

# Programação C++ para Jogos III



## Índice - Apresentação III

Namespaces .....	2
Biblioteca STL .....	6
Stack .....	7
Queue .....	11
Vector .....	14
Deque .....	22
Set .....	24
(Friend) .....	28
Map .....	29
Ellipsis .....	33

## Namespaces

Namespaces são como pacotes, nos quais o programador pode adicionar o que quiser: Variáveis, Classes, Funções, ou até mesmo outros namespaces.

Um projeto pode ter suas partes organizadas em Namespaces temáticos, como neste exemplo:

```
namespace Matematica
{
    const float PI = 3.1416;
    const float REL_AUREA = 1.625;
    const float E = 2.7182;

    float exp2(float v)
    {
        return v*v;
    }
}

int main(void) {
    // Acessa-se escrevendo NomeNamespace::componente
    float pi = Matematica::PI;
    cout << pi << endl;
    return 0;
}
```

O nome do Namespace pode ser omitido, se for incluída no início do programa a linha **using namespace**.

```
using namespace Matematica;
```

```
int main(void)
{
    float pi = PI;
    cout << pi << endl;
    return 0;
}
```

Desde o primeiro programa em C++ usamos um namespace chamado **std**. Neste namespace, dentre dezenas de outras coisas, estão **cin**, **cout**, **endl** e **string**. É possível usar estes componentes de outro modo, com **::**

```
#include <iostream>

int main(void)
{
    std::string s = "Palavra";
    std::cout << s << std::endl;
    return 0;
}
```

```
// Exemplo de namespace com componentes variados
```

```
namespace X
{
    int val = 30;

    void func(void)
    {
        cout << "Texto" << endl;
    }

    class A
    {
        public:
            char c;
    };

    struct B
    {
        char c;
    };

    namespace Y
    {
        float f = 40;
    }
}
```

## Biblioteca STL

- STL significa Standard Template Library. É uma biblioteca de C++ que contém algumas estruturas de dados, como Listas Encadeadas e Árvores Binárias.
- Estas estruturas de dados são chamadas Containers, e armazenam coleções de elementos. Funcionam como vetores, mas suas estruturas internas são diferentes, mais especializadas e mais otimizadas.
- Os Containers da STL são `Stack`, `Queue`, `Vector`, `Deque`, `List`, `Set`, `Multiset`, `Map`, `Multimap`, `Bitset` e `Priority Queue`.
- As estruturas mais comuns, e que serão abordadas neste material, são: `Stack`, `Queue`, `Vector`, `Deque`, `Set` e `Map`.
- Para conhecer a lista completa de Containers, entre em: <http://www.cplusplus.com/reference/stl/>

## Stack (Pilha)

Suponha que você está lavando pratos na cozinha de sua casa, e que cada prato lavado é colocado, momentaneamente, numa pilha de pratos ao seu lado.

Depois de uma certa quantidade de pratos a pilha fica muito grande e você decide pegá-los para guardar no armário. Você deve pegar de um em um, secar, e então colocar no armário.

Qual o prato que você pegará primeiro? Será o último que você empilhou, certo?



**Stacks** funcionam como pilhas de pratos. Elementos são apenas inseridos ou tirados, e o elemento tirado é sempre o que foi inserido por último.

Se forem inseridos em uma Stack de números inteiros os seguintes números: **6, 4, 8, 1, 9**

Eles serão retirados obrigatoriamente na ordem inversa: **9, 1, 8, 4, 6**.

