Material de Apoio 2

C++ Reference

#include <vector>

```
Using
namespace
std;

                int main()
                {
                        vector<int>  v;  //  Declaração  de  um  vector  do  tipo  int
                inicialmente vazio

                        v.push_back(1); // Insiro o elemento 1 no final do vetor
                        v.push_back(3); // Insiro o elemento 3 no final do vetor
                        v.push_back(5); // Insiro o elemento 5 no final do vetor
                        v.push_back(7); // Insiro o elemento 7 no final do vetor

                        // O vetor final é {1, 3, 5, 7}
                        return 0;
                }
```

```
#include      <vector>      //
Biblioteca    que    contém    a
estrutura vector
                                using namespace std;

                                int main()
                                {
```

```
	vector<int> v; // Declaração de um vector do tipo int inicialmente vazio

	v.push_back(1); // Insiro o elemento 1 no final do vetor
	v.push_back(3); // Insiro o elemento 3 no final do vetor
	v.push_back(5); // Insiro o elemento 5 no final do vetor
	v.push_back(7); // Insiro o elemento 7 no final do vetor
	// v = {1, 3, 5, 7}

	for(int i=0; i<(int)v.size(); i++) // Percorre o vetor da posição 0 até a última posição (v.size()-1)
	{
		printf("%d ", v[i]); // Imprimo o i-ésimo elemento do vetor
	}
	printf("\n");
	// Saída:
	// 1 3 5 7

	return 0;
}
```

Comandos:

popback(): Remove o último elemento do vetor (Complexidade O(n))

erase(): Remove os elementos de uma determinada posição ou intervalo do vector (Complexidade O(n)):

```
// v = {1, 9, 5, 3, 7}
v.erase(v.begin()+3); // Apaga o elemento da posição v.begin()+3 = 0+3 = 3, ou seja, apaga o elemento v[3] = 3
// v = {1, 9, 5, 7}
```

```
// v = {1, 9, 5, 7}
v.erase(v.begin()+1,v.begin()+3);  // Apaga os elementos do intervalo da posição 1 à 2, ou seja, apaga o 9 e o 5
// v = {1, 7}
```

clear(): Remove todos os elementos do vector (Complexidade O(n))

```
// v = {1, 9, 5, 3, 7}
v.clear(); // Remove todos os elementos do vetor
// v = {}
```

resize(n): Troca o tamanho do vetor e pode ou não inserir um determinado elemento nas posições (Complexidade O(n)):

```
// v = {1, 2, 3, 4, 5}
v.resize(8); // Muda o tamanho do vector v para 8
// v = {1, 2, 3, 4, 5, 0, 0, 0}

// v = {1, 2, 3, 4, 5, 0, 0, 0}
v.resize(12,-1);  // Muda o tamanho do vector v para 12 e nas posições novas/vazias insere o número -1
// v = {1, 2, 3, 4, 5, 0, 0, 0, -1, -1, -1, -1}
```

Ordenando um vector usando sort:

Para ordenar um **vector** utilizando o sort, podemos fazer o seguinte código:

Complexidade: $O(n \log n)$.

```cpp
#include <vector> // Biblioteca que contém a estrutura vector
#include <algorithm> // Biblioteca do sort
using namespace std;

int main()
{
        vector<int> v; // Declaração de um vector do tipo int

        v.push_back(3); // Insiro o elemento 3 no final do vetor
        v.push_back(1); // Insiro o elemento 1 no final do vetor
        v.push_back(7); // Insiro o elemento 7 no final do vetor
        v.push_back(5); // Insiro o elemento 5 no final do vetor

        // v = {3, 1, 7, 5}
        sort(v.begin(), v.end());
        // v = {1, 3, 5, 7}
        return 0;
}
```

Para ordenar um vector de maneira decrescente:

```cpp
bool compararDecrescente(int a, int b) {
    return a > b;
}

int main() {
    std::vector<int> vetor = {4, 2, 8, 1, 6};

    // Usar a função sort com o comparador personalizado
    std::sort(vetor.begin(), vetor.end(), compararDecrescente);

    // Imprimir o vetor ordenado em ordem decrescente
    for (int num : vetor) {
        std::cout << num << " ";
    }
    std::cout << std::endl;

    return 0;
}
```

Ordenar um vector por outros parâmetros, como por exemplo, o tamanho da palavra:

vector<string> words = {"xbc","pcxbcf","xb","cxbc","pcxbc"};

sort(words.begin(), words.end(), [](const string &a, const string &b) {
return a.length() < b.length();
});

Pair:

Pair

```
#include <iostream> // Necessita da biblioteca iostream para funcionar

using namespace std;

int main()
{
	pair< int, int> p1; // Declaração de um pair do tipo (int,int) ex: (100, 250)
	pair< int, string> p2; // Declaração de um pair do tipo (int,string) ex: (10,JOAO)

	return 0;
}
```

```
#include <iostream> // Necessita da biblioteca iostream para funcionar
```

```cpp
#include <vector> // Necessita da biblioteca vector para funcionar
using namespace std;

int main()
{
	vector< pair<int, int> > v = { {10, 1}, {1, 3}, {1, 2}, {2, 4}, {3, 5} }; // Declaração de um vector de pair (int,int)
	// v = { (10, 1), (1, 3), (1, 2), (2, 4), (3, 5) }

	v.push_back({7,8}); // Insiro o par (7,8) ao final do vetor
	// v = { (10, 1), (1, 3), (1, 2), (2, 4), (3, 5), (7,8)}

	printf("%d %d\n", v[4].first, v[4].second); // Imprimo os elementos do par na posição 4, ou seja, 3 e 5 respectivamente
	return 0;
}
```

```cpp
#include <iostream> // Necessita da biblioteca iostream para funcionar
#include <vector> // Necessita da biblioteca vector para funcionar
using namespace std;
```

```
int main()
{
vector<pair< pair<int, string>, int> > v; // Declaração de um vector de pair ((int,string),int)

v.push_back({{1,"ANA"},10}); // Insiro o par ((1,ANA),10) no vetor
v.push_back({{2,"BRUNO"},4}); // Insiro o par ((2,BRUNO),4) no vetor
v.push_back({{3,"CAIO"},7}); // Insiro o par ((3,CAIO),7) no vetor

for(int i=0; i<(int)v.size(); i++) // Percorro o vetor do ínicio ao fim
{
printf("Numero: %d, Nome: %s, Nota: %d\n", v[i].first.first, v[i].first.second.c_str(), v[i].second);
// Imprimo o primeiro elemento do primeiro elemento do pair (número de chamada) = v[i].first.first
// Imprimo o segundo elemento do primeiro elemento do pair (nome do aluno) = v[i].first.second
// Imprimo o único elemento do segundo elemento do pair = v[i].second
}
return 0;
}
```

```cpp
#include <iostream> // Necessita da biblioteca iostream para funcionar
using namepace std;

int main()
{
	pair<int, string> A = {10, "Thiago"};
	pair<int, string> B = {20, "Lawrence"};
	pair<int, string> C = {20, "Davi"};

	if(A < B) {
		// A é menor do que B pois 10 é menor do que 20. (Primeiro ele compara o primeiro valor).
	}
	if(C < B){
		// C é menor do que B nesse caso, pois como o primeiro valor é igual, ele compara pelo segundo. Como 'D' vem antes de 'L', C < B
	}
	return 0;
}
```

```cpp
#include <iostream> // Necessita da biblioteca iostream para funcionar

#include <vector> // Necessita da biblioteca vector para funcionar
#include <algorithm> // Necessita da biblioteca algorithm para funcionar
```

```
using namespace std;

int main()
{
	vector< pair<int, int> > v = {
{10, 1}, {1, 3}, {1, 2}, {2, 4}, {3, 5}
}; // Declaração de um vector de pair
	// v = { (10, 1), (1, 3), (1, 2),
(2, 4), (3, 5) }

	sort(v.begin(),	v.end());	//
Ordena o vector

	// v = { (1, 2), (1, 3), (2, 4),
(3, 5), (10, 1) }
	return 0;
}
```

