

clear();

Remove todos os elementos do map em tempo linear.

Funções

empty();

Retorna true se o map estiver vazio, caso contrário retorna false.

erase(elemento(s) do map);

Remove o elemento, passado por parâmetro, do map.

count(chave);

Retorna um inteiro com a quantidade de ocorrência da chave no map.

```
#include <map>
#include <iostream>
using namespace std;
```

Exemplo

```
int main()
{
    map<string, int> m;
    m["zero"] = 0;
    m["um"] = 1;
    m["dois"] = 2;
    m["tres"] = 3;
    m["quatro"] = 4;
    map<string, int>::iterator iter = m.begin();
    while (iter != m.end())
    {
        cout << iter->first << " = " << iter->second << endl;
        ++iter;
    }
    return 0;
}
```

```
dois = 2
quatro = 4
tres = 3
um = 1
zero = 0
```



```
#include <map>
#include <iostream>
using namespace std;
```

```
int main()
{
    map<string, int> m;
    string s;
    while (cin >> s) ++m[s];
    map<string, int>::iterator iter = m.begin();
    while (iter != m.end())
    {
        cout << "palavra: " << iter->first;
        cout << ", contagem: " << iter->second;
        cout << endl;
        ++iter;
    }
    return 0;
}
```

Exemplo

```
UERJ curso c++ curso UERJ
```

```
palavra: UERJ, contagem: 2
palavra: c++, contagem: 1
palavra: curso, contagem: 2
```

Outras funções

find(chave);

Retorna um **iterator** para o **elemento desejado**,
se **não encontrado** retorna um **iterator**
para o **final** do **mapa**.

insert(elemento);

Inserir no **map** o **elemento** desejado.

max_size();

Retorna o **tamanho máximo**
permitido para o **map** usado.

Usando o find()

```
#include <map>
#include <iostream>
using namespace std;
```

```
int main()
{
    map<string, int> m;
    string s;
    while (cin >> s) ++m[s];
    map<string, int>::iterator iter = m.find("UERJ");
    if (iter != m.end())
    {
        cout << "Voce digitou '" << iter->first << "' ";
        cout << iter->second << " vez(es)" << endl;
    }
    return 0;
}
```

```
UERJ uerj UERJ, UERJ
```

```
Voce digitou 'UERJ' 2 vez(es)
```

Usando o insert()

```
#include <map>
#include <iostream>
using namespace std;
```

```
int main()
{
    map<string, int> m;
    m.insert(make_pair( s, 1 ));
    return 0;
}
```

make_pair()

Retorna um **objeto** que contém **2** itens, **first** e
second. **make_pair** é uma maneira **rápida** de
criar uma **instância** da **classe pair**.

OBS.:

OK! mas o que seria um **pair**?

Pair é um **template padrão**, no **C++**, para
objetos com **2 variáveis**(**first** e **second**) com
tipos que **podem** ser
diferentes.



Outras funções

rbegin();

Retorna um **iterator** para o **primeiro** elemento
reverso do **map**, ou seja, o **último**.

rend();

Retorna um **iterator** para o **final** reverso do **map**,
ou seja, para o **começo** do **map**.

size();

Retorna o **tamanho** do **map**, ou seja,
o **número** de **elementos** do **map**.

Funções de busca

`lower_bound(chave);`

Retorna um **iterator** para o **primeiro** elemento **não-menor** que o **valor** especificado.

`upper_bound(chave);`

Retorna um **iterator** para o **primeiro** elemento **maior** que o **valor** especificado.

`equal_range(chave)`

A **função** retorna um **pair**, onde o **pair::first** é o **iterator** retornado por `lower_bound(x)`, e o **pair::second** é o **iterator** retornado por `upper_bound(x)`.

Problemas no Valladolid

11503 - Virtual Friends

10878 - Decode the tape

C++

Set



Set

Set é um **container** **associativo** que **representa** um **conjunto** **matemático**



Biblioteca
set

Como usá-la:

#include <set>



Peculiaridades

Trabalha com **um** conjunto de **chaves**.

Num **set** as **chaves** **não** podem ser **repetidas**.

Aceita **acesso bidirecional**, mas **não** **randômico**.

Set **inclui**, **=**, **!=**, **<**, **>**, **<=**, **>=** e **swap**.

Internamente, os **elementos** no **set** são **classificados** do **menor** para o **maior**, **segundo** **critério** de **ordenação** definido na **construção** do **container**.

Template

`set<tipo, comparação, container> nome;`

Tipo: Tipo de chave, tipo dos elementos do container.

Comparação: Classe de comparação, classe que recebe 2 argumentos do mesmo tipo da chave e retorna um valor booleano. A expressão `comp(a,b)`, onde `comp` é um objeto dessa classe de comparação e `a` e `b` são elementos do container, deve retornar `true` se `a` deve ser colocado antes de `b`.

