Tópicos Avançados em Algoritmos

Hamilton José Brumatto

Bacharelado em Ciências da Computação - UESC

7 de fevereiro de 2019

Algoritmos Genéricos

Estrutura de Dados

- Estrutura de Dados Comum: *Pair*
  - Cria um conjunto básico de dois elementos, digamos um *int* e um *char*.

par

```
class par_int_char {
public:
  int first;
  char second;

  par_int_char(int x, char y) :  first(x), second(y) { }
};
```

Estrutura de Dados

- Usando a estrutura:

main

```
int main() {
 par_int_char par1(12,'a');
 cout <<  par1.first <<  endl;
 cout <<  par1.second <<  endl;

 return 0;
}
```

Saída em console:

```
12
a
```

Estrutura de Dados

- Estrutura de Dados Comum: *Pair*
  - Vamos supor agora que precisemos de outro tipo de par, digamos *bool* e *double*.

par

```
class par_bool_double {
public:
 bool first;
 double second;

 par_bool_double(bool x, double y) :  first(x), second(y) { }
};
```

Estrutura de Dados

- Usando a estrutura:

main

```
int main() {
 par_bool_double par2(true,0.1);
 cout <<  par2.first <<  endl;
 cout <<  par2.second <<  endl;

 return 0;
}
```

Saída em console:

```
1
0.1
```

Estrutura de Dados

- Estrutura de Dados Comum: *Pair*
  - E se precisarmos de um *char* e *long* ?
  - E se a ordem do primeiro e segundo for importante, também?
  - Quantas estruturas teremos de construir?
- Para resolver este problema, temos as *Template Classes* (Classes Modelo)
- Criamos a estrutura baseada em classes modelos, e ao usar indicamos as classes específicas.

Estrutura de Dados

- Estrutura de Dados Comum: *Pair*
  - Utilizando *Template Classes*.

par

```
template< class T1, class T2>
class par {
public:
  T1 first;
  T2 second;

  par(T1 x, T2 y) :  first(x), second(y) { }
};
```

Estrutura de Dados

- Usando a estrutura:

main

```
int main() {
 par<int, char> par3(12,'a');
 par<bool, double> par4(true,0.1);
 par<double, long> par5(3.1415,9999999999);
 par<bool, bool> par3(false,true);
 cout <<  par3.first <<  ':' <<  par3.second <<  endl;
 cout <<  par4.first <<  ':' <<  par4.second <<  endl;
 cout <<  par5.first <<  ':' <<  par5.second <<  endl;
 cout <<  par6.first <<  ':' <<  par6.second <<  endl;

 return 0;
}
```

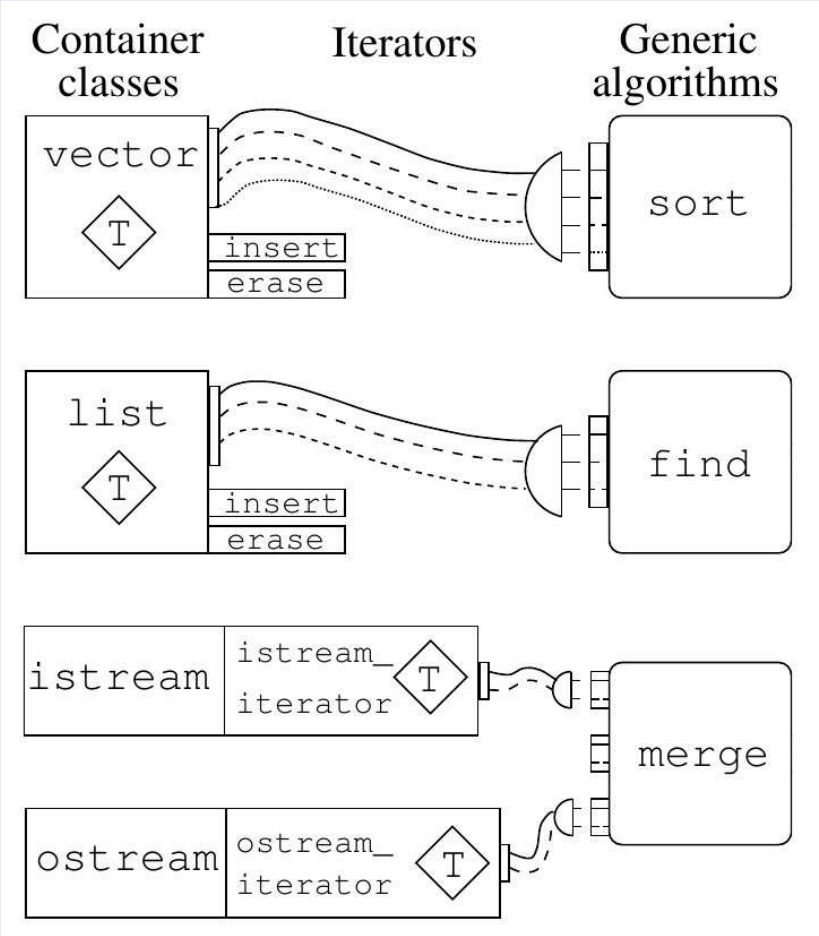
Estrutura de Dados e Algoritmos Genéricos