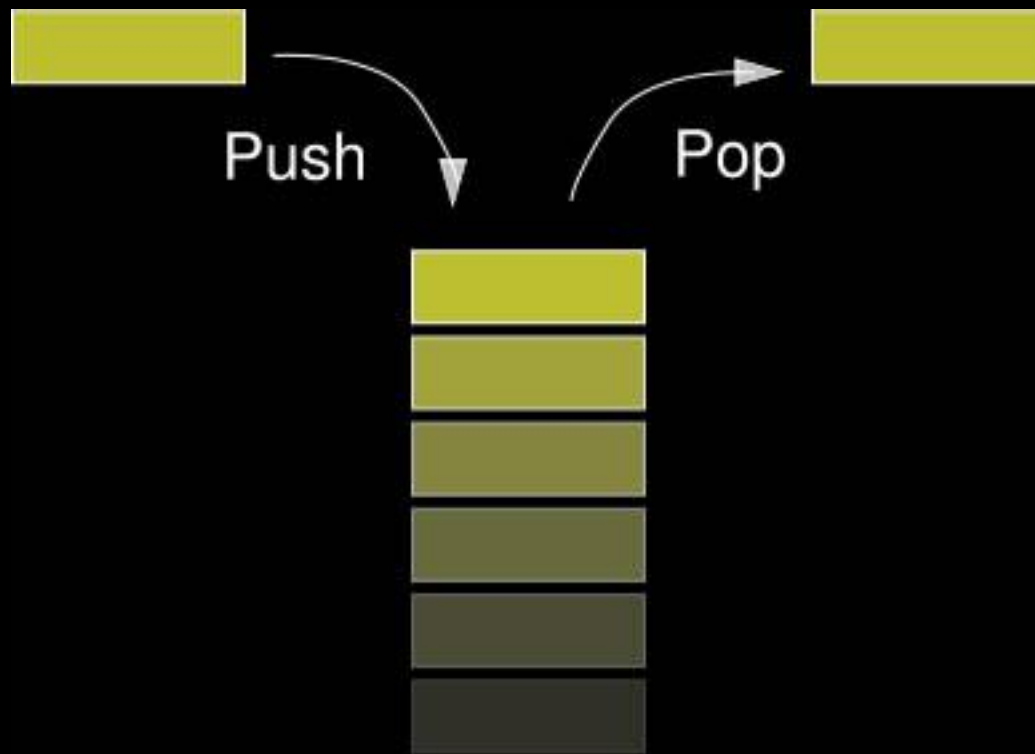
A vantagem do uso de Stacks é exatamente a sua especialização. Se tudo que você precisa no seu programa se resume a guardar números para eventualmente retirar na ordem inversa, a Stack deve ser usada, pela simplicidade e pela otimização.

Pilha errada →



O tipo `Stack<T>` possui os seguintes métodos:

`empty()` : retorna true caso a pilha esteja vazia;  
`size()` : retorna o número de elementos da pilha;  
`top()` : acessa o elemento do topo da pilha;  
`push(T)` : adiciona elemento no topo da pilha;  
`pop()` : retira o elemento do topo.



```

// Código Exemplo

#include <iostream>
#include <stack> // << Não esqueça de incluir...
using namespace std;

int main (void)
{
    // Declaração da pilha de inteiros.
    // Uma pilha pode ser de apenas um tipo de cada vez,
    // e este tipo é definido dentro de <>
    stack<int> s;

    s.push(6);
    s.push(4);
    s.push(8);
    s.push(1);
    s.push(9);

    cout << "size(): " << s.size() << endl;

    cout << "Imprimindo pilha: ";
    while(!s.empty()) {
        cout << s.top() << " ";
        s.pop();
    }
    cout << endl << "size(): " << s.size() << endl;
    return 0;
}

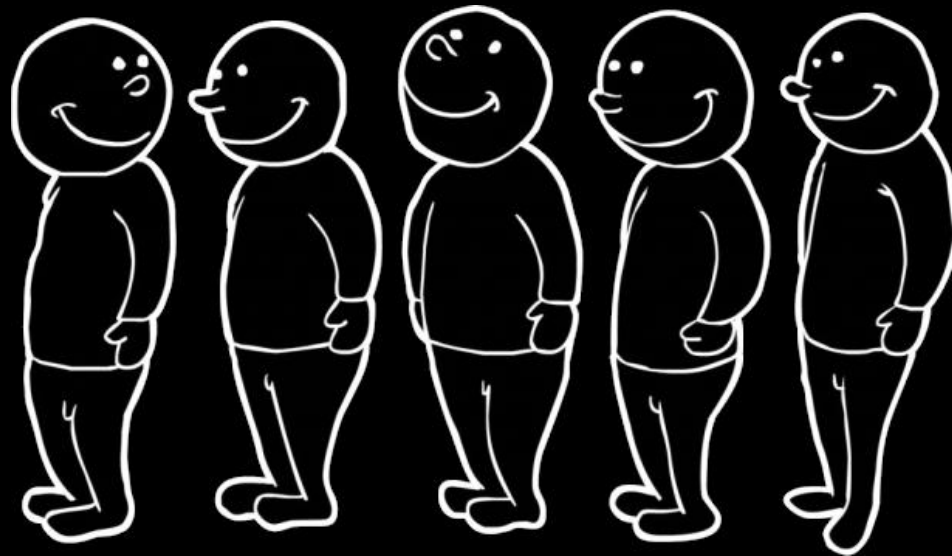
```