Container: Tipo do objeto usado para definir o modelo de alocação. Por padrão, é usada a classe `template allocator` para o tipo correspondente da chave, que define o modelo de alocação de memória mais simples e é independente do valor.

Construtores

```
#include <iostream>
#include <set>
using namespace std;

bool fncomp (int lhs, int rhs) {return lhs<rhs;}
struct classcomp {
    bool operator() (const int& lhs, const int& rhs) const
    {return lhs<rhs;}};

int main ()
{
    set<int> s1;
    int mi[] = {10,20,30,40,50};
    set<int> s2 (mi,mi+5);
    set<int> s3 (s2);
    set<int> s4 (s2.begin(), s2.end());
    set<int,classcomp> s5;
    set<int,bool(*) (int,int)> s6 (fncomp);
    return 0;
}
```

Funções

`begin();`

Retorna um iterator para o primeiro elemento do set.

`end();`

Retorna um iterator para o final do set.

`clear();`

Remove todos os elementos do set em tempo linear.

begin() e end()

```
#include <iostream>
#include <set>
using namespace std;
int main()
{
    int mi[] = {75,23,65,42,13};
    set<int> S(mi,mi+5);
    set<int>::iterator it = S.begin();
    cout << "S = { ";
    for( ; it != S.end(); ++it)
        cout << *it << " ";
    cout << "}" << endl;
    return 0;
}
```

S = { 13 23 42 65 75 }

clear()

```
#include <iostream>
#include <set>
using namespace std;
int main()
{
    int mi[] = {75,23,65,42,13};
    set<int> S(mi,mi+5);
    S.clear();
    set<int>::iterator it = S.begin();
    cout << "S = { ";
    for( ; it != S.end(); ++it)
        cout << *it << " ";
    cout << "}" << endl;
    return 0;
}
```

S = { }

Funções

`empty();`

Retorna `true` se o set estiver vazio, caso contrário retorna `false`.

`size();`

Retorna o tamanho do set em uma unsigned int.

`count(chave);`

Como o set não permite chaves duplicadas, Retorna 1 se possui a chave e 0 se não possui.

empty()

```
#include <iostream>
#include <set>
using namespace std;
```

```
S está vazio? Nao
S está vazio? Sim
```

```
int main()
{
    int mi[] = {75,23,65,42,13};
    set<int> S(mi,mi+5);
    cout << "S está vazio? ";
    cout << ((S.empty())?("Sim")):(("Nao")) << endl;
    S.clear();
    cout << "S está vazio? ";
    cout << ((S.empty())?("Sim")):(("Nao")) << endl;
    return 0;
}
```

size()

```
#include <iostream>
#include <set>
using namespace std;
```

```
S tem tamanho 5
S tem tamanho 0
```

```
int main()
{
    int mi[] = {75,23,65,42,13};
    set<int> S(mi,mi+5);
    cout << "S tem tamanho " << S.size() << endl;
    S.clear();
    cout << "S tem tamanho " << S.size() << endl;
    return 0;
}
```

count()

```
#include <iostream>
#include <set>
using namespace std;
```

```
S.count(23) 1
S.count(2) 0
S.count(54) 0
S.count(42) 1
```

```
int main()
{
    int mi[] = {75,23,65,42,13};
    set<int> S(mi,mi+5);
    cout << "S.count(23)" << S.count(23) << endl;
    cout << "S.count(2)" << S.count(2) << endl;
    cout << "S.count(54)" << S.count(54) << endl;
    cout << "S.count(42)" << S.count(42) << endl;
    return 0;
}
```

Funções

erase(elemento(s) do map);

Remove o(s) elemento(s), passado por parâmetro, do set.

find(chave);

Retorna um iterator para o elemento desejado, se não encontrado retorna um iterator para o final do set.

insert(elemento);

Insere no set o elemento desejado.

erase()

```
#include <iostream>
#include <set>
using namespace std;
```

```
void draw(set<int> s){
    set<int>::iterator it = s.begin(); cout << "S = { ";
    while(it != s.end()) { cout << *it << " "; ++it; }
    cout << "}" << endl;
}
```

```
int main()
{
    int mi[] = {75,23,65,42,13};
    set<int> S(mi,mi+5);
    draw(S); S.erase(23);
    draw(S); S.erase(75);
    draw(S); S.erase(23);
    draw(S); S.erase(65);
    draw(S); return 0;
}
```

```
S = { 13 23 42 65 75 }
S = { 13 42 65 75 }
S = { 13 42 65 }
S = { 13 42 65 }
S = { 13 42 }
```