- O exemplo anterior ilustra a necessidade de classes e algoritmos genéricos.
- A estrutura de dado define uma estrutura complexa, por exemplo: Fila:
  - Na fila os objetos são inseridos no final da fila.
  - Na fila os objetos são removidos do início da fila.
- A estrutura Fila é padrão, não importa o tipo de conteúdo: *int*, *char*, *pilhas* (que tal uma fila de pilhas?)
- Não faz sentido reescrever a estrutura de Fila, dependendo de seu conteúdo.

STL - *Standard Template Libraries*

- STL surgiu de anos de estudo em programação genérica.
- A ideia é construir uma biblioteca de classes genéricas eficientes com componentes reutilizáveis de software.
- Toda a adaptação de software deve ocorrer no preprocessamento da compilação, o que torna o modelo eficiente.
- Associados às classes genéricas, existem também algoritmos genéricos.
- Por exemplo, a uma classe que representa um vetor, podemos ter um algoritmo que realiza a ordenação. A uma classe que representa uma lista, um algoritmo que realiza uma busca.
- A ligação entre as classes (containers) e os algoritmos são os *Iterators*

STL - *Standard Template Libraries*



STL - *Standard Template Libraries*

- Os iterators deve definir o padrão de ligação, nem todo algoritmo genérico é ligável a qualquer classe genérica.
- Por exemplo, o algoritmo de ordenação é mais eficiente em uma classe de acesso aleatório ao invés de uma classe de lista.
- Os iterators são personalizáveis, o programador pode escrever as próprias definições de classes para os iterators.
- Os componentes genéricos se mostram muito mais úteis e, portanto, mais próprio a serem utilizados, do que o programador construir algoritmos e estrutura de dados na unha.

STL - *Standard Template Libraries*

- As principais qualidades do projeto de bibliotecas genéricas:
  - O componentes, organizados de forma precisa, que resultam desta abordagem permite muitas combinações muito mais úteis do que o projeto de componentes tradicional
  - O padrão de projeto também se aplica a outras áreas mais específicas: BD, UI, ...
  - Por utilizar mecanismos em tempo de compilação os componentes são interligados sem comprometer a eficiência do código. Comparado com outras técnicas baseadas em interfaces, heranças e funções virtuais.

A classe Par - uso de Templates

- Retomando nosso uso da classe Par, ela pode ser definida para qualquer tipo, por exemplo uma classe que construímos:

Expandindo o uso de classes STL

```
par<par_int_char, float> par7(par1,1.23);
// de forma equivalente:
par<par<int,char>, float> par8(par3,1.23);

cout << par7.first.first << ':' << par7.first.second << '::' <<
par7.second << endl;
cout << par8.first.first << ':' << par8.first.second << '::' <<
par8.second << endl;
```

Saída:

```
12:1::1.23
12:1::1.23
```

Tipos Template

- Podemos concluir que podemos utilizar um tipo que é Template em qualquer posição onde é esperado um tipo.
- Apesar do conceito ser simples, ao se criar uma classe Template, a solução que simplesmente substitui o tipo nas declarações não a torna eficiente.
- A classe *par*, por exemplo, em sua declaração:
  `par(T1 x, T2 y);`
  passa os valores de `x` e `y` por valor, ou seja, se os objetos são grandes, gasta-se eficiência na cópia dos valores. Seria mais eficiente passar os parâmetros como um referência constante, só é passado o endereço:
  `par(const T1& x, const T2& y);`