Ordenar vector de Pairs ignorando o primário:

```
struct ValueComparator {
	bool operator()(const pair<string, pair<int, pair<int, int>>>& a, const
pair<string, pair<int, pair<int, int>>>& b) {
    return a.second > b.second;
  }
};

sort(vector.begin(), vector.end(), ValueComparator());
```

Para acessarmos o primeiro elemento do pair, utilizamos o comando first. De forma similar, o segundo elemento é acessado por meio do comando second. É bastante comum utilizarmos o tipo pair junto com um vector, tornando-o um vetor de pares de valores. Por exemplo, v = {(1,3),(2,5),(9,4)}. Neste vector de pairs, temos que v[1].first é igual a 2 e v[1].second é igual a 5.

Member functions

(constructor)	constructs new pair (public member function)
operator=	assigns the contents (public member function)
swap (C++11)	swaps the contents (public member function)

Non-member functions

make_pair	creates a pair object of type, defined by the argument types (function template)
operator== operator!= (removed in C++20) operator< (removed in C++20) operator<= (removed in C++20) operator> (removed in C++20) operator>= (removed in C++20) operator<=> (C++20)	lexicographically compares the values in the pair (function template)
std::swap(std::pair) (C++11)	specializes the std::swap algorithm (function template)
std::get(std::pair) (C++11)	accesses an element of a pair (function template)

Helper classes

std::tuple_size<std::pair> (C++11)	obtains the size of a pair (class template specialization)
std::tuple_element<std::pair> (C++11)	obtains the type of the elements of pair (class template specialization)
std::basic_common_reference<std::pair> (C++23)	determines the common reference type of two pairs (class template specialization)
std::common_type<std::pair> (C++23)	determines the common type of two pairs (class template specialization)

#include <stack>

```
#include <stack> //
Biblioteca da stack
                        using namespace std;


                        int main()
                        {
                                stack<int> pilha; // Declaro a stack do tipo int


                                pilha.push(1); // Insiro o elemento 1 no topo da
                        pilha
```

```
        pilha.push(3); // Insiro o elemento 3 no topo da
pilha
        pilha.push(7); // Insiro o elemento 7 no topo da
pilha

        printf("%d\n", pilha.top()); // Imprimo o valor 7
(topo da pilha)
        pilha.pop(); // Removo o topo da pilha (7)
        printf("%d\n", pilha.top()); // Imprimo o valor 3
(novo topo da pilha)
        pilha.pop(); // Removo o topo da pilha (3)
        printf("%d\n", pilha.top()); // Imprimo o valor 1
(novo topo da pilha)
        pilha.pop(); // Removo o topo da pilha (1)

        return 0;
}
```

Acesso a todos os elementos

Assim como no vector, a função size() retorna o tamanho atual da pilha. Além disso, existe também a função empty(), que retorna um bool informando se a pilha está vazia ou não. Portanto, podemos iterar por todos os elementos de uma pilha da seguinte forma:

```
#include    <stack>    //
Biblioteca da stack
                        using namespace std;

                        int main()
                        {
```

```
	stack<int> pilha; // Declaro a stack do tipo int

	pilha.push(1); // Insiro o elemento 1 no topo da pilha
	pilha.push(3); // Insiro o elemento 3 no topo da pilha
	pilha.push(7); // Insiro o elemento 7 no topo da pilha
	pilha.push(10); // Insiro o elemento 10 no topo da pilha

	while(!pilha.empty()) // Enquanto a pilha não estiver vazia
	{
		printf("%d ", pilha.top()) // Imprime o elemento no topo da pilha
		pilha.pop(); // Remove o elemento do topo da pilha
	}

	// O algoritmo acima imprime todos os elementos da pilha do topo até o final
	// Saída: 10 7 3 1
	return 0;
}
```

Member types

C++98 C++11

member type	definition	notes
`value_type`	The first template parameter (`T`)	Type of the elements
`container_type`	The second template parameter (`Container`)	Type of the ***underlying container***
`reference`	`container_type::reference`	usually, `value_type&`
`const_reference`	`container_type::const_reference`	usually, `const value_type&`
`size_type`	an unsigned integral type	usually, the same as `size_t`

Member functions

(constructor)	Construct stack (public member function)
empty	Test whether container is empty (public member function)
size	Return size (public member function)
top	Access next element (public member function)
push	Insert element (public member function)
emplace	Construct and insert element (public member function)
pop	Remove top element (public member function)
swap	Swap contents (public member function)

Non-member function overloads

relational operators	Relational operators for stack (function)
swap (stack)	Exchange contents of stacks (public member function)

Non-member class specializations

uses_allocator<stack>	Uses allocator for stack (class template)

#include <queue>

A fila possui duas operações muito importantes: o push() e o pop(). A operação push() insere um elemento no fim da fila e pop() remove o elemento mais à frente. Para acessar o elemento na frente da fila, podemos usar a função front().

```
#include <queue> //
Biblioteca da fila
                                using namespace std;


                                int main()
                                {
```

```
	queue<int> fila; // Declaração de uma queue
do tipo int

	fila.push(1); // Insiro o elemento 1 atrás na
fila
	fila.push(3); // Insiro o elemento 3 atrás na
fila
	fila.push(4); // Insiro o elemento 4 atrás na
fila
	fila.push(9); // Insiro o elemento 9 atrás na
fila

	printf("%d\n", fila.front()); // Imprimo o 1°
valor da fila = 1
	fila.pop(); // Removo o 1
	printf("%d\n", fila.front()); // Imprimo o 1°
valor da fila = 3
	fila.pop(); // Removo o 3
	printf("%d\n", fila.front()); // Imprimo o 1°
valor da fila = 4
	fila.pop(); // Removo o 4
	printf("%d\n", fila.front()); // Imprimo o 1°
valor da fila = 9
	fila.pop(); // Removo o 9

	// Fila vazia
	return 0;
}
```

Assim como na pilha, podemos usar as funções size() e empty() para imprimir todos os elementos presentes em uma fila. Porém, lembre-se que este processo apaga todos os elementos da fila/pilha.

```
#include <queue> //
Biblioteca da fila
                        using namespace std;

                        int main()
                        {
                               queue<int> fila; // Declaração de uma queue do
                        tipo int

                               fila.push(1); // Insiro o elemento 1 atrás na
                        fila
                               fila.push(3); // Insiro o elemento 3 atrás na
                        fila
                               fila.push(4); // Insiro o elemento 4 atrás na
                        fila
                               fila.push(9); // Insiro o elemento 9 atrás na
                        fila

                               while(!fila.empty()) // Enquanto a fila não
                        estiver vazia
                               {
                                      printf("%d ", fila.front()); // Imprimo
                        o 1° elemento da fila
                                      fila.pop(); // Apago o 1° elemento da
                        fila
                               }
                               return 0;
                        }
```

fx Member functions

(constructor)	Construct queue (public member function)
empty	Test whether container is empty (public member function)
size	Return size (public member function)
front	Access next element (public member function)
back	Access last element (public member function)
push	Insert element (public member function)
emplace	Construct and insert element (public member function)
pop	Remove next element (public member function)
swap	Swap contents (public member function)

fx Non-member function overloads

relational operators	Relational operators for queue (function)
swap (queue)	Exchange contents of queues (public member function)

fx Non-member class specializations

uses_allocator<queue>	Uses allocator for queue (class template)