erase() e find()

```
#include <iostream>
#include <set>
using namespace std;
```

```
void draw(set<int> s){
    set<int>::iterator it = s.begin(); cout << "S = { ";
    while(it != s.end()) { cout << *it << " "; ++it; }
    cout << "}" << endl;
}
```

```
int main()
{
    int mi[] = {75,23,65,42,13};
    set<int> S(mi,mi+5);
    draw(S); S.erase(S.begin(), S.find(42));
    draw(S); S.erase(S.find(65), S.end());
    draw(S); S.erase(S.find(42));
    draw(S);
    return 0;
}
```

```
S = { 13 23 42 65 75 }
S = { 13 42 65 75 }
S = { 42 }
S = { }
```

```
#include <iostream>
#include <set>
using namespace std;
void draw(set<int> s){
    set<int>::iterator it = s.begin(); cout << "S = { ";
    while(it != s.end()) { cout << *it << " "; ++it; }
    cout << "}" << endl;
}
int main()
{
    int mi[] = {1,7,9,8,2};
    set<int> S(mi,mi+5);
    draw(S); S.insert(5);
    draw(S); S.insert(6);
    draw(S); S.insert(3);
    draw(S); S.insert(7);
    draw(S); return 0;
}
```

insert()

```
S = { 1 2 7 8 9 }
S = { 1 2 5 7 8 9 }
S = { 1 2 5 6 7 8 9 }
S = { 1 2 3 5 6 7 8 9 }
S = { 1 2 3 5 6 7 8 9 }
```

Usando e abusando do insert()

pair = insert(elemento);

Insere no set o elemento desejado, e retorna um pair, sendo pair::first um iterator apontando para o elemento, e pair::second é definido como true se o elemento foi inserido e false caso ele já existia.

insert(primeiro, ultimo);

Insere no set a sequência de elementos [primeiro, ultimo) se possível.

```
#include <iostream>
#include <set>
using namespace std;
void draw(set<int> s){
    set<int>::iterator it = s.begin(); cout << "S = { ";
    while(it != s.end()) { cout << *it << " "; ++it; }
    cout << "}" << endl;
}
int main()
{
    int mi[] = {1,7,9,8,2};
    set<int> S(mi,mi+5);
    set<int>::iterator it; draw(S);
    pair<set<int>::iterator, bool> ret;
    ret = S.insert(2);
    if(!ret.second) cout << *ret.first << " jah existente!" << endl;
    ret = S.insert(5);
    if(ret.second) cout << *ret.first << " inserido!" << endl;
    draw(S); int mi2[] = {10,3,11,2}; S.insert(mi2,mi2+4);
    draw(S); return 0;
}
```

insert()

```
S = { 1 2 7 8 9 }
2 jah existente!
5 inserido!
S = { 1 2 5 7 8 9 }
S = { 1 2 3 5 7 8 9 10 11 }
```

set_union

(União de dois intervalos ordenados)

template<set1, set2, resultado>

set_union(first1, last1, first2, last2, resultado);

Constroi um intervalo ordenado no local apontado por resultado com a união dos dois intervalos [first1,last1) e [first2,last2) como conteúdo.

```
#include <iostream>
#include <set>
using namespace std;
void draw(set<int> s){
    set<int>::iterator it = s.begin(); cout << "{ ";
    while(it != s.end()) { cout << *it << " "; ++it; }
    cout << "}" << endl;
}
int main()
{
    int mi[] = {1,7,9,8,2};
    int mi2[] = {8,3,11,2,9};
    set<int> a(mi,mi+5), b(mi2,mi2+5), u;
    draw(a); draw(b);
    set_union(a.begin(), a.end(), b.begin(),
              b.end(), inserter(u, u.begin()));
    draw(u);
    return 0;
}
```

set_union

```
{ 1 2 7 8 9 }
{ 2 3 8 9 11 }
{ 1 2 3 7 8 9 11 }
```

set_intersection

(Interseção de dois intervalos ordenados)

template<set1, set2, resultado>

set_intersection(first1, last1, first2, last2, resultado);

Constroi um intervalo ordenado no local apontado por resultado com a interseção dos dois intervalos [first1,last1) e [first2,last2) como conteúdo.

```
#include <iostream>
#include <set>
using namespace std;
void draw(set<int> s){
    set<int>::iterator it = s.begin(); cout << "{ ";
    while(it != s.end()) { cout << *it << " "; ++it; }
    cout << "}" << endl;
}
int main()
{
    int mi[] = {1,7,9,8,2};
    int mi2[] = {8,3,11,2,9};
    set<int> a(mi,mi+5), b(mi2,mi2+5), i;
    draw(a); draw(b);
    set_intersection(a.begin(), a.end(), b.begin(),
        b.end(), inserter(i, i.begin()));
    draw(i);
    return 0;
}
```

set_intersection

```
{ 1 2 7 8 9 }
{ 2 3 8 9 11 }
{ 2 8 9 }
```

set_difference

(Diferença de dois intervalos ordenados)

```
template<set1, set2, resultado>
```

```
set_difference(first1, last1, first2, last2,
    resultado);
```