A classe *pair* do STL

- Segue a definição da classe *pair* na biblioteca STL

pair

```
template< class T1, class T2>
class pair {
public:
  T1 first;
  T2 second;

  pair(const T1& x, const T2& y) :  first(x), second(y) { }
};
```

Funções Template

- Como já expresso, além das classes genéricas é necessário um código genérico para lidar com as classes.
- Vamos considear um código que retorne o maior de dois números:

```
int intmaior(int x, int y) {
 return x < y ?  y :  x;
}
```

- Mas isto não é eficiente, podemos pensar em termos de tipos genéricos:

```
template class<T>
T maior(T x, T y) {
 return x < y ?  y :  x;
}
```

Funções Template

- Usando a função maior.

main

```
int main() {
 int u = 3, v = 4;
 double d = 4.7;
 par<double, long> par5(3.1415,9999999999);

 cout << maior(u, v) << endl;
 cout << maior(d, 9.3) << endl;
 cout << maior(u, d) << endl;
 cout << maior(par5,par5).first << endl;

 return 0;
}
```

- As duas linhas grifadas em vermelho geram erro de compilação

Funções Template

- Corrigindo o código
  - A linha: `cout << maior(u, d) << endl;` não há como corrigir, isto é um erro lógica. Estamos tentando comparar dois elementos que são de natureza distina.
  - A correção neste caso só é possível com a eliminação da linha.
  - Na linha: `cout << maior(par5,par5).first << endl;` não temos o mesmo problema, pois ambos elementos são do mesmo tipo.
  - O fato é que a classe *par* nós criamos, logo o C++ não sabe como comparar os elementos.
  - Precisamos explicitamente declarar como os elementos serão comparados.

```
bool operator<(const par<double, long>& x, const par<double, long>& y) {
  // vamos compará-los apenas pelos primeiros membros:
  return x.first < y.first;
}
```

Funções Template

- Apesar de ser programação genérica, ao criar o `operador<`, precisamos explicitamente indicar que o primeiro membro é *double* e o segundo *long*. Pois são estes elementos que estamos efetivamente usando no nosso código.
- Da mesma forma que na declaração da classe, para evitar uma cópia dos valores, na função que implementamos, é interessante passar os valores como referência:

```
template <class T>
const T& max(const T& x, const T& y) {
 return x < y ?  y :  x;
}
```

- Esta é a definição da função usada na biblioteca STL

Template Member Functions

- Além do tipo template declarado no nível da classe, cada método da classe pode ter uma definição própria de Template, independente o tipo declarado na classe:

```
template <class T>
class vector {
 ...
 template <class Iterator>
 insert(Iterator position, Iterator, first, Iterator last);
 ...
};
```

- O método `insert` pode ser utilizado para inserir elementos copiados de outra container (uma lista, por exemplo) utilizando iterators que o outro container fornece.0

Parâmetros padrão para Templates

- Algumas classes (ou funções) podem ser construídas com base em Templates. Porém ao usar, a escolha mais frequente para o tipo é um tipo específico.
- É possível definir Templates onde, na ausência da declaração de um tipo de escolha, assume-se um padrão.
- Veja a declaração da classe *vector* no STL:

```
template <class T, class Allocator = allocator>
class vector {
...
};
```

- As classes containeres do STL utiliza um gerente de armazenamento definido por `Allocator` e `allocator` é um tipo (classe) que provê uma forma padrão deste gerenciamento.
- Assim, ao declarar: `vector<int>`, estamos fazendo o equivalente a: `vector<int,allocator>`.

Exemplo do Vector

- *Vector* são containers de vetor em sequência que pode mudar de tamanho.
- Tal qual vetores (arrays) Vector representa elementos em alocação contígua de memória.
  - Os elementos podem ser acessados, de forma eficiente, por indireção de ponteiros (`v[i]` ou `*(&v + i)`)
  - Diferente de arrays, ele pode crescer dinamicamente, sendo que o armazenamento é gerenciado pelo `Allocator`
  - *Vector* consome mais memória do que arrays, para crescer de forma eficiente.

```
template <class T, class Alloc = allocator<T> > class vector;
```

