Universidade de Brasília
Departamento de Ciência da Computação

Fundamentos da Linguagem C++
CIC0169 - Programação Competitiva

Prof. Dr. Vinícius Ruela Pereira Borges

viniciusrpb@unb.br

Brasília-DF, 2020

Informação

- Esses slides foram redigidos e produzidos pelo Prof. Dr. Vinícius R. P. Borges;

- Material didático de referência:

  Halim S., Halim F., *Competitive Programming 3: The New Lower Bound of Programming Contests*, 3ª ed, 2018.

  - Textos do repositório do Grupo UnBalloon [1]

[1] `https://github.com/UnBalloon`

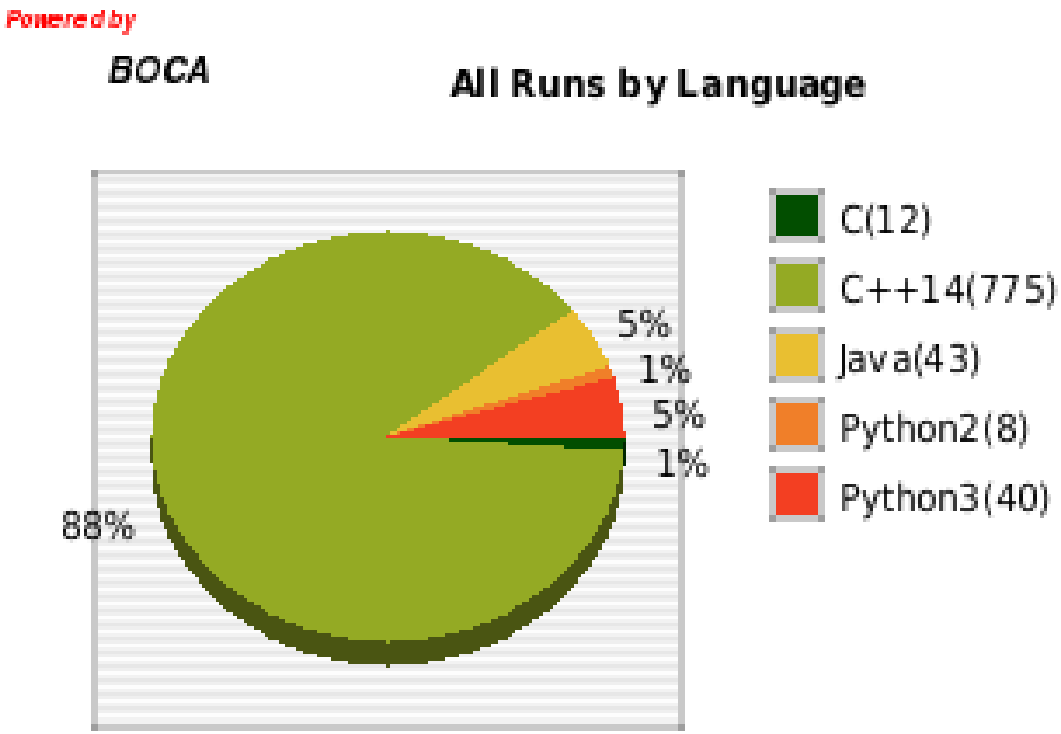
Roteiro

- Motivação
- Fundamentos C++
- Tipos de dados e variáveis
- Leitura e escrita de dados
- Strings
- Pairs e Iterators

Motivação

Porquê C++?

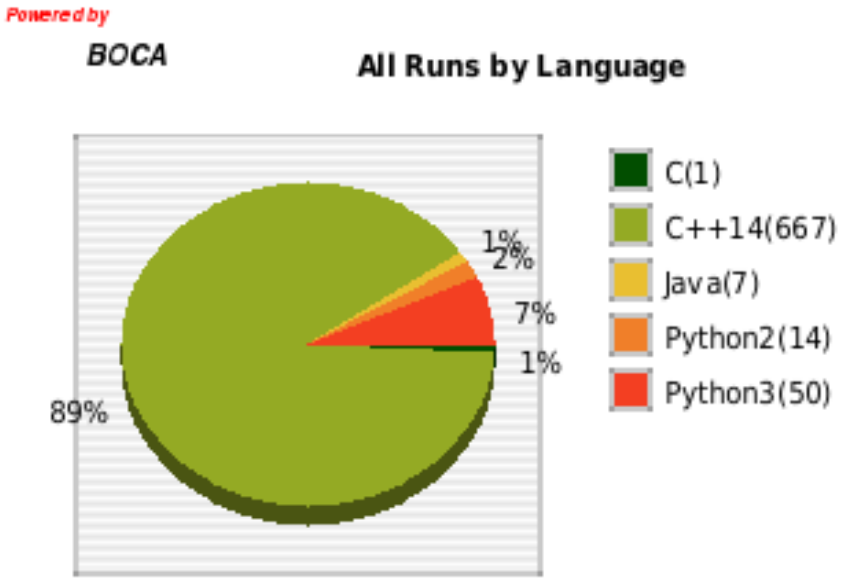
- Contabilização das linguagens de programação de acordo com todas as submissões na final brasileira da Maratona SBC de Programação 2018:



Extraído do site oficial da Maratona SBC de Programação.

Porquê C++?

- Contabilização das linguagens de programação de acordo com todas as submissões na final brasileira da Maratona SBC de Programação 2019:



Extraído do site oficial da Maratona SBC de Programação.

Porquê C++?

- As linguagens C/C++ possibilitam o uso dos comandos `typedefs` e *macro-features* que são empregadas frequentemente por competidores visando:

Porquê C++?