Constrói um **intervalo ordenado** no local apontado por **resultado** com a diferença dos **dois intervalos** **[first1,last1)** e **[first2,last2)** como conteúdo.

```
#include <iostream>
#include <set>
using namespace std;
void draw(set<int> s){
    set<int>::iterator it = s.begin(); cout << "{ ";
    while(it != s.end()) { cout << *it << " "; ++it; }
    cout << "}" << endl;
}
int main()
{
    int mi[] = {1,7,9,8,2};
    int mi2[] = {8,3,11,2,9};
    set<int> a(mi,mi+5), b(mi2,mi2+5), d;
    draw(a); draw(b);
    set_difference(a.begin(), a.end(), b.begin(),
        b.end(), inserter(d, d.begin()));
    draw(d);
    return 0;
}
```

set_difference

```
{ 1 2 7 8 9 }
{ 2 3 8 9 11 }
{ 1 7 }
```

Bitset

Bitset é um **container** **designado** para **armazenar** bits(**bool**, **0** ou **1**)

Construtor

```
bitset<tamanho> nome;
```

Funções

```
size();
```

Retorna o **tamanho** do **bitset**.

```
count();
```

Retorna **quantidade** de **1** no **bitset**.

Construtores

```
#include <iostream>
#include <bitset>
#include <string>
using namespace std;
```

```
int main()
{
    bitset<5> b1;
    bitset<5> b2(16ul);
    bitset<5> b3(string("01011"));
    cout << b1 << endl;
    cout << b2 << endl;
    cout << b3 << endl;
    return 0;
}
```

```
00000
10000
01011
```

Função set()

`set();`

Preenche **todas** as **posições** do **bitset** com **1**.

`set(posição);`

Preenche a **posição** indicada do **bitset** com **1**.

`set(posição, valor);`

Preenche a **posição** indicada do **bitset** com o **valor** indicado.

Função reset()

`reset();`

Preenche **todas** as **posições** do **bitset** com **0**.

`reset(posição);`

Preenche a **posição** indicada do **bitset** com **0**.

Função flip()

`flip();`

Inverte **todas** as **posições** do **bitset** colocando **1** onde era **0** e **0** onde era **1**.

`flip(posição);`

Inverte o bit da **posição** especificada no **bitset** colocando **1** se era **0** e **0** se era **1**.

Funções de acesso

`test(índice);`

Retorna o **bit** na **posição** indicada do **bitset**.

`[índice];`

Acessa o elemento da **posição** passada.

Operador []

```
#include <iostream>
#include <bitset>
#include <string>
using namespace std;
```

```
int main()
{
    bitset<5> bs(string("01011"));
    cout << boolalpha;
    for (size_t i=0; i<bs.size(); ++i)
        cout << bs[i] << endl;
    return 0;
}
```

```
true
true
false
true
false
```

test()

```
#include <iostream>
#include <bitset>
#include <string>
using namespace std;
```

```
int main()
{
    bitset<5> bs(string("01011"));
    cout << boolalpha;
    for (size_t i=0; i<bs.size(); ++i)
        cout << bs.test(i) << endl;
    return 0;
}
```

```
true
true
false
true
false
```


Outras funções

`to_ulong();`

Converte o bitset em número no formato ulong.

`any();`

Retorna uma bool relativa ao teste de conter algum bit do bitset preenchido com 1.

`none();`

Retorna uma bool relativa ao teste de não conter algum bit do bitset preenchido com 1.

Operadores

XOR - `1001 ^ 0011 = 1010;`

AND - `1010 & 0011 = 0010;`

OR - `0010 | 0011 = 0011;`

SHL - `0011 <= 2 = 1100;`

SHR - `1100 >= 1 = 0110;`

NOT - `~0011 = 1100;`

EQUALS - `0110 == 0011 = false;`

DIFFERENT - `0110 != 0011 = true;`

Problemas no Valladolid

496- Simply Subsets

11583- Alien DNA

353- Pesky Palindromes

11601- Avoiding Overlaps

562- Dividing Coins

10664- Luggage

Edição:

Caio César F. A. Lima

Supervisão:

Paulo E. Duarte Pinto

Revisão e conteúdo adicional:

Bianca Rosa

Giancarlo França

Colaboração:

Anderson Napoleão de Mello

David Barreto Ferreira

Denilson Guimarães Tavares

Diogo Soares de Souza

Felipe Lemos Prado

José Antônio Pereira de Oliveira

Juan Pedro Alves Lopes

Moyses da Silva Sampaio Junior

Otávio de Andrade Cardoso

Pedro Henrique Ribeiro

Raquel Marcolino de Souza