## Queue (Fila)

O tipo `Queue` pode também ser comparado a uma entidade cotidiana. Pense no funcionamento de uma fila qualquer, como a de um banco. Quem chega por último deve entrar no final da fila. E o primeiro a ser atendido é quem chegou antes de todos os outros.

Em uma `Queue`, elementos são retirados na mesma ordem que foram incluídos. Exatamente o contrário da `Stack`.



Queues funcionam como filas comum. São inseridos elementos, como numa pilha, mas os primeiros a sair são os que estão no início.

Se forem inseridos em uma Queue de números inteiros os seguintes números: 6, 4, 8, 1, 9

Eles serão retirados nesta mesma ordem: 6, 4, 8, 1, 9.

O tipo Queue<T> possui os seguintes métodos:

empty() : retorna true caso a fila esteja vazia;  
size() : retorna o número de elementos da fila;  
front() : acessa o primeiro elemento da fila;  
back() : acessa o último elemento da fila;  
push(T) : adiciona elemento no fim da fila;  
pop() : retira o elemento do início da fila.

- Lembrando que os Ts significam "Qualquer tipo", porém apenas um por fila.



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

int main(void)
{
    queue<int> q;
    q.push(6);
    q.push(4);
    q.push(8);
    q.push(1);
    q.push(9);

    cout << "size(): " << q.size() << endl;
    cout << "front(): " << q.front() << endl;
    cout << "back(): " << q.back() << endl;
    cout << "Imprimindo fila: ";

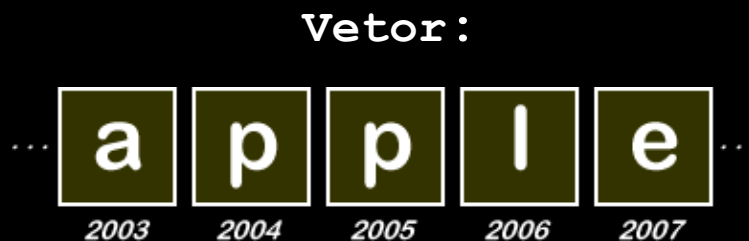
    while(!q.empty()) {
        cout << q.front() << " ";
        q.pop();
    }
    cout << endl << "size(): " << q.size() << endl;
    return 0;
}
```



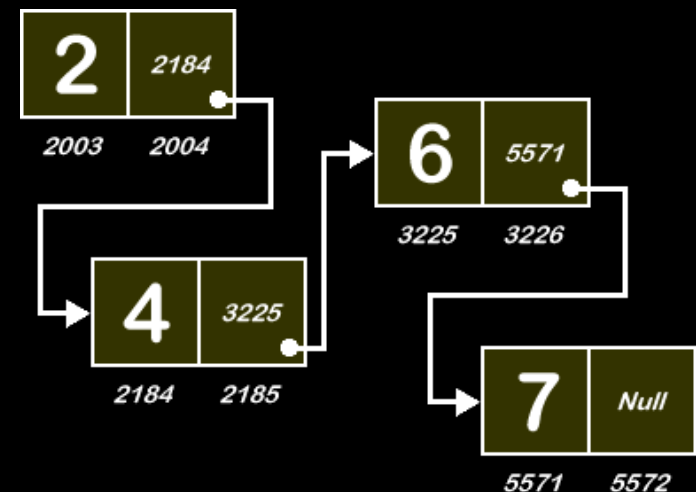
## Vector

Vector é um Container muito similar à estrutura de dado chamada **Lista Encadeada**. Listas Encadeadas são sequenciais como vetores, mas não possuem índices, e seus elementos podem ser deletados sem deixar buracos na estrutura. Em vetores não é possível apagar elementos.

Observação: Vetores ocupam posições seguidas de memória, e isto torna rápido o acesso aos seus elementos. Listas Encadeadas, embora funcione como uma sequência, têm na verdade seus elementos em posições aleatórias de memória. Listas não são lentas, mas são mais lentas do que vetores.



## Lista Encadeada:



O tipo `Vector` pode ter qualquer de seus elementos `deletados`, como numa lista encadeada, e `possui índices`, como num vetor.

Alguns métodos de `Vector<T>`:

```
operador[i] : acessa a posição i;  
at(i)       : o mesmo que usar [];  
size()      : retorna o número de elementos;  
resize(x)   : modifica o tamanho alocado para x unidades;  
capacity()  : informa o valor alocado;  
empty()     : retorna true se o Vector estiver vazio;  
begin()     : retorna um iterador para a primeira posição;  
end()       : retorna um iterador para a última posição;  
front()     : acessa o primeiro elemento;  
back()      : acessa o último elemento;  
erase(it)   : remove elemento na posição do iterador.  
push_back(T) : adiciona o elemento T no final;  
pop_back()  : remove o último elemento;  
insert(it,T) : adiciona um elemento T dado um iterador;  
clear()     : apaga todos os elementos.
```



## Construtores:

```
vector<int> v1;
```

É criado um vector vazio;

```
vector<int> v2(3,5);
```

É criado um vector de tamanho inicial 3, cujas posições têm valor 5;

```
vector<int> v3(v2.begin(),v2.end());
```

Os elementos de v2 são copiados a partir do iterador;

```
vector<int> v4(v3);
```

v4 copia o conteúdo de v3.

```
int vetor[] = { 4 , 8 , 90 , 15, 87 };
```

```
vector<int> v5(vetor,vetor + sizeof(vetor)/sizeof(int));
```

Copia elementos de um vetor.



```
// Vector funcionando como uma pilha.

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

int main (void)
{
    vector<int> v;
    // Elementos adicionados na seguinte ordem:
    // 1, 3, 9, 45, 33, 27
    v.push_back(1);
    v.push_back(3);
    v.push_back(9);
    v.push_back(45);
    v.push_back(33);
    v.push_back(27);

    while(!v.empty())
    {
        cout << v.back() << " ";
        v.pop_back();
    }
    return 0;
}
```



```

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

int main(void)
{
    vector<string> v;
    vector<string>::iterator it;
    cout << "Qtd Elementos: " << v.size() << endl;
    // Os elementos são inseridos sempre em v.begin(),
    // ou seja, na primeira posição do vector.
    v.insert(v.begin(), "A");
    v.insert(v.begin(), "B");
    v.insert(v.begin(), "C");
    v.insert(v.begin(), "D");
    // O "A" é inserido primeiro, mas ao fim, ele está
    // na quarta posição.
    for(it = v.begin() ; it < v.end() ; it++) {
        cout << *it << " ";
    }

    cout << endl << "Qtd Elementos: " << v.size() << endl;

    return 0;
}

```



```

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

int main(void)
{
    vector<int> k;

    k.push_back(1);
    cout << "Com " << k.size();
    cout << " elemento, a capacidade alocada eh ";
    cout << k.capacity() << endl;

    k.push_back(5);
    cout << "Com " << k.size();
    cout << " elemento, a capacidade alocada eh ";
    cout << k.capacity() << endl;

    k.push_back(7);
    cout << "Com " << k.size();
    cout << " elemento, a capacidade alocada eh ";
    cout << k.capacity() << endl;

    k.push_back(9);
    k.push_back(3);
    cout << "Com " << k.size();
    cout << " elemento, a capacidade alocada eh ";
    cout << k.capacity() << endl;