#include <set>

Set (ou conjunto) é uma estrutura de dados da STL, na biblioteca <set>, que guarda elementos distintos de maneira ordenada. Ele tem como membros funções como insert(), erase() e find(), que possuem complexidade O(log n) onde n é o tamanho do conjunto. É uma estrutura muito útil, uma vez que, já implementada no C++, é muito simples de ser utilizada e torna alguns algoritmos muito mais eficientes.

```
#include <set> //
Biblioteca do set
```

```
using namespace std;

int main()
{
set<int> t; // Declaração de um set que armazena
valores inteiros
}
```

```
#include <set> //
Biblioteca do set
using namespace std;

int main()
{
set<int> t; // Declaração de um set de int

t.insert(5); // Adiciona o elemento 5 no set
t.insert(12); // Adiciona o elemento 12 no set
// t = {5, 12}

t.insert(3); //Adiciona o elemento 3 no set
// t = {3, 5, 12} (set sempre é ordenado)

t.insert(5); // Não adiciona o elemento 5 no set porque
ele já está presente
// t = {3, 5, 12}
}
```

```
#include <set> // Biblioteca do
set
using namespace std;
```

```
int main()
{
set<int> t; // Declaração de um set de int

t.insert(5); // Adiciona o elemento 5 no set
t.insert(12); // Adiciona o elemento 12 no
set
t.insert(3); // Adiciona o elemento 3 no set
// t = {3, 5, 12}

t.erase(5); //remove o elemento 5 do set
// t = {3, 12}
}
```

```
#include <iostream> // Bibiloteca
iostream necessária para funcionar
#include <set> // Biblioteca do set
using namespace std;

int main()
{
set<int> t; // Declaração de um set de
int

t.insert(5); // Adiciona o elemento 5
no set
t.insert(12); // Adiciona o elemento
12 no set
t.insert(3); // Adiciona o elemento 3
no set
// t = {3, 5, 12}
```

```
if(t.find(5) != t.end()) // Verifico
se o 5 está no meu set
{
printf("5\n"); // Se estiver, imprimo
o valor
}
}
```

A função find() verifica se um elemento está inserido no set, retornando um ponteiro que aponta para esse elemento. Caso ele não esteja no conjunto, o ponteiro aponta para o fim do set (definido pela função end()). Como as outras funções, também possui complexidade O(logn)

```
#include <iostream> // Bibiloteca
iostream necessária para funcionar
```

```
#include <set> // Biblioteca do set
using namespace std;

int main()
{
set<int> t; // Declaração de um set de
int

t.insert(5); // Adiciona o elemento 5
no set
t.insert(12); // Adiciona o elemento
12 no set
t.insert(3); // Adiciona o elemento 3
no set
// t = {3, 5, 12}
```

```
if(t.find(5) != t.end()) // Verifico
        se o 5 está no meu set
                {
printf("5\n"); // Se estiver, imprimo
                o valor
                }
                }
```

Outras funções

A lista abaixo possui outras funções muito utilizadas:

begin(): retorna um ponteiro que aponta para o inicio do set, em $O(1)$.
end(): retorna um ponteiro que aponta para o fim do set, em $O(1)$.
empty(): retorna um valor booleano, que indica se o set está ou não vazio, em $O(1)$.
size(): retorna quantidade de elementos do set, em $O(1)$.
clear(): remove todos os elementos do set, em $O(n)$.
lower_bound(x): retorna um ponteiro que aponta para o primeiro valor que não vem antes que x, em $O(log\ n)$.
upper_bound(x): retorna um ponteiro que aponta para o primeiro valor que vem depois de x, em $O(log\ n)$.

Member functions

(constructor)	constructs the set (public member function)
(destructor)	destructs the set (public member function)
operator=	assigns values to the container (public member function)
get_allocator	returns the associated allocator (public member function)

Iterators

begin cbegin (C++11)	returns an iterator to the beginning (public member function)
end cend (C++11)	returns an iterator to the end (public member function)
rbegin crbegin (C++11)	returns a reverse iterator to the beginning (public member function)
rend crend (C++11)	returns a reverse iterator to the end (public member function)

Capacity

empty	checks whether the container is empty (public member function)
size	returns the number of elements (public member function)
max_size	returns the maximum possible number of elements (public member function)

Modifiers

`clear`	clears the contents (public member function)
`insert`	inserts elements or nodes (since C++17) (public member function)
`insert_range` (C++23)	inserts a range of elements (public member function)
`emplace` (C++11)	constructs element in-place (public member function)
`emplace_hint` (C++11)	constructs elements in-place using a hint (public member function)
`erase`	erases elements (public member function)
`swap`	swaps the contents (public member function)
`extract` (C++17)	extracts nodes from the container (public member function)
`merge` (C++17)	splices nodes from another container (public member function)

Lookup

`count`	returns the number of elements matching specific key (public member function)
`find`	finds element with specific key (public member function)
`contains` (C++20)	checks if the container contains element with specific key (public member function)
`equal_range`	returns range of elements matching a specific key (public member function)
`lower_bound`	returns an iterator to the first element *not less* than the given key (public member function)
`upper_bound`	returns an iterator to the first element *greater* than the given key (public member function)

Observers

key_comp	returns the function that compares keys (public member function)
value_comp	returns the function that compares keys in objects of type value_typ (public member function)

Non-member functions

operator== operator!= (removed in C++20) operator< (removed in C++20) operator<= (removed in C++20) operator> (removed in C++20) operator>= (removed in C++20) operator<=> (C++20)	lexicographically compares the values in the set (function template)
std::swap(std::set)	specializes the std::swap algorithm (function template)
erase_if(std::set) (C++20)	erases all elements satisfying specific criteria (function template)

UNORDERED_SET É A MESMA COISA SEM ORDENAR

#include <map>

Para declarar um map, é preciso usar a biblioteca <map>. Após isso, especificamos o tipo dos elementos usados como chave e o tipo dos valores que serão mapeados para cada chave.

```
#include      <iostream>      //
Biblioteca iostream necessária
para funcionar
```

```
#include <map> // Biblioteca do map
using namespace std;

int main()
{
        map<string, int> mapa1; // Declaração de
map mapa1 que possui strings como chaves e ints
como valores
```

```
	map<int, double> mapa2; // Declaração de map mapa2 que possui doubles como chaves e ints como valores

	return 0;
}
```

Há duas maneiras de se inserir um par {chave, valor} no map. A primeira utiliza uma ideia semelhante ao pair e a segunda uma ideia semelhante à atribuição de um valor a uma determinada posição de um vetor. Em ambas as formas, a complexidade da inserção é de O(log(n))

```
#include <iostream> // Biblioteca iostream necessária para funcionar
#include <map> // Biblioteca do map
using namespace std;

int main()
{
	map<string, int> mapa1; // Declaração do map mapa1 que possui strings como chaves e ints como valores

	// 1° método - usamos o comando insert e inserimos o par {chave, valor}
	mapa1.insert(make_pair("Ana", 5));

	// 2° método - usamos a atribuição de valor como em vetores
	mapa1["Ana"] = 5;

	return 0;
}
```