- As linguagens C/C++ possibilitam o uso dos comandos `typedefs` e *macro-features* que são empregadas frequentemente por competidores visando:
  - conveniência

Porquê C++?

- As linguagens C/C++ possibilitam o uso dos comandos `typedefs` e *macro-features* que são empregadas frequentemente por competidores visando:
  - conveniência
  - sumarização de código-fonte

Porquê C++?

- As linguagens C/C++ possibilitam o uso dos comandos `typedefs` e *macro-features* que são empregadas frequentemente por competidores visando:
  - conveniência
  - sumarização de código-fonte
  - velocidade de programação

Porquê C++?

- As linguagens C/C++ possibilitam o uso dos comandos `typedefs` e *macro-features* que são empregadas frequentemente por competidores visando:
  - conveniência
  - sumarização de código-fonte
  - velocidade de programação
  - velocidade de execução de programas em C++

Fundamentos C++

Estrutura geral de um programa em C

- Em linguagem C, pode-se definir

```
1  #include<stdio.h>
2
3  int main()
4  {
5
6      /* Declaracao de variaveis*/
7
8      /* Comandos principais */
9
10     return 0;
11 }
```

Estrutura geral de um programa em C++

- Em linguagem C++, temos a seguinte estrutura:

```
1  #include<bits/stdc++.h>
2
3  using namespace std;
4
5  int main()
6  {
7
8      /* Declaracao de variaveis*/
9
10     /* Comandos principais */
11
12     return 0;
13 }
```

Estrutura geral de um programa em C++

C

```c
#include<stdio.h>

int main()
{

    /* Declaracao de variaveis
       */

    /* Comandos principais */

    return 0;
}
```

C++

```cpp
#include<bits/stdc++.h>

using namespace std;

int main()
{

    /* Declaracao de variaveis
       */

    /* Comandos principais */

    return 0;
}
```

Biblioteca STL

- Na programação C++ para competições, emprega-se a bibliotaca *Standard Template Library* (STL)

- Uma das principais vantagens desta biblioteca consiste na simplificação do esforço de programação com algoritmos de ordenação e estruturas de dados, como:
  - estáticas (vetores, matrizes)
  - lineares dinâmicas (listas, pilhas, filas, filas de prioridade, árvores de busca binária)
  - lineares estáticas (árvores de busca binária)

Estrutura geral de um programa em C++

- O comando `include` incorpora todos os recursos da biblioteca STL no código-fonte

```
1  #include<bits/stdc++.h>
```

Estrutura geral de um programa em C++

- O comando `include` incorpora todos os recursos da biblioteca STL no código-fonte

```
1 #include<bits/stdc++.h>
```

- O comando abaixo indica que o escopo dos objetos e classes da STL devem valer para o código-fonte inteiro

```
1 using namespace std;
```

Estrutura geral de um programa em C++

- Por exemplo, repare o uso de `std::` antes da declaração de cada objeto abaixo:

```
1  #include<bits/stdc++.h>
2
3  int main()
4  {
5      std::list<int> l;
6      std::vector<int> v;
7
8      v.push_back(1);
9
10     /* ... */
11
12     return 0;
13 }
```

Estrutura geral de um programa em C++

- Incorporando o `comando namespace`, veja como o comando `std` se torna menos repetitivo no código-fonte:

```
1  #include<bits/stdc++.h>
2
3  using namespace std;
4
5  int main()
6  {
7      list<int> l;
8      vector<int> v;
9
10     v.push_back(1);
11
12     /* ... */
13
14     return 0;
15 }
```

Compilando um código-fonte C++

- Suponha que o arquivo de código-fonte seja nomeado como `cf578b.cpp`

Compilando um código-fonte C++

- Suponha que o arquivo de código-fonte seja nomeado como `cf578b.cpp`

- Para compilar o código-fonte, digita-se o comando:

```
1  g++ cf578b.cpp -std=c++11 -o binr
```

Compilando um código-fonte C++

- Suponha que o arquivo de código-fonte seja nomeado como `cf578b.cpp`

- Para compilar o código-fonte, digita-se o comando:

```
1 g++ cf578b.cpp -std=c++11 -o binr
```

em que **`binr`** é nome do arquivo binário (executável) gerado.

- Para executar o programa:

```
1 ./binr
```

Tipos de Dados

Tipos de Dados

Tipo	Memória (bits)	Intervalo
`char`	8	$[-128, 127]$

`char` é associado a um inteiro de 8 bits

Tipos de Dados

Tipo	Memória (bits)	Intervalo
`char`	8	$[-128, 127]$
`unsigned char`	8	$[0, 255]$

`char` é associado a um inteiro de 8 bits

Tipos de Dados

Tipo	Memória (bits)	Intervalo
`char`	8	$[-128, 127]$
`unsigned char`	8	$[0, 255]$
`short`	16	$[-32768, 32767]$
`unsigned short`	16	$[0, 65535]$

`char` é associado a um inteiro de 8 bits

Tipos de Dados

Tipo	Memória (bits)	Intervalo
`char`	8	$[-128, 127]$
`unsigned char`	8	$[0, 255]$
`short`	16	$[-32768, 32767]$
`unsigned short`	16	$[0, 65535]$
`int`	32	$\approx [-2 \times 10^9, 2 \times 10^9]$
`unsigned int`	32	$[0, 4 \times 10^9]$

`char` é associado a um inteiro de 8 bits

Tipos de Dados

Tipo	Memória (bits)	Intervalo
`char`	8	$[-128, 127]$
`unsigned char`	8	$[0, 255]$
`short`	16	$[-32768, 32767]$
`unsigned short`	16	$[0, 65535]$
`int`	32	$\approx [-2 \times 10^9, 2 \times 10^9]$
`unsigned int`	32	$[0, 4 \times 10^9]$
`long long`	64	$\approx [-9 \times 10^{18}, 9 \times 10^{18}]$
`unsigned long long`	64	$[0, 10^{19}]$