    return 0;
}

```

**capacity()** é o espaço alocado para o vector. Sempre que este espaço estoura, a capacidade é dobrada.

Esta dobra frequente pode tornar seu programa lento em caso de muitas inserções.

Para que isto não aconteça, use **resize()** sempre que o tamanho máximo desejado for conhecido com antecedência.



## // Exemplo de uso dos métodos erase() e clear()

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

int main(void)
{
    // Vector criado com 6 elementos.
    vector<char> v;

    v.push_back('a');
    v.push_back('b');
    v.push_back('c');
    v.push_back('d');
    v.push_back('e');
    v.push_back('f');

    // Apaga os dois primeiros elementos:
    v.erase(v.begin(), v.begin() + 2);

    // Apaga o último elemento:
    v.erase(v.end() - 1);

    // Vector não precisa de iteradores para listagem:
    for(unsigned int i = 0 ; i < v.size() ; i++)
    {
        cout << v[i] << " ";
    }

    // Todos os elementos apagados:
    v.clear();
    cout << endl << "v.size(): " << v.size() << endl;
    return 0;
}
```



```

// A função sort() ordena o conteúdo de um Vector

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

bool Ordena (int a, int b) {
    return a > b;
}

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

    v.push_back(5);
    v.push_back(2);
    v.push_back(7);
    v.push_back(3);
    v.push_back(2);
    v.push_back(1);

    // Parâmetros: Iterador para o início,
    // Iterador para o fim, e Método de Ordenacao, opcional.
    sort(v.begin(), v.end(), Ordena);

    for(register int i = 0 ; i < v.size() ; ++i) {
        cout << v[i] << " ";
    }

    return 0;
}

```



## Deque

“Deque” significa “double-ended queue”, “fila com dois fins”. Na prática é como um vector, mas permite adição e remoção não só no fim, mas também no **início**.

Assim como o Vector, o deque é diferente de uma lista encadeada comum por possuir o método `at()` e o operador `[]`.

Em relação aos métodos, as únicas diferenças entre Deque e Vector são:

- deque **não possui** os métodos `capacity()` e um outro que não citei, `reserve()`;
- deque **possui** os métodos `push_front(T)` e `pop_front()`, que adicionam e retiram elementos do início. O `push_front()` foi simulado no slide 18.



## Deque usado como pilha:

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

int main(void)
{
    deque<char> d;
    d.push_back('a');
    d.push_back('b');
    d.push_back('c');
    d.push_back('d');

    while(!d.empty())
    {
        cout << d.back() << " ";
        d.pop_back();
    }
    return 0;
}
```

## Deque usado como fila:

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

int main(void)
{
    deque<char> d;
    d.push_back('a');
    d.push_back('b');
    d.push_back('c');
    d.push_back('d');

    while(!d.empty())
    {
        cout << d.front() << " ";
        d.pop_front();
    }
    return 0;
}
```

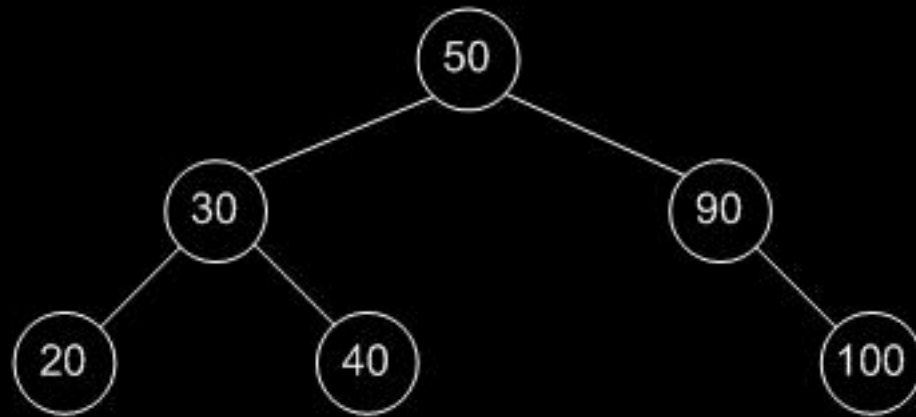


## Set (Conjunto)

Sets são como conjuntos comuns: ***não aceitam elementos repetidos.***

Diferente de vetores, deques e vectors, elementos em um Set não podem ser acessados por índices, por não existir uma ordem pré definida de seus elementos. Internamente, os elementos não são sequenciais.

O tipo Set é uma **árvore binária balanceada**, estrutura de dados bastante otimizada para busca de informações.