Para saber se uma certa chave foi inserida no map, podemos usar o comando find. Caso a chave não tenha sido inserida, esta função retorna um ponteiro para o final do map, que pode ser acessado com o comando end. Novamente, a complexidade desta operação é de O(log(n)):

```
#include <iostream> // Biblioteca iostream necessária para funcionar
#include <map> // Biblioteca do map
using namespace std;

int main()
{
	map<string, int> mapa1; // Declaração do map mapa1 que possui strings como chaves e ints como valores

	mapa1["Lucas"] = 13; // Insiro a chave "Lucas" de valor 13

	if(mapa1.find("Lucas") == mapa1.end()) // Se a chave "Lucas" não foi inserida no mapa
	{
		printf("Chave não encontrada"); // Imprimo "Chave não encontrada"
	}
	else // Senão, imprimo o seu valor
	{
		printf("%d", mapa1["Lucas"]);
	}
}
```

Para remover uma chave do map, basta utilizar o comando erase e especificar a chave desejada. A complexidade de remover uma chave é O(log(n)):

```cpp
#include <iostream> // Biblioteca iostream necessária para funcionar

#include <map> // Biblioteca do map
using namespace std;

int main()
{
	map<string, int> mapa1; // Declaração do map mapa1 que possui strings como chaves e ints como valores

	mapa1["Lucas"] = 13; // Insiro a chave "Lucas" de valor 13

	if(mapa1.find("Lucas") == mapa1.end()) // Se a chave "Lucas" não foi inserida no mapa
	{
		printf("Chave não encontrada\n"); // Imprimo "Chave não encontrada"
	}
	else // Senão, imprimo o seu valor
	{
		printf("%d\n", mapa1["Lucas"]);
	}

	mapa1.erase("Lucas"); // Removo a chave "Lucas" do mapa usando o comando erase

	if(mapa1.find("Lucas") == mapa1.end()) // Se a chave "Lucas" não foi inserida no mapa
	{
		printf("Chave não encontrada\n"); // Imprimo "Chave não encontrada"
	}
```

```
        else // Senão, imprimo o seu valor
        {
                printf("%d\n", mapa1["Lucas"]);
        }
}
```

Ainda é possível percorrer um map, iterando por todas as chaves existentes na estrutura. Para fazer isso, usaremos o mesmo processo para iterar um set: Usaremos um iterator que se localiza na posição inicial do map (obtido pelo comando begin) e irá navegar por todos os elementos no mapa até o seu fim (função end). Vale lembrar que os pares (chave, valor) presentes no map são mantidos de forma ordenada (por padrão, crescentemente) pelo mesmo comparador do tipo pair.

```
#include <iostream> // Biblioteca iostream necessária para funcionar
#include <map> // Biblioteca do map
using namespace std;

int main()
{
        map<string, int> mapa1; // Declaração do map mapa1 que possui strings como chaves e ints como valores

        mapa1["Lucas"] = 13; // Insiro a chave "Lucas" de valor 13
        mapa1["Ana"] = 7; // Insiro a chave "Ana" de valor 7
        mapa1["Thiago"] = 20; // Insiro a chave "Thiago" de valor 20

        map<string, int>::iterator it;
```

```
        for(it=mapa1.begin();       it!=mapa1.end();
it++) // O iterator it irá percorrer o map do seu
ínicio ao fim
        {
                // O iterator it será um par de dois
valores:
                printf("Chave:  %d\n",  (*it).first);
// O seu valor first será a chave na posição atual
do map
                printf("Valor:  %d\n",  (*it).second);
// O seu valor second será o valor mapeado à chave
        }
}
```

Outra forma de percorrer um map:

For (pair<int, int> par : map) {
cout << par.first << " " << par.second << endl;
}

Member classes

`value_compare`	compares objects of type `value_type` (class)

Member functions

(constructor)	constructs the map (public member function)
(destructor)	destructs the map (public member function)
`operator=`	assigns values to the container (public member function)
`get_allocator`	returns the associated allocator (public member function)
Element access	
`at`	access specified element with bounds checking (public member function)
`operator[]`	access or insert specified element (public member function)
Iterators	
`begin` `cbegin` (C++11)	returns an iterator to the beginning (public member function)
`end` `cend` (C++11)	returns an iterator to the end (public member function)
`rbegin` `crbegin` (C++11)	returns a reverse iterator to the beginning (public member function)
`rend` `crend` (C++11)	returns a reverse iterator to the end (public member function)
Capacity	
`empty`	checks whether the container is empty (public member function)
`size`	returns the number of elements (public member function)
`max_size`	returns the maximum possible number of elements (public member function)

insert_range (C++23)	inserts a range of elements (public member function)
insert_or_assign (C++17)	inserts an element or assigns to the current element if the key already exists (public member function)
emplace (C++11)	constructs element in-place (public member function)
emplace_hint (C++11)	constructs elements in-place using a hint (public member function)
try_emplace (C++17)	inserts in-place if the key does not exist, does nothing if the key exists (public member function)
erase	erases elements (public member function)
swap	swaps the contents (public member function)
extract (C++17)	extracts nodes from the container (public member function)
merge (C++17)	splices nodes from another container (public member function)
Lookup	
count	returns the number of elements matching specific key (public member function)
find	finds element with specific key (public member function)
contains (C++20)	checks if the container contains element with specific key (public member function)
equal_range	returns range of elements matching a specific key (public member function)
lower_bound	returns an iterator to the first element *not less* than the given key (public member function)
upper_bound	returns an iterator to the first element *greater* than the given key (public member function)
Observers	
key_comp	returns the function that compares keys (public member function)
value_comp	returns the function that compares keys in objects of type value_type (public member function)

No caso do unordered_map, possui as mesmas funções, porém não é necessariamente ordenado, alterando as complexidades das suas funções.

#include <queue>

A *Priority_Queue* (ou Fila de Prioridade) é uma estrutura de dados bastante útil da STL do C++, e possui muitas semelhanças com a queue. Estra estrutura é armazena uma fila de elementos de tal forma que o elemento na mais à frente é sempre o de maior valor.

```
#include <iostream> //
Biblioteca iostream necessária
para funcionar
```

```
#include <queue> // Biblioteca da queue
using namespace std;
```

```
int main()
{
	priority_queue<int> fila1; // Fila de prioridade fila1 armazenando elementos do tipo int
	priority_queue<double> fila1; // Fila de prioridade fila2 armazenando elementos do tipo double

	return 0;
}
```

Inserção de elementos

A sintaxe para inserir um elemento na priority_queue é a mesma da queue - basta utilizar o comando push(), como indicado no código abaixo:

```
#include <iostream> // Biblioteca iostream necessária para funcionar
#include <queue> // Biblioteca da queue
using namespace std;

int main()
{
	priority_queue<int> fila1; // Fila de prioridade fila1 armazenando elementos do tipo int

	fila1.push(4); // Insiro o 4 na fila
	fila1.push(3); // Insiro o 3 na fila
}
```