Vector - Métodos: Construtor

Construtor	Significado
`vector<T> v();`	Um vetor de elementos do tipo T vazio
`vector<T> v(n);`	Um vetor com `n` elementos do tipo T
`vector<T> v(n,x);`	Um vetor com `n` elementos `x`
`template<Iterator>`	
`vector<T> v(first, last);`	Vetor preenchido, first e last são iterators
`vector<T> v(w);`	Cópia do vetor `w` do mesmo tipo.

```
#include <vector>
#include <string.h>

using namespace std;

int main() {
  char teste[] = ''Teste de vetor'';

  vector<char> v1(teste, teste+strlen(teste));
  return 0;
}
```

Vector - Métodos: Iterator

Método	Significado
`begin`	retorna um iterator para o início
`end`	retorna um iterator para o final
`rbegin`	retorna um iterator reverso para o início reverso
`rend`	retorna um iterator reverso para o fim reverso

Vector - Métodos: Capacidade

Método	Significado
`size`	retorna o tamanho (quantidade de elementos)
`max_size`	retorna o tamanho máximo que pode atingir
`resize`	modifica a quantidade de elementos
`capacity`	retorna o tamanho alocado
`empty`	verifica se está vazio
`reserve`	modifica o tamanho alocado
`shrink_to_fit`	modifica capacity para size

Vector - Métodos: Acesso aos elementos

Método	Significado
`operator[]`	acessa o elemento
`at`	acessa o elemento
`front`	acessa o primeiro elemento
`back`	acessa o último elemento
`data`	retorna um ponteiro para o primeiro elemento

Vector - Métodos: Modificadores

Método	Significado
`assign`	atribui um conjunto de valores
`push_back`	insere elementos no final do vetor
`pop_backdd`	remove o último elemento
`insert`	insere deslocando elementos
`erase`	remove um elemento deslocando os demais
`swap`	troca o conteúdo de dois vetores do mesmo tipo
`clear`	esvazia o vetor
`emplace`	semelhante a `insert`
`emplace_back`	semelhante a `push_back`

O algoritmo genérico: `reverse`

- Um exemplo de um algoritmo genérico é o `reverse`:

```
template <class BidiretionalIterator>
 void reverse (BidirectionalIterator first, BidirectionalIterator last) {
 while ((first!=last)&&(first!=--last)) {
  std::iter_swap(first,last);
 }
}
```

- E para o `vector` nós temos os Iterators: `begin()` e `end()`
- Vamos usar o algoritmo genérico em um vector:

O algoritmo genérico: reverse

```
#include <iostream>
#include <algorithm>
#include <vector>
#include <string.h>
using namespace std;

ostream& operator<<(ostream& o, const vector<char>& v) {
 for(int i=0; i<v.size(); i++) o << v[i];
 return o;
}

int main() {
 char teste[] = ''Teste de vetor'';
 cout << ''Exemplo do algoritmo genérico \''reverse\'''' << endl;
 vector<char> v1(teste, teste+strlen(teste));

 reverse(v1.begin(), v1.end());

 cout << v1 << endl;
 return 0;
}
```

Dá para fazer mais simples?

Usando o container string

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

int main() {
 string teste = ''Teste de vetor'';
 cout << ''Exemplo do algoritmo genérico \''reverse\'''' << endl;

 reverse(teste.begin(), teste.end());