O tipo `Set<T>` possui os seguintes métodos:

```
insert(T) : adiciona elementos, e ignora se for repetido;
erase(T)  : apaga um elemento;
erase(it) : apaga uma posição de um iterador;
clear()   : apaga todos os elementos;
find(T)   : retorna um iterador com a posição de T;
count(T)  : retorna true se o elemento T existir no Set;
empty()   : retorna true se o Set estiver vazio;
size()    : informa a quantidade de elementos;
begin()   : retorna um iterador para a primeira posição.
end()     : retorna um iterador para a última posição.
```

- Como o tipo `Set` não pode ter seus elementos acessados por índices, a única forma de percorrer os elementos é com um objeto do tipo *iterador*.

- O `set`, por padrão, organiza os elementos do menor para o maior. Mas o programador tem liberdade para mudar o tipo de comparação, como será visto no próximo exemplo.



```

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

// Neste exemplo, está sendo criado um Set de pessoas. Elas são
// ordenadas da maior para a menor idade, e são listadas pelo nome.

class Pessoa {
    friend ostream& operator<<(ostream& o, Pessoa p);

private:
    string _nome;
    int _idade;
public:
    string &nome(void) { return _nome; }
    int &idade(void) { return _idade; }
    Pessoa(string nome, int idade) : _nome(nome) , _idade(idade) {}
};

class Comparador {
public:
    bool operator() (Pessoa a, Pessoa b)
    {
        return a.idade() > b.idade();
    }
};

ostream& operator<<(ostream& o, Pessoa p) {
    o << p._nome;
    return o;
}

int main(void) {
    set<Pessoa,Comparador> s;
    s.insert(Pessoa("Maria",30));
    s.insert(Pessoa("Joao",10));
    s.insert(Pessoa("Helena",15));

    set<Pessoa>::iterator it;
    for(it = s.begin() ; it != s.end() ; it++) {
        cout << *it << endl;
    }
    return 0;
}

```



```

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

// Exemplo um pouco mais simples do que o anterior:

int main(void)
{
    set<int> s;

    s.insert(0);
    s.insert(8);
    s.insert(4);
    s.insert(2);
    s.insert(7);
    s.insert(5);

    s.erase(2);
    s.erase(s.find(4));

    cout << "Qtd elementos: " << s.size() << endl;

    set<int>::iterator it;
    for(it = s.begin() ; it != s.end() ; it++) {
        cout << *it << endl;
    }
    return 0;
}

```



- Uma classe é considerada `comparadora` desde que contenha uma redefinição do `operador parêntesis` que retorne `bool`.
- Para que um `iterador` referente a um tipo arbitrário possa ser impresso na tela, o operador `<<` deve ser redefinido na classe deste tipo. O mesmo deve ser feito para qualquer operador que não esteja definido na classe mas venha a ser usado.
- O modificador `friend` permite que um atributo `private` de uma classe seja modificável em outra classe ou função.
- O comparador pode também ser uma `struct`.

## Map

Um Map é uma estrutura que relaciona uma **chave** a um **conteúdo/valor**, como num vetor comum.

Vetores são acessados assim:

```
Vet[2] = 9.5;
```

Ou seja, no formato:

```
Vet[Chave] = Valor;
```

Os índices de um vetor sempre são números inteiros positivos, e neste exemplo o vetor é de floats. Vetores sempre são uma relação entre **unsigned ints** (chave) e algum outro tipo arbitrário (conteúdo).

Vetores não podem ter mais de um valor para cada índice, mas um mesmo valor pode se repetir para índices diferentes. Em outras palavras: chaves não se repetem, mas os valores relacionados a ela podem se repetir livremente.

```
Vetor[2] = 9.5;
```

```
Vetor[3] = 9.5;
```

Assim como vetores, Maps também possuem índices e valores, mas os índices não precisam ser ints. Este é um exemplo de Map que relaciona strings com strings:

```
Vetor["Um"] = "One";
```



Este Map existe

'a'	45
'%'	72
	⋮
'&'	72

Este Map não existe

'a'	45
'a'	72
	⋮
'&'	38

Apesar da semelhança entre Vetores e Maps, estes não são sequenciais. Sua estrutura interna, assim como no Set, é uma **árvore binária balanceada**. Podemos acessar uma posição de um Map por um índice, caso este índice seja conhecido. Mas para listar todos os elementos, é necessário usar um iterador.