Acessar o maior elemento

Como já dito, o elemento de maior valor presente na estrutura é sempre o elemento na frente da fila. Para encontrar o maior elemento, basta usar o comando top().

```
#include <iostream> // Biblioteca
iostream necessária para
funcionar
                                    #include <algorithm> // Biblioteca algorithm
                                    necessária para funcionar
                                    #include <queue> // Biblioteca da queue
                                    using namespace std;

                                    int main()
                                    {
                                           priority_queue<int> fila1; // Fila de
                                    prioridade fila1 armazenando elementos do tipo
                                    int

                                           fila1.push(3); // Insiro o 3 na fila
                                           fila1.push(4); // Insiro o 4 na fila

                                           printf("%d", fila1.top()); // Imprimo o
                                    maior número (topo) = 4
                                    }
```

Remoção de um elemento

Para remover um elemento da frente da fila, o comando pop() é utilizado, da mesma forma que na *queue*. Lembre-se que, diferentemente da queue, o elemento no topo da fila é sempre o elemento de maior valor presente nela.

```cpp
#include <iostream> // Biblioteca iostream necessária para funcionar
#include <algorithm> // Biblioteca algorithm necessária para funcionar
#include <queue> // Biblioteca da queue
using namespace std;

int main()
{
	priority_queue<int> fila1; // Fila de prioridade fila1 armazenando elementos do tipo int

	fila1.push(3); // Insiro o 3 na fila
	fila1.push(4); // Insiro o 4 na fila

	printf("%d", fila1.top()); // Imprimo o maior número (topo) = 4

	fila1.pop(); // Removo o topo (4)

	printf("%d", fila1.top()); // Imprimo o novo maior número (topo) = 3
}
```

Ordenação da fila de maneira crescente

É possível fazer uma alteração na fila para que o elemento do topo seja sempre o menor. Para isso, basta modificar a declaração da priority_queue da seguinte forma:

```
#include <iostream> //
Biblioteca iostream
necessária para funcionar
                        #include <algorithm> // Biblioteca algorithm
                        necessária para funcionar
                        #include <queue> // Biblioteca da queue
                        using namespace std;

                        int main()
                        {
                               priority_queue<int, vector<int>, greater<int>>
                        fila1; // Fila de prioridade fila1 armazenando
                        elementos do tipo int
                               // Agora menor elemento sempre no topo

                               fila1.push(3); // Insiro o 3 na fila
                               fila1.push(4); // Insiro o 4 na fila

                               printf("%d", fila1.top()); // Imprimo o menor
                        número (topo) = 3

                               priority_queue<double, vector<double>,
                        greater<double>> fila2; // Fila de prioridade fila1
                        armazenando elementos do tipo double
                               // Agora menor elemento sempre no topo

                               fila2.push(5.2); // Insiro o 5.2 na fila
                               fila2.push(3.14); // Insiro o 3.14 na fila

                               printf("%d", fila2.top()); // Imprimo o menor
                        número (topo) = 3.14
                        }
```

Member functions

(constructor)	Construct priority queue (public member function)
empty	Test whether container is empty (public member function)
size	Return size (public member function)
top	Access top element (public member function)
push	Insert element (public member function)
emplace	Construct and insert element (public member function)
pop	Remove top element (public member function)
swap	Swap contents (public member function)

Non-member function overloads

swap (queue)	Exchange contents of priority queues (public member function)

Non-member class specializations

uses_allocator<queue>	Uses allocator for priority queue (class template)

#include <algorithm>

Non-modifying sequence operations

Defined in header <algorithm>

all_of (C++11) any_of (C++11) none_of (C++11)	checks if a predicate is `true` for all, any or none of the elements in a range (function template)
ranges::all_of (C++20) ranges::any_of (C++20) ranges::none_of (C++20)	checks if a predicate is `true` for all, any or none of the elements in a range (niebloid)
for_each	applies a function to a range of elements (function template)
ranges::for_each (C++20)	applies a function to a range of elements (niebloid)
for_each_n (C++17)	applies a function object to the first n elements of a sequence (function template)
ranges::for_each_n (C++20)	applies a function object to the first n elements of a sequence (niebloid)
count count_if	returns the number of elements satisfying specific criteria (function template)
ranges::count (C++20) ranges::count_if (C++20)	returns the number of elements satisfying specific criteria (niebloid)
mismatch	finds the first position where two ranges differ (function template)
ranges::mismatch (C++20)	finds the first position where two ranges differ (niebloid)
find find_if find_if_not (C++11)	finds the first element satisfying specific criteria (function template)
ranges::find (C++20) ranges::find_if (C++20) ranges::find_if_not (C++20)	finds the first element satisfying specific criteria (niebloid)
ranges::find_last (C++23) ranges::find_last_if (C++23) ranges::find_last_if_not (C++23)	finds the last element satisfying specific criteria (niebloid)
find_end	finds the last sequence of elements in a certain range (function template)
ranges::find_end (C++20)	finds the last sequence of elements in a certain range (niebloid)
find_first_of	searches for any one of a set of elements (function template)
ranges::find_first_of (C++20)	searches for any one of a set of elements (niebloid)

`ranges::find_first_of` (C++20)	searches for any one of a set of elements (niebloid)
`adjacent_find`	finds the first two adjacent items that are equal (or satisfy a given predicate) (function template)
`ranges::adjacent_find` (C++20)	finds the first two adjacent items that are equal (or satisfy a given predicate) (niebloid)
`search`	searches for a range of elements (function template)
`ranges::search` (C++20)	searches for a range of elements (niebloid)
`search_n`	searches a range for a number of consecutive copies of an element (function template)
`ranges::search_n` (C++20)	searches for a number consecutive copies of an element in a range (niebloid)
`ranges::starts_with` (C++23)	checks whether a range starts with another range (niebloid)
`ranges::ends_with` (C++23)	checks whether a range ends with another range (niebloid)

Modifying sequence operations

Defined in header <algorithm>

`copy` `copy_if` (C++11)	copies a range of elements to a new location (function template)
`ranges::copy` (C++20) `ranges::copy_if` (C++20)	copies a range of elements to a new location (niebloid)
`copy_n` (C++11)	copies a number of elements to a new location (function template)
`ranges::copy_n` (C++20)	copies a number of elements to a new location (niebloid)
`copy_backward`	copies a range of elements in backwards order (function template)
`ranges::copy_backward` (C++20)	copies a range of elements in backwards order (niebloid)
`move` (C++11)	moves a range of elements to a new location (function template)
`ranges::move` (C++20)	moves a range of elements to a new location (niebloid)
`move_backward` (C++11)	moves a range of elements to a new location in backwards order (function template)
`ranges::move_backward` (C++20)	moves a range of elements to a new location in backwards order (niebloid)
`fill`	copy-assigns the given value to every element in a range (function template)
`ranges::fill` (C++20)	assigns a range of elements a certain value (niebloid)