`char` é associado a um inteiro de 8 bits

Tipos de Dados

Tipo	Memória (bits)	Precisão
`float`	32	7 casas decimas
`double`	64	15 casas decimais

- Existe o tipo `long double`, que possui precisão e espaço em memória mínimos em relação ao tipo `double`;
- No GCC do Linux, uma variável `long double` ocupa 80 bits na memória, possuindo 18 casas decimais de precisão;

Dúvidas

- Qual a diferença entre os especificadores das variáveis abaixo?

```
1  long l;
2  long int li;
3  long long ll;
4  long long int lli;
```

Dúvidas

- As variáveis abaixo:

```
1  long l;
2  long int li;
```

- A existência desses dois especificadores se deve, indiretamente, à mudança das arquiteturas 32 bits para 64 bits e as diferentes versões de compiladores C/C++

Dúvidas

- Agora, garante-se que as variáveis abaixo ocupem 64 bits.

```
1  long long ll;
2  long long int lli;
```

Leitura e Escrita

Leitura/Escrita de Dados

- Nas competições de programação, os dados são lidos e impressos a partir da entrada e saída padrão (*Standard Input/Output*)

Leitura/Escrita de Dados

- Nas competições de programação, os dados são lidos e impressos a partir da entrada e saída padrão (*Standard Input/Output*)

- Na linguagem C++, podemos usar as funções `scanf` e `printf` normalmente:

```
1  #include<bits/stdc++.h>
2
3  using namespace std;
4
5  int main()
6  {
7      double var_d;
8
9      scanf("%lf",&var_d);
10     printf("%lf\n",var_d);
11     return 0;
12 }
```

Leitura/Escrita de Dados

- Pode-se também empregar os objetos da classe `iostream` `cin` e `cout`, que estão inclusos na biblioteca STL

Leitura/Escrita de Dados

- Pode-se também empregar os objetos da classe `iostream` `cin` e `cout`, que estão inclusos na biblioteca STL

- O comando `cout` é uma `stream` de saída padrão que escreve caracteres como dados formatados utilizando o operador de extração (<<);

```
1  #include<bits/stdc++.h>
2
3  using namespace std;
4
5  int main()
6  {
7      cout << "Hello world!\n";
8
9      return 0;
10 }
```

Leitura/Escrita de Dados

- Pode-se também empregar os objetos da classe `iostream` `cin` e `cout`, que estão inclusos na biblioteca STL
- O comando `cin` corresponde ao `stream` da linguagem C `stdin`
- Nesse caso, os caracteres podem ser lidos da fonte (por exemplo, o teclado) de maneira formatada utilizando o operador de extração (>>);

Leitura/Escrita de Dados

- Por exemplo:

```
1  #include<bits/stdc++.h>
2
3  using namespace std;
4
5  int main()
6  {
7      int age;
8
9      cout << "Digite sua idade: \n";
10     cin >> age;
11     cout << "Sua idade eh " << age << "\n";
12
13     return 0;
14 }
```

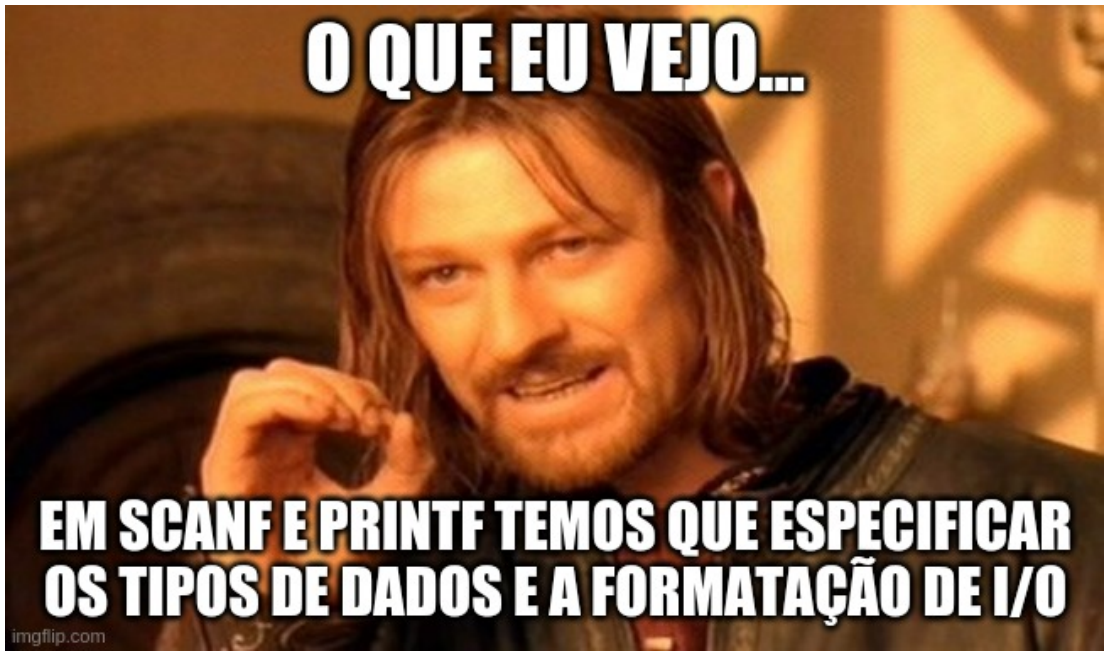
Leitura/Escrita de Dados

- Apesar desses comandos possuírem complexidade computacional constante ($O(1)$), o tempo de execução de `cin` e `cout` é consideravelmente maior em relação a `scanf` e `printf`

- Qual o motivo?