 cout << teste << endl;
 return 0;
}
```

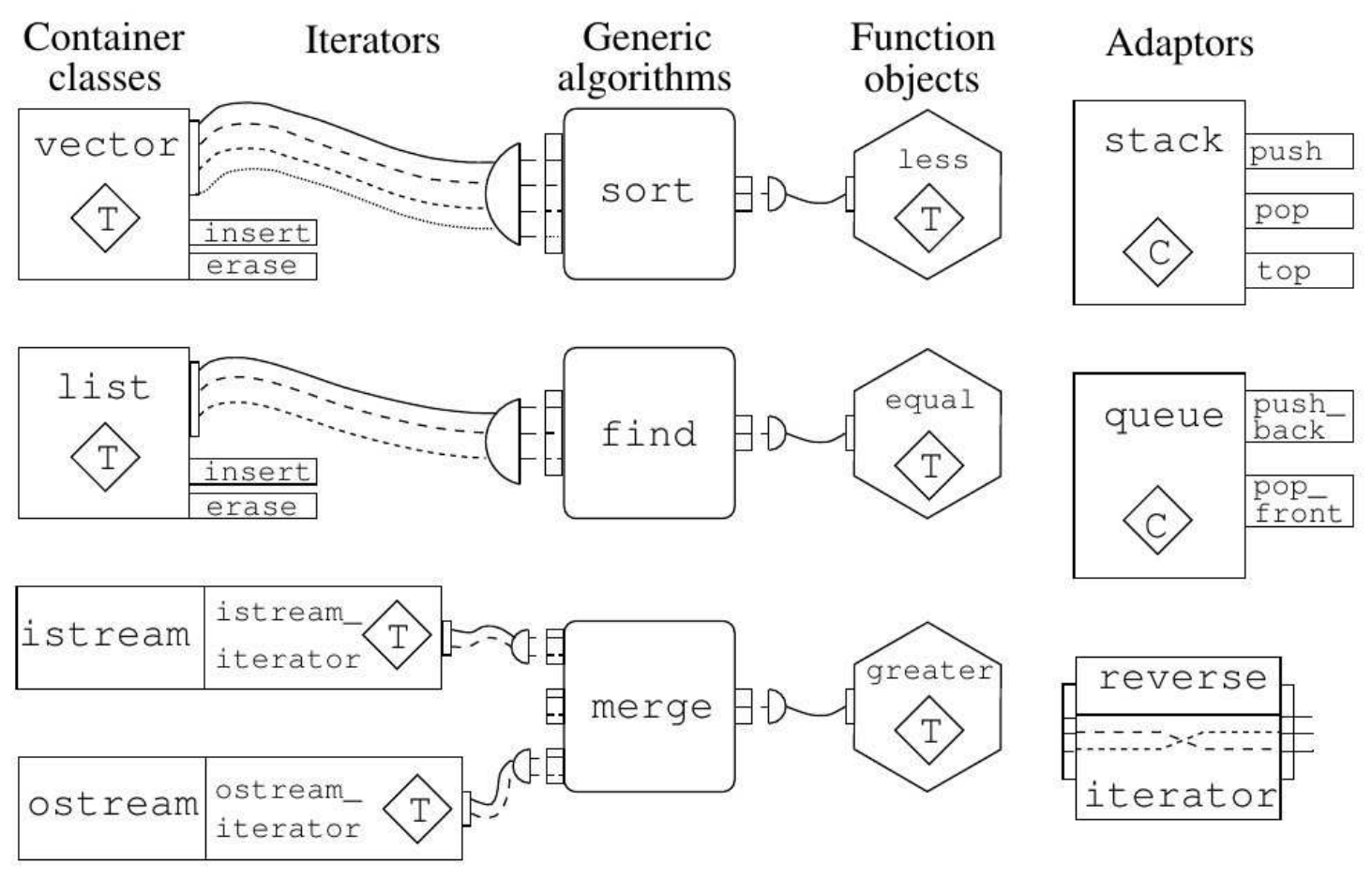
Algoritmo genérico

- Um exemplo de algoritmo genérico é o reverse
- Utilizamos o reverse para trocar a ordem dos elementos em um `vector`.
- Da mesma forma o fizemos para o `string`
- O detalhe é que ambos containeres definem iterators do tipo bidirecional necessário para o algoritmo.
- Qualquer outro container que defina iterators deste tipo pode receber o algoritmo `reverse`.

STL e seus componentes

- O STL contém 6 tipos principais de componentes:
  - Containers: Objetos que armazenam coleções de outros objetos
  - Algoritmos Genéricos: Algoritmos que realizam ações sobre containers, baseados em ligações através de iterators.
  - Iterators: Chave principal para o STL, uma visão simples é considerar iterators como ponteiros (eles agem como ponteiros), e o ponteiro é um iterator.
  - Objetos Funções (Function Objects): É possível definir uma classe que não possua dado armazenado, apenas função, o objeto desta classe é uma função.
  - Adaptadores (Adaptors): Adaptadores podem alterar as características de um iterator ou de um container gerando nova funcionalidade.
  - Alocadores (Allocators): Todos containeres possuem um alocador para gerenciar o conteúdo baseado no modelo de memória. Como o STL já possui um Allocator default, não vamos tratar deste componente.

STL e seus componentes: 5 aqui



Containers

- São objetos que armazenam coleções de outros objetos. Dividimos em três categorias:
- Containers de Sequência (Sequence Containers):
  - Um vetor tradicional do C: `T a[n]`. Provê acesso aleatório em tempo constante.
  - Vector: `vector<T>`. Provê acesso aleatório em tempo constante em uma sequência de tamanho variável, inserções e remoções em tempo constante no final.
  - Deque: `deque<T>`. Semelhante ao Vector, porém as inserções e remoções em tempo constante podem ocorrem em ambas pontas.
  - Lista: `forward_list<T>` e `list<T>`. Sequência de tamanho variado com acesso em tempo linear ($O(n)$), mas tempo constante de inserção e remoção em qualquer posição. A primeira tem ligação simples e a segunda dupla.

Containers

- Containers de Associação com Ordenação (Sorted Associative Containers):
  - Set: `template <class T, class Compare=less<T>> class set`: `set<Key>`, aceita chaves únicas. A chave é o próprio elemento, armazenado de forma ordenada. Um elemento é diferente do outro. Os elementos são constantes, não podem ser modificados.
  - MultiSet: `multiset<Key>`, semelhante ao Set, mas mais de um elemento pode possuir o mesmo valor.
  - Map: `map<Key,T>`, permite associar um objeto T a um valor de chave Key. As chaves são únicas.