`fill_n`	copy-assigns the given value to N elements in a range (function template)
`ranges::fill_n` (C++20)	assigns a value to a number of elements (niebloid)
`transform`	applies a function to a range of elements, storing results in a destination range (function template)
`ranges::transform` (C++20)	applies a function to a range of elements (niebloid)
`generate`	assigns the results of successive function calls to every element in a range (function template)
`ranges::generate` (C++20)	saves the result of a function in a range (niebloid)
`generate_n`	assigns the results of successive function calls to N elements in a range (function template)
`ranges::generate_n` (C++20)	saves the result of N applications of a function (niebloid)
`remove` `remove_if`	removes elements satisfying specific criteria (function template)
`ranges::remove` (C++20) `ranges::remove_if` (C++20)	removes elements satisfying specific criteria (niebloid)
`remove_copy` `remove_copy_if`	copies a range of elements omitting those that satisfy specific criteria (function template)
`ranges::remove_copy` (C++20) `ranges::remove_copy_if` (C++20)	copies a range of elements omitting those that satisfy specific criteria (niebloid)
`replace` `replace_if`	replaces all values satisfying specific criteria with another value (function template)
`ranges::replace` (C++20) `ranges::replace_if` (C++20)	replaces all values satisfying specific criteria with another value (niebloid)
`replace_copy` `replace_copy_if`	copies a range, replacing elements satisfying specific criteria with another value (function template)
`ranges::replace_copy` (C++20) `ranges::replace_copy_if` (C++20)	copies a range, replacing elements satisfying specific criteria with another value (niebloid)
`swap`	swaps the values of two objects (function template)
`swap_ranges`	swaps two ranges of elements (function template)
`ranges::swap_ranges` (C++20)	swaps two ranges of elements (niebloid)
`iter_swap`	swaps the elements pointed to by two iterators (function template)

reverse	reverses the order of elements in a range (function template)
ranges::reverse (C++20)	reverses the order of elements in a range (niebloid)
reverse_copy	creates a copy of a range that is reversed (function template)
ranges::reverse_copy (C++20)	creates a copy of a range that is reversed (niebloid)
rotate	rotates the order of elements in a range (function template)
ranges::rotate (C++20)	rotates the order of elements in a range (niebloid)
rotate_copy	copies and rotate a range of elements (function template)
ranges::rotate_copy (C++20)	copies and rotate a range of elements (niebloid)
shift_left shift_right (C++20)	shifts elements in a range (function template)
ranges::shift_left ranges::shift_right (C++23)	shifts elements in a range (niebloid)
random_shuffle (until C++17) shuffle (C++11)	randomly re-orders elements in a range (function template)
ranges::shuffle (C++20)	randomly re-orders elements in a range (niebloid)
sample (C++17)	selects n random elements from a sequence (function template)
ranges::sample (C++20)	selects n random elements from a sequence (niebloid)
unique	removes consecutive duplicate elements in a range (function template)
ranges::unique (C++20)	removes consecutive duplicate elements in a range (niebloid)
unique_copy	creates a copy of some range of elements that contains no consecutive duplicates (function template)
ranges::unique_copy (C++20)	creates a copy of some range of elements that contains no consecutive duplicates (niebloid)

Partitioning operations

Defined in header <algorithm>

is_partitioned (C++11)	determines if the range is partitioned by the given predicate (function template)
ranges::is_partitioned (C++20)	determines if the range is partitioned by the given predicate (niebloid)
partition	divides a range of elements into two groups (function template)
ranges::partition (C++20)	divides a range of elements into two groups (niebloid)
partition_copy (C++11)	copies a range dividing the elements into two groups (function template)
ranges::partition_copy (C++20)	copies a range dividing the elements into two groups (niebloid)
stable_partition	divides elements into two groups while preserving their relative order (function template)
ranges::stable_partition (C++20)	divides elements into two groups while preserving their relative order (niebloid)
partition_point (C++11)	locates the partition point of a partitioned range (function template)
ranges::partition_point (C++20)	locates the partition point of a partitioned range (niebloid)

Sorting operations

Defined in header <algorithm>

is_sorted (C++11)	checks whether a range is sorted into ascending order (function template)
ranges::is_sorted (C++20)	checks whether a range is sorted into ascending order (niebloid)
is_sorted_until (C++11)	finds the largest sorted subrange (function template)
ranges::is_sorted_until (C++20)	finds the largest sorted subrange (niebloid)
sort	sorts a range into ascending order (function template)
ranges::sort (C++20)	sorts a range into ascending order (niebloid)
partial_sort	sorts the first N elements of a range (function template)
ranges::partial_sort (C++20)	sorts the first N elements of a range (niebloid)
partial_sort_copy	copies and partially sorts a range of elements (function template)
ranges::partial_sort_copy (C++20)	copies and partially sorts a range of elements (niebloid)
stable_sort	sorts a range of elements while preserving order between equal elements (function template)

`ranges::stable_sort` (C++20)	sorts a range of elements while preserving order between equal elements (niebloid)
`nth_element`	partially sorts the given range making sure that it is partitioned by the given element (function template)
`ranges::nth_element` (C++20)	partially sorts the given range making sure that it is partitioned by the given element (niebloid)

Binary search operations (on sorted ranges)

Defined in header <algorithm>

`lower_bound`	returns an iterator to the first element *not less* than the given value (function template)
`ranges::lower_bound` (C++20)	returns an iterator to the first element *not less* than the given value (niebloid)
`upper_bound`	returns an iterator to the first element *greater* than a certain value (function template)
`ranges::upper_bound` (C++20)	returns an iterator to the first element *greater* than a certain value (niebloid)
`binary_search`	determines if an element exists in a partially-ordered range (function template)
`ranges::binary_search` (C++20)	determines if an element exists in a partially-ordered range (niebloid)
`equal_range`	returns range of elements matching a specific key (function template)
`ranges::equal_range` (C++20)	returns range of elements matching a specific key (niebloid)

Other operations on sorted ranges

Defined in header <algorithm>

`merge`	merges two sorted ranges (function template)
`ranges::merge` (C++20)	merges two sorted ranges (niebloid)
`inplace_merge`	merges two ordered ranges in-place (function template)
`ranges::inplace_merge` (C++20)	merges two ordered ranges in-place (niebloid)

Set operations (on sorted ranges)

Defined in header <algorithm>

`includes`	returns `true` if one sequence is a subsequence of another (function template)
`ranges::includes` (C++20)	returns `true` if one sequence is a subsequence of another (niebloid)
`set_difference`	computes the difference between two sets (function template)
`ranges::set_difference` (C++20)	computes the difference between two sets (niebloid)
	computes the intersection of two sets

`set_intersection`	computes the intersection of two sets (function template)
`ranges::set_intersection` (C++20)	computes the intersection of two sets (niebloid)
`set_symmetric_difference`	computes the symmetric difference between two sets (function template)
`ranges::set_symmetric_difference` (C++20)	computes the symmetric difference between two sets (niebloid)
`set_union`	computes the union of two sets (function template)
`ranges::set_union` (C++20)	computes the union of two sets (niebloid)

Heap operations

Defined in header <algorithm>

`is_heap` (C++11)	checks if the given range is a max heap (function template)
`ranges::is_heap` (C++20)	checks if the given range is a max heap (niebloid)
`is_heap_until` (C++11)	finds the largest subrange that is a max heap (function template)
`ranges::is_heap_until` (C++20)	finds the largest subrange that is a max heap (niebloid)
`make_heap`	creates a max heap out of a range of elements (function template)
`ranges::make_heap` (C++20)	creates a max heap out of a range of elements (niebloid)
`push_heap`	adds an element to a max heap (function template)
`ranges::push_heap` (C++20)	adds an element to a max heap (niebloid)
`pop_heap`	removes the largest element from a max heap (function template)
`ranges::pop_heap` (C++20)	removes the largest element from a max heap (niebloid)
`sort_heap`	turns a max heap into a range of elements sorted in ascending order (function template)
`ranges::sort_heap` (C++20)	turns a max heap into a range of elements sorted in ascending order (niebloid)