Leitura/Escrita de Dados

- Apesar desses comandos possuírem complexidade computacional constante ($O(1)$), o tempo de execução de `cin` e `cout` é consideravelmente maior em relação a `scanf` e `printf`



Leitura/Escrita de Dados

- Apesar desses comandos possuírem complexidade computacional constante ($O(1)$), o tempo de execução de `cin` e `cout` é consideravelmente maior
- Internamente, sabe-se que `iostream` incorpora o sistema de buffer da biblioteca `stdio`
  - além disso, o fluxo de dados na entrada e saída é único
- Por isso, `cin` gasta mais tempo para sincronizar com esse buffer, fazendo com que as chamadas `cin` e `scanf` necessitam ser intercaladas.

Leitura/Escrita de Dados

- Pode-se desabilitar a sincronização de `scanf` e `cin` em processo de entrada e saída utilizando a função:

```
1 ios::sync_with_stdio(false);
```

- Além disso, inclui-se o método `tie()` para garantir a liberação dos dados na saída (no método `cout`) antes de que novos dados de entrada sejam lidos (por meio de `cin`)

```
1 cin.tie(NULL);
```

Leitura/Escrita de Dados

- Obtém-se então o seguinte código-fonte:[2]

```
1  #include<bits/stdc++.h>
2
3  using namespace std;
4
5  int main()
6  {
7      std::ios::sync_with_stdio(false);
8      cin.tie(NULL);
9
10     int x;
11     cin >> x;
12     cout << x << "\n";
13
14     return 0;
15 }
```

[2] Extraído de https://www.geeksforgeeks.org/fast-io-for-competitive-programming/

Formato de Leitura/Escrita de Dados

Tipo	Formato
`char`	`%c`
`short int`	`%hd`
`int`	`%d`
`unsigned int`	`%u`
`long long int`	`%lld`
`float`	`%f`
`double`	`%lf`
`long double`	`%Lf`
‘‘`string`’’ (`char *`)	`%s`

`char` é associado a um inteiro de 8 bits

Cuidado...

- Repare no trecho de código-fonte a seguir:

```
1
2   /* ... */
3
4   int x = 100000;   /* 10^5 */
5   int y = 100000;   /* 10^5 */
6
7   int z = x*y;
8
9   printf("z = %d\n",z);
10
11  /* ... */
```

- Qual a saída retornada por este trecho de código-fonte?

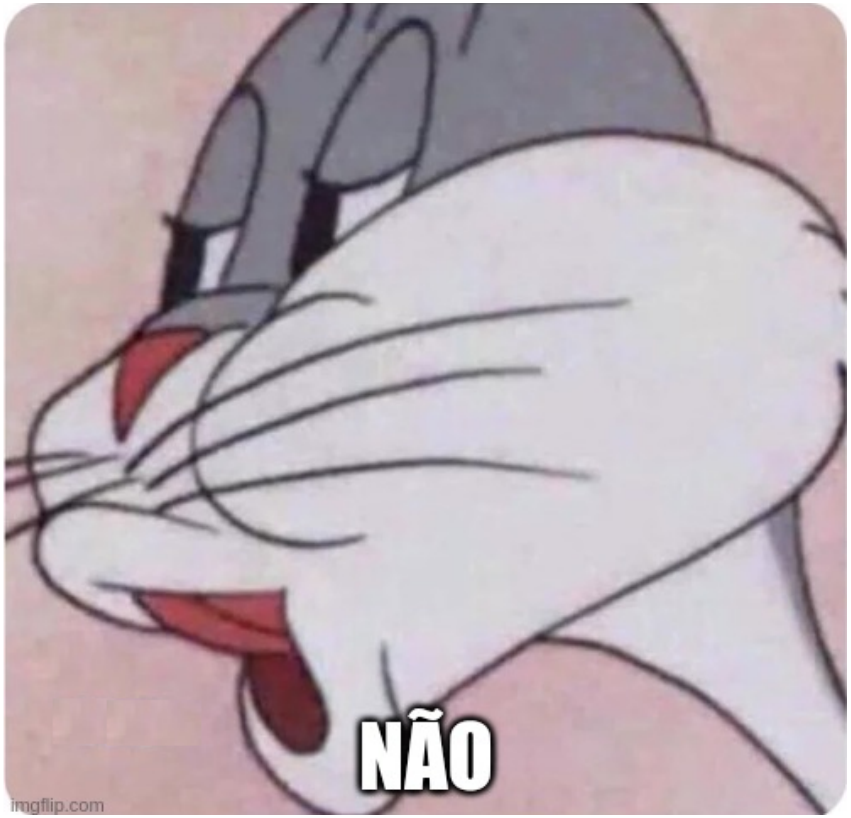
Cuidado...

- Repare no trecho de código-fonte a seguir:

```
1
2   /* ... */
3
4   int x = 100000;   /* 10^5 */
5   int y = 100000;   /* 10^5 */
6
7   int z = x*y;
8
9   printf("z = %d\n",z);
10
11  /* ... */
```

- Qual a saída retornada por este trecho de código-fonte?
  - Se $x = 10^5$ e $y = 10^5$, o programa imprime 10^{10}...

Cuidado...



Overflow

- Repare que as variáveis `x`, `y` e `z` são do tipo `int`
  - Cada uma consegue armazenar o valor $\approx 2 \times 10^9$
- O valor calculado e armazenado em z é 10^{10}
  - tal valor extrapola 2×10^9
- Diz-se então que ocorreu um *overflow* na variável z!