  - Multimap: `multimap<Key,T>`, semelhante a Map, mas as chaves podem ter valor repetido.

Containers

- Containers de Associação sem Ordenação (Unordered Associative Containers):
- Diferente dos Sorted Associative Containers, as chaves não são mantidas de forma ordenada, porém em uma tabela HASH (agrupados em caixas), no modelo Bucked Sort.
- Enquanto que no Sorted Associative Containers as buscas individuais são em tempo ($O(\log n)$, no Unordered é ($O(1)$).
- Porém é menos eficiente se for tratar de um intervalo de elementos.
- Os containers são: `unordered_set`, `unordered_multiset`, `unordered_map` e `unordered_multimap`.

Referência STL: http://www.cplusplus.com/reference/stl/

Headers		<array>	<vector>	<deque>	<forward_list>	<list>
Members		array	vector	deque	forward_list	list
	constructor	*implicit*	vector	deque	forward_list	list
	destructor	*implicit*	~vector	~deque	~forward_list	~list
	operator=	*implicit*	operator=	operator=	operator=	operator=
iterators	begin	begin	begin	begin	begin before_begin	begin
	end	end	end	end	end	end
	rbegin	rbegin	rbegin	rbegin		rbegin
	rend	rend	rend	rend		rend
const iterators	cbegin	cbegin	cbegin	cbegin	cbegin cbefore_begin	cbegin
	cend	cend	cend	cend	cend	cend
	crbegin	crbegin	crbegin	crbegin		crbegin
	crend	crend	crend	crend		crend
capacity	size	size	size	size		size
	max_size	max_size	max_size	max_size	max_size	max_size
	empty	empty	empty	empty	empty	empty
	resize		resize	resize	resize	resize
	shrink_to_fit		shrink_to_fit	shrink_to_fit		
	capacity		capacity			
	reserve		reserve			
element access	front	front	front	front	front	front
	back	back	back	back		back
	operator[]	operator[]	operator[]	operator[]		
	at	at	at	at		
modifiers	assign		assign	assign	assign	assign
	emplace		emplace	emplace	emplace_after	emplace
	insert		insert	insert	insert_after	insert
	erase		erase	erase	erase_after	erase
	emplace_back		emplace_back	emplace_back		emplace_back
	push_back		push_back	push_back		push_back
	pop_back		pop_back	pop_back		pop_back
	emplace_front			emplace_front	emplace_front	emplace_front
	push_front			push_front	push_front	push_front
	pop_front			pop_front	pop_front	pop_front
	clear		clear	clear	clear	clear
	swap	swap	swap	swap	swap	swap
list operations	splice				splice_after	splice
	remove				remove	remove
	remove_if				remove_if	remove_if
	unique				unique	unique
	merge				merge	merge
	sort				sort	sort
	reverse				reverse	reverse
observers	get_allocator		get_allocator	get_allocator	get_allocator	get_allocator
	data	data	data			

Containers

- Neste exemplo, não está listado aqui, mas também fizemos a sobreposição do operador <<

Exemplo usando o `list`