Minimum/maximum operations

Defined in header <algorithm>

`max`	returns the greater of the given values (function template)
`ranges::max` (C++20)	returns the greater of the given values (niebloid)
`max_element`	returns the largest element in a range (function template)
`ranges::max_element` (C++20)	returns the largest element in a range (niebloid)

Minimum/maximum operations

Defined in header <algorithm>

max	returns the greater of the given values (function template)
ranges::max (C++20)	returns the greater of the given values (niebloid)
max_element	returns the largest element in a range (function template)
ranges::max_element (C++20)	returns the largest element in a range (niebloid)
min	returns the smaller of the given values (function template)
ranges::min (C++20)	returns the smaller of the given values (niebloid)
min_element	returns the smallest element in a range (function template)
ranges::min_element (C++20)	returns the smallest element in a range (niebloid)
minmax (C++11)	returns the smaller and larger of two elements (function template)
ranges::minmax (C++20)	returns the smaller and larger of two elements (niebloid)
minmax_element (C++11)	returns the smallest and the largest elements in a range (function template)
ranges::minmax_element (C++20)	returns the smallest and the largest elements in a range (niebloid)
clamp (C++17)	clamps a value between a pair of boundary values (function template)
ranges::clamp (C++20)	clamps a value between a pair of boundary values (niebloid)

Comparison operations

Defined in header <algorithm>

equal	determines if two sets of elements are the same (function template)
ranges::equal (C++20)	determines if two sets of elements are the same (niebloid)
lexicographical_compare	returns `true` if one range is lexicographically less than another (function template)
ranges::lexicographical_compare (C++20)	returns `true` if one range is lexicographically less than another (niebloid)
lexicographical_compare_three_way (C++20)	compares two ranges using three-way comparison (function template)

Permutation operations

Defined in header <algorithm>

is_permutation (C++11)	determines if a sequence is a permutation of another sequence (function template)
ranges::is_permutation (C++20)	determines if a sequence is a permutation of another sequence (niebloid)

`next_permutation`	generates the next greater lexicographic permutation of a range of elements (function template)
`ranges::next_permutation` (C++20)	generates the next greater lexicographic permutation of a range of elements (niebloid)
`prev_permutation`	generates the next smaller lexicographic permutation of a range of elements (function template)
`ranges::prev_permutation` (C++20)	generates the next smaller lexicographic permutation of a range of elements (niebloid)

Numeric operations

Defined in header `<numeric>`

`iota` (C++11)	fills a range with successive increments of the starting value (function template)
`ranges::iota` (C++23)	fills a range with successive increments of the starting value (niebloid)
`accumulate`	sums up or folds a range of elements (function template)
`inner_product`	computes the inner product of two ranges of elements (function template)
`adjacent_difference`	computes the differences between adjacent elements in a range (function template)
`partial_sum`	computes the partial sum of a range of elements (function template)
`reduce` (C++17)	similar to `std::accumulate`, except out of order (function template)
`exclusive_scan` (C++17)	similar to `std::partial_sum`, excludes the ith input element from the ith sum (function template)
`inclusive_scan` (C++17)	similar to `std::partial_sum`, includes the ith input element in the ith sum (function template)
`transform_reduce` (C++17)	applies an invocable, then reduces out of order (function template)
`transform_exclusive_scan` (C++17)	applies an invocable, then calculates exclusive scan (function template)
`transform_inclusive_scan` (C++17)	applies an invocable, then calculates inclusive scan (function template)

Operations on uninitialized memory

Defined in header `<memory>`

`uninitialized_copy`	copies a range of objects to an uninitialized area of memory (function template)
`ranges::uninitialized_copy` (C++20)	copies a range of objects to an uninitialized area of memory (niebloid)

uninitialized_copy_n (C++11)	copies a number of objects to an uninitialized area of memory (function template)
ranges::uninitialized_copy_n (C++20)	copies a number of objects to an uninitialized area of memory (niebloid)
uninitialized_fill	copies an object to an uninitialized area of memory, defined by a range (function template)
ranges::uninitialized_fill (C++20)	copies an object to an uninitialized area of memory, defined by a range (niebloid)
uninitialized_fill_n	copies an object to an uninitialized area of memory, defined by a start and a count (function template)
ranges::uninitialized_fill_n (C++20)	copies an object to an uninitialized area of memory, defined by a start and a count (niebloid)
uninitialized_move (C++17)	moves a range of objects to an uninitialized area of memory (function template)
ranges::uninitialized_move (C++20)	moves a range of objects to an uninitialized area of memory (niebloid)
uninitialized_move_n (C++17)	moves a number of objects to an uninitialized area of memory (function template)
ranges::uninitialized_move_n (C++20)	moves a number of objects to an uninitialized area of memory (niebloid)
uninitialized_default_construct (C++17)	constructs objects by default-initialization in an uninitialized area of memory, defined by a range (function template)
ranges::uninitialized_default_construct (C++20)	constructs objects by default-initialization in an uninitialized area of memory, defined by a range (niebloid)
uninitialized_default_construct_n (C++17)	constructs objects by default-initialization in an uninitialized area of memory, defined by a start and a count (function template)
ranges::uninitialized_default_construct_n (C++20)	constructs objects by default-initialization in an uninitialized area of memory, defined by a start and count (niebloid)
uninitialized_value_construct (C++17)	constructs objects by value-initialization in an uninitialized area of memory, defined by a range (function template)
ranges::uninitialized_value_construct (C++20)	constructs objects by value-initialization in an uninitialized area of memory, defined by a range (niebloid)

`uninitialized_value_construct_n` (C++17)	constructs objects by value-initialization in an uninitialized area of memory, defined by a start and a count (function template)
`ranges::uninitialized_value_construct_n` (C++20)	constructs objects by value-initialization in an uninitialized area of memory, defined by a start and a count (niebloid)
`destroy` (C++17)	destroys a range of objects (function template)
`ranges::destroy` (C++20)	destroys a range of objects (niebloid)
`destroy_n` (C++17)	destroys a number of objects in a range (function template)
`ranges::destroy_n` (C++20)	destroys a number of objects in a range (niebloid)
`destroy_at` (C++17)	destroys an object at a given address (function template)
`ranges::destroy_at` (C++20)	destroys an object at a given address (niebloid)
`construct_at` (C++20)	creates an object at a given address (function template)
`ranges::construct_at` (C++20)	creates an object at a given address (niebloid)

C library

Defined in header `<cstdlib>`

`qsort`	sorts a range of elements with unspecified type (function)
`bsearch`	searches an array for an element of unspecified type (function)

#include <math.h>

Functions

Trigonometric functions

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

Hyperbolic functions

cosh	Compute hyperbolic cosine (function)
sinh	Compute hyperbolic sine (function)
tanh	Compute hyperbolic tangent (function)
acosh	Compute area hyperbolic cosine (function)
asinh	Compute area hyperbolic sine (function)
atanh	Compute area hyperbolic tangent (function)