Overflow

- O trecho de código-fonte correto é:

```
1  /* ... */
2
3  long long int x = 100000;
4  long long int y = 100000;
5
6  long long int z = x*y;
7
8  printf("z = %lld\n",z);
9
10 /* ... */
```

Strings

Strings

- É uma classe na linguagem C++ cujo objeto representa uma sequência de caracteres

- Podemos declarar uma string conforme:

  `string nomevar;`
  `string nomevar` = *constante*;
  `string nomevar` = `char *  variavel;`
  `string nomevar(char *  variavel);`
  `string nomevar`(*tamanho*, constante `char`);

Strings

- É uma classe na linguagem C++ cujo objeto representa uma sequência de caracteres

```
1  string a = "abc";
2  string b = "def";
```

- **Concatenação:** utilizando o operador + entre duas strings (variável ou constante), podemos obter uma nova string

```
1  #include <bits/stdc++.h>
2
3  using namespace std;
4
5  int main()
6  {
7      string a = "abc";
8      string b = "def";
9      string c = a+b;
10
11     cout << c << endl;
12     printf("%s\n",c.c_str()); /* esse comando gera o
           mesmo efeito no cout na l. 11*/
13
14     return 0;
15 }
```

Strings

- A função to_string converte um valor inteiro para uma string:

```
1  #include <bits/stdc++.h>
2
3  using namespace std;
4
5  int main()
6  {
7      int x = 1320;
8      string a = to_string(x);
9
10     cout << a << endl;
11
12     return 0;
13 }
```

Pairs

Pairs

- *Pair* é um contêiner que consiste de dois elementos de dados ou objetos
- O primeiro elemento é referenciado como `first` e o segundo como `second`, sendo que a ordem de armazenamento é dada por (`first`, `second`)
- Um *Pair* pode combinar elementos de diferentes tipo ou classe entre si, formando um objeto único
- Objetos *Pair* podem ser atribuídos, copiados e comparados, como também formar *arrays* e outras estruturas

Pairs

- Podemos declarar um `pair` como:

  `pair<tipo de dados 1, tipo de dados 2> variavel;`

- e iniciá-lo por meio do comando `make_pair`:

  `variavel` = `make_pair`(constante `tipo de dados 1`, constante `tipo de dados 2`);

Pairs

- Exemplo 1: um `pair` que armazena dois valores inteiros

```
1  #include<bits/stdc++.h>
2
3  using namespace std;
4
5  int main()
6  {
7      pair<int,int> var_pii;
8
9      var_pii = make_pair(-2,4);
10
11     printf("(%d,%d)\n",var_pii.first,var_pii.second);
12
13     return 0;
14 }
```

Pairs

- Exemplo 2: um `pair` que armazena um valor `int` e outro `double`

```
1  #include<bits/stdc++.h>
2
3  using namespace std;
4
5  int main()
6  {
7      pair<int,double> var_pid;
8
9      var_pid = make_pair(100,3.25);
10
11     printf("(%d,%lf)\n",var_pid.first,var_pid.second);
12     // cout << "(" << var_pid.first << "," << var_pid.
           second << ")\n";
13
14     return 0;
15 }
```

Pairs

- Exemplo 3: um `pair` que armazena uma `string` e outro valor `int`

```
1  #include<bits/stdc++.h>
2
3  using namespace std;
4
5  int main()
6  {
7      pair<string,int> var_pstri;
8
9      var_pstri = make_pair("Tom and Jerry",1);
10
11     printf("(%s,%d)\n",var_pstri.first.c_str(),
           var_pstri.second);
12
13     return 0;
14 }
```

Pairs

- Exemplo 4: comparando variáveis `pair` conforme seus elementos `first`

```
1  #include<bits/stdc++.h>
2
3  using namespace std;
4
5  int main()
6  {
7      pair<int,int> v1,v2;
8
9      v1 = make_pair(3,1);
10     v2 = make_pair(2,2);
11
12     if(v1 > v2)  // equivalente a v1.first > v2.first
13         printf("v1 > v2\n");
14     else
15         printf("v2 >= v1\n");
16
17     return 0;
18 }
```

Pairs

- Exemplo 5: comparando variáveis `pair` utilizando seus elementos `second`

```
1  #include<bits/stdc++.h>
2
3  using namespace std;
4
5  int main()
6  {
7      pair<int,int> v1,v2;
8
9      v1 = make_pair(3,1);
10     v2 = make_pair(2,2);
11
12     if(v1.second > v2.second)
13         printf("v1 > v2\n");
14     else
15         printf("v2 >= v1\n");
16
17     return 0;
18 }
```

Vectors e iterators

Vector

- `vector` pode ser entendido como uma estruturas de dados similar a um `array` de tamanho expansível;

- A diferença principal entre `vector` e array é a **alocação**: no array adota-se alocação estática, enquanto que no `vector` a alocação é dinâmica;

```
1  vector<int> vi;
2  vector<double> vd;
3  vector<pair<int,double>> vid;
4  vector<string> vs;
```

Vector

- É possível definir o tamanho do vetor e valores iniciais para cada um de seus elementos. Veja:

```
1  vector<int> v(100,5);
```

- No trecho de código-fonte acima significa, um vetor de inteiros com 100 posições foi criado, cujos todos os elementos são iguais a 5.