```
int main() {
 string teste = ''Teste de lista'';
 string::iterator i = teste.begin();
 list<char> lst;

 while(i != teste.end())
  lst.push_back((char) *i++);

 reverse(lst.begin(), lst.end());

 cout << lst << endl;
 return 0;
}
```

Algoritmos Genéricos: <algorithm>

- Os algoritmos genéricos de STL estão declarados no include <algorithm>. São classificados como:
- Operações em sequência que não modificam.
  - all_of, any_of, none_of, for_each, find, find_if, find_if_not, find_end, find_first_of, adjacent_find, count, count_if, mismatch, equal, is_permutation, search, search_n.
- Operações em sequência que modificam.
  - copy, copy_n, copy_if, copy_backward, move, move_backward, swap, swap_ranges, iter_swap, transform, replace, replace_if, replace_copy, replace_copy_if, fill, fill_n, generate, generate_n, remove, remove_if, remove_copy, remove_copy_if, unique, unique_copy, reverse, reverse_copy, rotate, rotate_copy, random_shuffle, shuffle.

Algoritmos Genéricos: <algorithm>

- Partições:
  - `is_partitioned`, `partition`, `stable_partition`, `partition_copy`, `partition_point`.
- Ordenação:
  - `sort`, `stable_sort`, `partial_sort`, `partial_sort_copy`, `is_sorted`, `is_sorted_until`, `nth_element`.
- Busca binária (operando em intevalos ordenados/particionados):
  - `lower_bound`, `upper_bound`, `equal_range`, `binary_search`.
- Merge (intercalação):
  - `merge`, `inplace_merge`, `includes`, `set_union`, `set_intersection`, `set_difference`, `set_symmetric_difference`.

Algoritmos Genéricos: <algorithm>

- Heap:
  - `push_heap`, `pop_heap`, `make_heap`, `sort_heap`, `is_heap`, `is_heap_until`
- Min/Max:
  - `min`, `max`, `minmax`, `min_element`, `max_element`, `minmax_element`.
- Outros:
  - `lexicographical_compare`, `next_permutation`, `prev_permutation`.

Algoritmos Genéricos

- Vamos considerar 2 destes algoritmos: `foreach` e `merge`
- `foreach`

```
template <class InputIterator, class Function>
 Function for_each (InputIterator first, InputIterator last, Function fn);
```

- A implementação deste algoritmo é:

```
template<class InputIterator, class Function>
 Function for_each(InputIterator first, InputIterator last, Function fn) {
  while (first!=last) {
   fn (*first);
   ++first;
  }
 return move(fn);
}
```

- `fn` tanto pode ser função, como Objeto Função.

Algoritmos Genéricos: for_each

- Para não fazer a sobreposição do operador <<

```
void func(char c) {
  cout << c;
}

int main() {
  string teste = ''Teste de lista'';
  string::iterator i = teste.begin();
  list<char> lst;

  while(i != teste.end())
    lst.push_back((char) *i++);

  reverse(lst.begin(), lst.end());

  for_each(lst.begin(), lst.end(), func);
  cout << endl;
  return 0;
}
```

Algoritmos Genéricos: merge

- Opera sobre duas sequências, do mesmo tipo, ordenadas, e gera uma sequência que é intercalação de ambas:

```
template <class InputIterator1, class InputIterator2, class OutputIterator>
 OutputIterator merge (InputIterator1 first1, InputIterator1 last1,
                       InputIterator2 first2, InputIterator2 last2,
                       OutputIterator result);
```

- A implementação é:

```
template <class InputIterator1, class InputIterator2, class OutputIterator>
 OutputIterator merge (InputIterator1 first1, InputIterator1 last1,
                       InputIterator2 first2, InputIterator2 last2,
                       OutputIterator result) {
 while (true) {
  if (first1==last1) return std::copy(first2,last2,result);
  if (first2==last2) return std::copy(first1,last1,result);
  *result++ = (*first2<*first1)?  *first2++ :  *first1++;
 }
}
```

Algoritmos Genéricos: merge

Exemplo usando o merge