Exponential and logarithmic functions

exp	Compute exponential function (function)
frexp	Get significand and exponent (function)
ldexp	Generate value from significand and exponent (function)
log	Compute natural logarithm (function)
log10	Compute common logarithm (function)
modf	Break into fractional and integral parts (function)
exp2	Compute binary exponential function (function)
expm1	Compute exponential minus one (function)
ilogb	Integer binary logarithm (function)
log1p	Compute logarithm plus one (function)
log2	Compute binary logarithm (function)
logb	Compute floating-point base logarithm (function)
scalbn	Scale significand using floating-point base exponent (function)
scalbln	Scale significand using floating-point base exponent (long) (function)

Power functions

pow	Raise to power (function)
sqrt	Compute square root (function)
cbrt	Compute cubic root (function)
hypot	Compute hypotenuse (function)

Error and gamma functions

erf	Compute error function (function)
erfc	Compute complementary error function (function)
tgamma	Compute gamma function (function)
lgamma	Compute log-gamma function (function)

Rounding and remainder functions

ceil	Round up value (function)
floor	Round down value (function)
fmod	Compute remainder of division (function)
trunc	Truncate value (function)
round	Round to nearest (function)
lround	Round to nearest and cast to long integer (function)
llround	Round to nearest and cast to long long integer (function)
rint	Round to integral value (function)
lrint	Round and cast to long integer (function)
llrint	Round and cast to long long integer (function)
nearbyint	Round to nearby integral value (function)
remainder	Compute remainder (IEC 60559) (function)
remquo	Compute remainder and quotient (function)

Floating-point manipulation functions

copysign	Copy sign (function)
nan	Generate quiet NaN (function)
nextafter	Next representable value (function)
nexttoward	Next representable value toward precise value (function)

Floating-point manipulation functions

copysign	Copy sign (function)
nan	Generate quiet NaN (function)
nextafter	Next representable value (function)
nexttoward	Next representable value toward precise value (function)

Minimum, maximum, difference functions

fdim	Positive difference (function)
fmax	Maximum value (function)
fmin	Minimum value (function)

Other functions

fabs	Compute absolute value (function)
abs	Compute absolute value (function)
fma	Multiply-add (function)

Macros / Functions

These are implemented as macros in C and as functions in C++:

Classification macro / functions

fpclassify	Classify floating-point value (macro/function)
isfinite	Is finite value (macro)
isinf	Is infinity (macro/function)
isnan	Is Not-A-Number (macro/function)
isnormal	Is normal (macro/function)
signbit	Sign bit (macro/function)

Comparison macro / functions

isgreater	Is greater (macro)
isgreaterequal	Is greater or equal (macro)
isless	Is less (macro)
islessequal	Is less or equal (macro)
islessgreater	Is less or greater (macro)
isunordered	Is unordered (macro)

Macro constants

math_errhandling	Error handling (macro)
INFINITY	Infinity (constant)
NAN	Not-A-Number (constant)
HUGE_VAL	Huge value (constant)
HUGE_VALF	Huge float value
HUGE_VALL	Huge long double value (constant)

This header also defines the following macro constants (since C99/C++11):

macro	type	description
MATH_ERRNO MATH_ERREXCEPT	int	Bitmask value with the possible values math_errhandling can take.
FP_FAST_FMA FP_FAST_FMAF FP_FAST_FMAL	int	Each, if defined, identifies for which type fma is at least as efficient as x*y+z.
FP_INFINITE FP_NAN FP_NORMAL FP_SUBNORMAL FP_ZERO	int	The possible values returned by fpclassify.
FP_ILOGB0 FP_ILOGBNAN	int	Special values the ilogb function may return.

Types

double_t	Floating-point type (type)
float_t	Floating-point type (type)

Algoritmos Matemáticos

Fatorial:

Python:

```
def oddprod(l,h)
  p = 1
  ml = (l%2>0) ? l : (l+1)
  mh = (h%2>0) ? h : (h-1)
  while ml <= mh do
    p = p * ml
    ml = ml + 2
  end
  p
end

def fact(k)
  f = 1
  for i in 1..k-1
    f *= oddprod(3, 2 ** (i + 1) - 1)
  end
  2 ** (2 ** k - 1) * f
end

print fact(15)
```

```python
import math

def factorial_optimized(n):
    result = 1
    for i in range(2, n + 1):
        if math.isqrt(i) ** 2 == i:  # Check if i is a perfect square
            continue
        while n % i == 0:
            result *= i
            n //= i
    return result
```

C++

```
#include <iostream>
using namespace std;
int result[1000] = {0};
int fact(int n) {
```

```
  if (n >= 0) {
    result[0] = 1;
    for (int i = 1; i <= n; ++i) {
      result[i] = i * result[i - 1];
    }
    return result[n];
  }
}

int main() {
  int n;
  while (1) {
    cout<<"Enter integer to compute factorial (enter 0 to exit): ";
    cin>>n;
    if (n == 0)
    break;
    cout<<fact(n)<<endl;
  }
  return 0;
}
```

Fibonacci

Python:

```
# Fibonacci Series using
# Optimized Method

# function that returns nth
# Fibonacci number


def fib(n):

  F = [[1, 1],
     [1, 0]]
  if (n == 0):
    return 0
  power(F, n - 1)

  return F[0][0]


def multiply(F, M):

  x = (F[0][0] * M[0][0] +
     F[0][1] * M[1][0])
  y = (F[0][0] * M[0][1] +
     F[0][1] * M[1][1])
```

```
    z = (F[1][0] * M[0][0] +
         F[1][1] * M[1][0])
    w = (F[1][0] * M[0][1] +
         F[1][1] * M[1][1])

    F[0][0] = x
    F[0][1] = y
    F[1][0] = z
    F[1][1] = w

# Optimized version of
# power() in method 4


def power(F, n):

    if(n == 0 or n == 1):
        return
    M = [[1, 1],
         [1, 0]]

    power(F, n // 2)
    multiply(F, F)

    if (n % 2 != 0):
        multiply(F, M)


# Driver Code
if __name__ == "__main__":
    n = 9
    print(fib(n))

# This code is contributed
# by ChitraNayal
```

C++

```
// Fibonacci Series using Optimized Method
#include <bits/stdc++.h>
using namespace std;

void multiply(int F[2][2], int M[2][2]);
void power(int F[2][2], int n);

// Function that returns nth Fibonacci number
int fib(int n)
{
    int F[2][2] = { { 1, 1 }, { 1, 0 } };
    if (n == 0)
        return 0;
    power(F, n - 1);

    return F[0][0];
}

// Optimized version of power() in method 4
void power(int F[2][2], int n)
{
    if (n == 0 || n == 1)
```

```
        return;
    int M[2][2] = { { 1, 1 }, { 1, 0 } };

    power(F, n / 2);
    multiply(F, F);

    if (n % 2 != 0)
        multiply(F, M);
}

void multiply(int F[2][2], int M[2][2])
{
    int x =  F[0][0] * M[0][0] + F[0][1] * M[1][0];
    int y =  F[0][0] * M[0][1] + F[0][1] * M[1][1];
    int z =  F[1][0] * M[0][0] + F[1][1] * M[1][0];
    int w = F[1][0] * M[0][1] + F[1][1] * M[1][1];

    F[0][0] = x;
    F[0][1] = y;
    F[1][0] = z;
    F[1][1] = w;
}

// Driver code
int main()
{
    int n = 9;

    cout << fib(9);
    getchar();

    return 0;
}

// This code is contributed by Nidhi_biet
```

Número primo (algoritmo batido)

```
for (int i = 2; i < sqrt(n);i++) {
        if (n % i == 0) {
            return false;
        }
    }
```