Vector

- O método `size()` retorna a quantidade de elementos existentes em um `vector`. A complexidade é $O(1)$;

```
1  int n = v.size();
```

Vector

- `push_ back` é a função utilizada para inserir elementos em um objeto vector com complexidade $O(1)$.

```
1  #include <bits/stdc++.h>
2
3  using namespace std;
4
5  int main()
6  {
7      vector<int> v;
8
9      v.push_back(9);
10     v.push_back(2);
11     v.push_back(3);
12
13     printf("%d %d %d\n",v[0],v[1],v[2]);
14
15     return 0;
16 }
```

Vector: o que o código-fonte abaixo faz?

```
1  #include <bits/stdc++.h>
2
3  using namespace std;
4
5  int main()
6  {
7      int n,aux;
8      vector<int> v;
9
10     scanf("%d",&n);
11
12     for(int i = 0; i < n; i++)
13     {
14         scanf("%d",&aux);
15         v.push_back(aux);
16     }
17
18     for(int i = 0; i < v.size(); i++)
19     {
20         printf("%d",v[i]);
21     }
22
23     return 0;
24 }
```

Iterators

- Iterators são tipos específicos de ponteiros que referenciam endereços de memória de objetos e contêiners STL

```
1  #include<bits/stdc++.h>
2
3  using namespace std;
4
5  int main()
6  {
7      vector<int> v= { 1, 2, 3, 4, 5 };
8      vector<int>::iterator ptr;
9
10     printf("Elementos do vetor: ");
11     for(ptr = v.begin(); ptr < v.end(); ptr++)
12         printf("%d ",*ptr);
13
14     printf("\n");
15
16     return 0;
17 }
```

Demais funções e comandos

Funções min e max

- A funções `min` e `max` comparam dois valores e retornam o menor e maior valores respectivamente

```
1  #include<bits/stdc++.h>
2
3  using namespace std;
4
5  int main()
6  {
7      double v[3] = {3.562,6.3232,1.762};
8      double menor = min(v[0],v[2]);
9      double maior = max(v[2],v[1]);
10
11     printf("menor: %lf maior: %lf\n",menor,maior);
12
13     return 0;
14 }
```

Define

- `Define` é uma expressão pré-definida pelo programador que denota que um comando b será tratado no código-fonte como a

```
1 #define a b
```

- Exemplo: O tipo de dados `long long int` será conhecido no código-fonte apenas por `ll`;

```
1 #define ll long long int
```

Template

```
1  #include <bits/stdc++.h>
2
3  #define vi vector<int>
4  #define ll long long
5  #define pb push_back
6  #define mp make_pair
7  #define ii pair<int,int>
8  #define N 100000
9
10 using namespace std;
11
12 int main()
13 {
14     vector<int> v;
15     v.pb(N); /* insere 100000 no vector v */
16     vector<ii> v2;  /* vetor de pair<int,int> */
17     v2.pb(mp(1,0)); /* inserindo o par (1,0) no vetor de par
           */
18
19     return 0;
20 }
```

Define

- O *template* geral de um programador em competições de programação:

```
1  #include <bits/stdc++.h>
2
3  #define vi vector<int>
4  #define ll long long
5  #define pb push_back
6  #define mp make_pair
7  #define ii pair<int,int>
8  #define N 100000
9
10 using namespace std;
11
12 int main()
13 {
14     /* ... */
15
16     return 0;
17 }
```

Auto

- Podemos pedir para o C++ inferir o tipo de uma variável determinando seu tipo como `auto`:

```
1  #include <bits/stdc++.h>
2
3  using namespace std;
4
5  int main()
6  {
7      auto val = 4;
8      auto val2 = 5.21;
9      auto par = make_pair(2,4);
10
11     printf("%d : %lf\n",val,val2);
12     printf("[%d,%d]\n",par.first,par.second);
13
14     return 0;
15 }
```

Ordenação

Função sort

- A função sort pode ser empregada para ordenar os elementos de um vetor (array ou `vector`);
- É garantido que a complexidade da função `sort` é $O(nlogn)$ [3].

[3] Fonte: `http://www.cplusplus.com/reference/algorithm/sort/?kw=sort`

Função sort: ordenação ascendente

```
1  #include<bits/stdc++.h>
2
3  using namespace std;
4
5  int main()
6  {
7      vector<int> v;
8
9      v.push_back(4);
10     v.push_back(3);
11     v.push_back(2);
12     v.push_back(1);
13
14     sort(v.begin(),v.end(),less<int>());
15
16     for(int i = 0; i < v.size(); i++)
17     {
18         printf("%d ",v[i]);
19     }
20
21     return 0;
22 }
```

Função sort: ordenação descendente

```
1  #include<bits/stdc++.h>
2
3  using namespace std;
4
5  int main()
6  {
7      vector<int> v;
8
9      v.push_back(4);
10     v.push_back(3);
11     v.push_back(2);
12     v.push_back(1);
13
14     sort(v.begin(),v.end(),greater<int>());
15
16     for(int i = 0; i < v.size(); i++)
17     {
18         printf("%d ",v[i]);
19     }
20
21     return 0;
22 }
```

Função de comparação

- Função que pode ser implementada como um critério de comparação em funções como `sort` e `upper_bound`/(`lower_bound`)

Função de comparação

- Função que pode ser implementada como um critério de comparação em funções como `sort` e `upper_bound`/(`lower_bound`)

- Também são implementadas na estrutura de dados fila de prioridade;

Função de comparação

- Função que pode ser implementada como um critério de comparação em funções como `sort` e `upper_bound`/(`lower_bound`)
- Também são implementadas na estrutura de dados fila de prioridade;
- Lembrando da função `sort`:

```
1 sort(v.begin(),v.end(), comparacao)
```

Função de comparação

- Exemplo de uma função de comparação:

```
1  bool comparacao(int a, int b){
2    if(a > b) return true;
3    return false;
4  }
```

- O que nossa função de comparação quer saber é se o valor de a deve vir antes do valor de b no vetor final ordenado;

- A função retorna `true` se o valor $a > b$, ou seja, a deve aparecer antes de b;

Função de comparação

- Exemplo de uma função de comparação:

```
1  bool comparacao(int a, int b){
2    if(a > b) return true;
3    return false;
4  }
```

- O que nossa função de comparação quer saber é se o valor de a deve vir antes do valor de b no vetor final ordenado;
- Caso $b >= a$, a função retorna `false`, o que indica que esses valores devem ser trocados de posição;
- Assim, o vetor será ordenado de forma descendente.