Comparadores também podem ser definidos nos Maps.



O tipo `Map<K,V>` possui os seguintes métodos:

```
operador[] : usado para adicionar elementos;
erase(K)    : apaga um elemento, dada uma chave;
erase(it)   : apaga um elemento, dado um iterador com a chave;
clear()     : apaga todos os elementos;
find(K)     : retorna um iterador com a posição de K;
count(K)    : retorna true se o elemento K existir no Map;
empty()     : retorna true se o Map estiver vazio;
size()      : informa a quantidade de elementos;
begin()     : retorna um iterador para a primeira posição.
end()       : retorna um iterador para a última posição.
```

O iterador para Map possui dois parâmetros importantes:

```
i->first : Chave na posição específica.
i->second : Valor na posição específica.
```

- Assim como no Set, a ordem natural dos elementos é do menor para o maior, e a ordem alfabética.



```

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

// Exemplo de dicionário Português-Inglês usando Map

struct Comparador {
    bool operator()(string a, string b) {
        return a > b;
    }
};

int main(void) {
    map<string, string, Comparador> Dicionario;

    Dicionario["Mesa"]    = "Table";
    Dicionario["Tabela"] = "Table";
    Dicionario["Carro"]  = "Car";
    Dicionario["Arvore"] = "Tree";
    Dicionario["Azul"]   = "Blue";
    Dicionario["Banana"] = "Banana";

    map<string, string>::iterator i;

    cout << "Lista de palavras que significam 'Table' em ingles:" << endl;
    for(i = Dicionario.begin() ; i != Dicionario.end() ; ++i) {
        if(i->second == "Table") {
            cout << i->first << endl;
        }
    }

    cout << endl << "Lista de palavras que sao iguais em ambos os idiomas:" << endl;
    for(i = Dicionario.begin() ; i != Dicionario.end() ; ++i) {
        if(i->first == i->second) {
            cout << i->first << endl;
        }
    }
    return 0;
}

```



## Ellipsis (Reticências)

As funções `scanf` e `printf` podem receber um número indefinido de parâmetros. Você havia notado?

```
printf( "%d" , a );  
printf( "%d %d" , a , b );  
printf( "%d %d %d" , a , b , c );
```

Provavelmente não, mas agora notou. E deve estar se perguntando como isso é feito, e como repetir o mesmo em seu código. As funções `scanf` e `printf` são implementadas com uma ferramenta de C chamada **Ellipsis**.

O cabeçalho da função deve ter apenas um parâmetro, e este parâmetro deve ser seguido por reticências:

```
void funcao(int c,...) {}
```

Os infinitos parâmetros que você passar serão listados sequencialmente na memória. Ou seja, são facilmente acessíveis via ponteiro.

Veja um exemplo bastante simples no próximo slide.



```

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

// Esta função supõe que o primeiro parâmetro
// é a quantidade de valores. scanf e printf fazem o mesmo,
// contando a quantidade de ocorrências de '%' na frase.
void imprime(int c,...)
{
    cout << "Imprimindo..." << endl;
    int *p = &c;

    for(register int i = 1 ; i <= c ; ++i)
    {
        cout << p[i] << " ";
    }
    cout << endl;
}

int main(void)
{
    imprime(3,1,3,5);
    imprime(5,2,4,6,8,10);

    return 0;
}

```



Agora que você já matou a curiosidade, saiba que **Ellipsis** devem ser **evitadas**.

No exemplo anterior eu supus que os parâmetros seriam **apenas ints**. Mas e se for passado um **char**, ou um objeto qualquer que seja bastante complexo? Não existe checagem de tipos; esta deve ser feita pelo próprio programador.

Este problema ocorre com as funções `printf` e `scanf`, e geram warnings e erros quando os parâmetros não batem com o que deveriam. Sua função precisaria tratar os mesmos casos.

Para exemplos pessoais e pequenos, as reticências podem ajudar, mas em projetos grandes, podem causar bugs complexos e difíceis de serem encontrados.



THE #1 PROGRAMMER EXCUSE  
FOR LEGITIMATELY SLACKING OFF:

"MY CODE'S COMPILING."