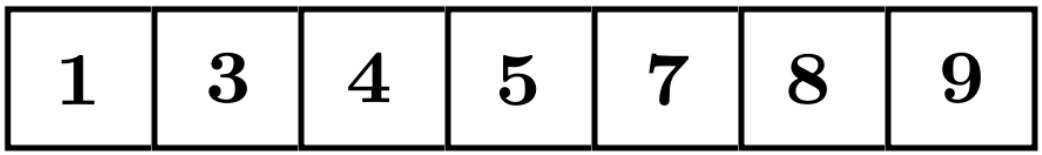
Função sort: ordenação descendente

```
1  #include<bits/stdc++.h>
2
3  using namespace std;
4
5  bool comparacao(int a, int b){
6    if(a > b) return true;
7    return false;
8  }
9
10 int main()
11 {
12
13     /* ... */
14
15     sort(v.begin(),v.end(),comparacao);
16
17     /* ... */
18
19     return 0;
20 }
```

Um pouco de busca binária

Upper bound

- O método `upper_bound` executa busca binária em um vetor para encontrar o primeiro elemento estritamente maior do que um determinado valor;

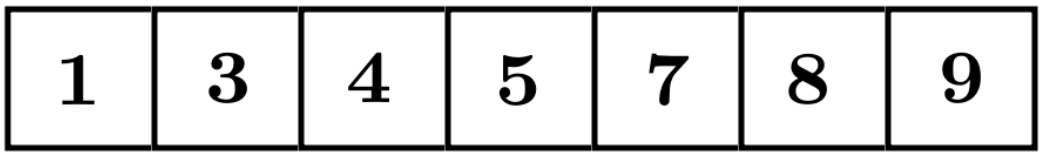


Upper bound

- O método `upper_bound` executa busca binária em um vetor para encontrar o primeiro elemento estritamente maior do que um determinado valor;

Upper bound

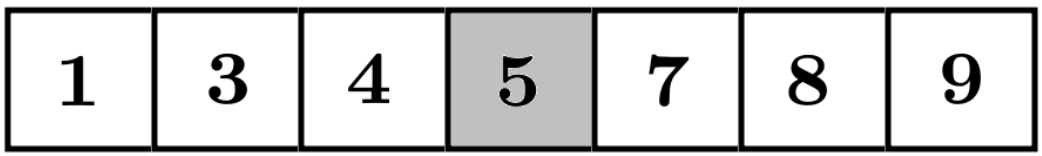
- O método `upper_bound` executa busca binária em um vetor para encontrar o primeiro elemento estritamente maior do que um determinado valor;



$$\mathbf{key} = \mathbf{5}$$

Upper bound

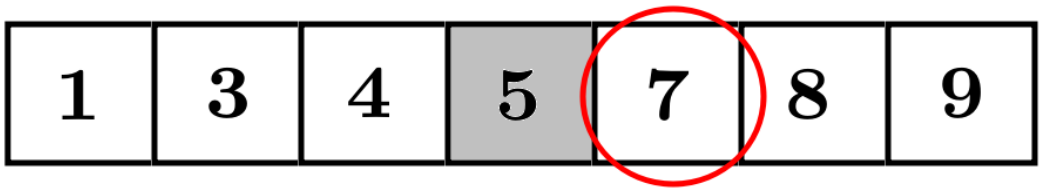
- Se o elemento a ser buscado é 5 e é encontrado



$$\mathbf{key = 5}$$

Upper bound

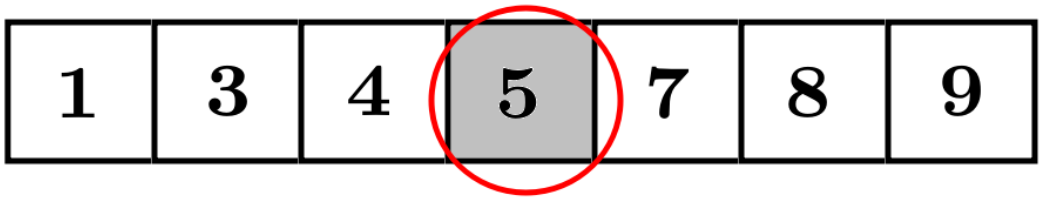
- Encontra-se o elemento estritamente superior a 5, que é 7



$$\mathbf{key} = \mathbf{5}$$

Lower bound

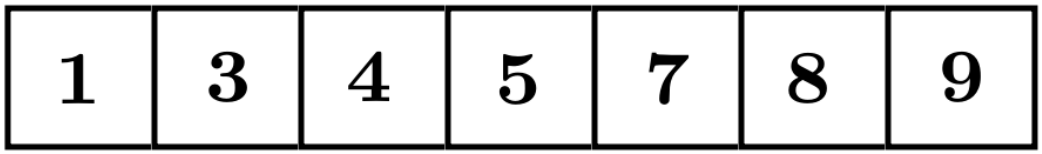
- O método `lower_bound` executa busca binária em um vetor para encontrar o primeiro elemento maior ou igual do que um determinado valor.



$$\mathbf{key} = \mathbf{5}$$

Lower bound

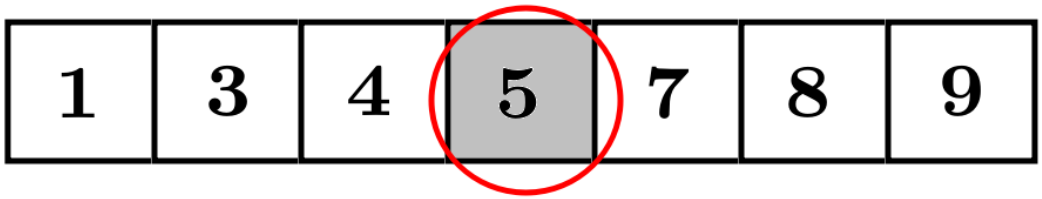
- O método `lower_bound` executa busca binária em um vetor para encontrar o primeiro elemento maior ou igual do que um determinado valor.



$$\mathbf{key} = \mathbf{5}$$

Lower bound

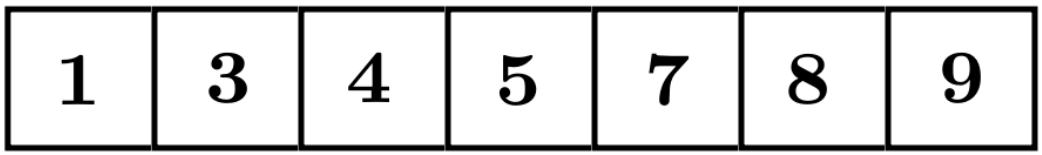
- Resposta do `lower_bound` para o elemento de busca igual a 5 é 5.



$$\mathbf{key} = \mathbf{5}$$

Lower bound

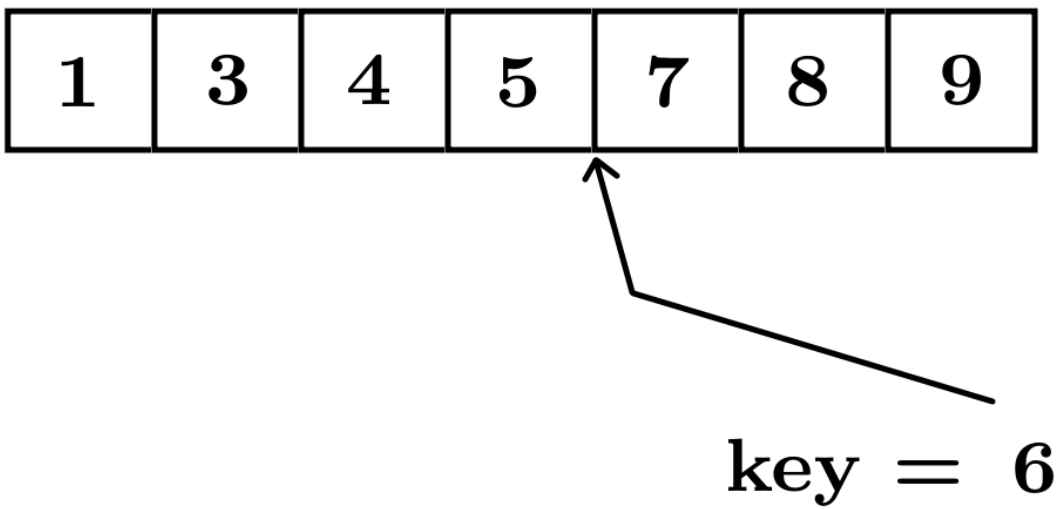
- Agora queremos encontrar o primeiro elemento maior ou igual a 6



$$\mathbf{key} = \mathbf{6}$$

Lower bound

- Embora 6 não esteja presente nesse vetor, a resposta de `lower_bound` é 7.



Upper bound e lower bound

- Os métodos `upper_bound` e `lower_bound` retornam um iterator, sendo necessário verificar se a operação de busca foi bem sucedida:

Upper bound e lower bound

- Os métodos `upper_bound` e `lower_bound` retornam um iterator, sendo necessário verificar se a operação de busca foi bem sucedida:
  - se um elemento for encontrado, é retornado um iterator que aponta para seu endereço no vetor;
  - caso contrário, é retornado o iterator '.end()';

Upper bound e lower bound

- Os métodos `upper_bound` e `lower_bound` retornam um iterator, sendo necessário verificar se a operação de busca foi bem sucedida:
  - se um elemento for encontrado, é retornado um iterator que aponta para seu endereço no vetor;
  - caso contrário, é retornado o iterator '.end()';

- Complexidade é $O(logn)$.

Upper bound e lower bound

```
1  #include<bits/stdc++.h>
2
3  using namespace std;
4
5  int main()
6  {
7      vector<int> v;
8
9      v.push_back(1);
10     v.push_back(3);
11     v.push_back(4);
12     v.push_back(5);
13     v.push_back(7);
14     v.push_back(8);
15     v.push_back(9);
16
17     auto ub = upper_bound(v.begin(),v.end(),5);
18
19     auto lb = lower_bound(v.begin(),v.end(),5);
20
21     printf("Upperbound: %d Lowerbound: %d\n",*ub,*lb);
22
23     return 0;
24 }
```

Outras funções úteis

- `random_shuffle`;
- `reverse`;
- `next_permutation` e `prev_permutation`.

Universidade de Brasília
Departamento de Ciência da Computação

Fundamentos da Linguagem C++
CIC0169 - Programação Competitiva

Prof. Dr. Vinícius Ruela Pereira Borges

viniciusrpb@unb.br

Brasília-DF, 2020