```
int main () {
 int first[] = {5,10,15,20,25};
 int second[] = {10,20,30,40,50};
 vector<int> v(10);

 merge (first,first+5,second,second+5,v.begin());

 cout << ''O vetor resultante contém:'';
 for (vector<int>::iterator it=v.begin(); it!=v.end(); it++)
   cout << ' ' << *it;
 cout << endl;

 return 0;
}
```

Algoritmos genéricos: "Ao infinito... e além!"

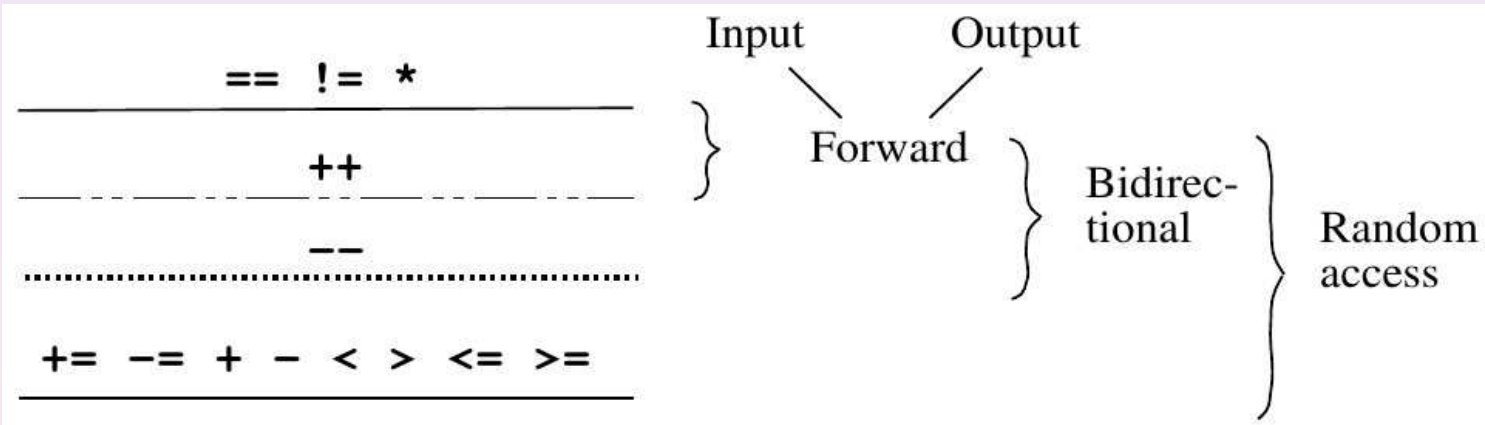
- `<algorithm>` é apenas uma biblioteca de algoritmos genéricos, em específico STL.
- Existem outras bibliotecas (nem todas aqui, nem todas de algoritmos genéricos):
  - `<memory>`: Elementos de memória
  - `<numeric>`: Operações numéricas gerais.
  - `<string>`: Strings
  - `<valarray>`: Vetores de valores numéricos
- Além destas, as bibliotecas para (`<iterator>`) iterators, (`<functional>`) objetos funções, (`<initializer_list>`) listas de inicialização, ...

Iterators

- Compreender `iterators` é a peça chave para entender completamente o STL e como utilizá-lo.
- Os algoritmos genéricos são escritos em função de iterators.
- Os containers provêem iterators que podem ser ligados aos algoritmos genéricos.
- Os iterators são diferenciado por categorias hierárquicas.
- Os ponteiros são a forma mais básica de iterators, no entanto iterators de outros tipos existem, sendo que todos realizam as operações que os ponteiros realizam: `++`, e `*`.
  - `++i` avança o iterator `i` para a próxima localização
  - `*i` retorna o valor da expressão (`x = *i`), ou a localização onde armazenar um valor: (`*i = x`)

Iterators

- A figura abaixo representa as categorias de iterators que são: de entrada (input) e de saída (output), e os hierárquicos: direto (forward), bidirecional (bidirectional) e acesso aleatório (random access), através de suas relações hierárquicas:



  - Indireção: ==, != e *
  - Incremento: ++
  - Decremento: --
  - Grandes saltos: +=, -=, + e - e
  - Comparações: <, >, <= e >=

Objetos Funções (Function Objects)

- Aqui começamos a aproveitar características da Orientação a Objetos.
- Objetos Funções são obtidos de classes onde há a sobreposição do operador "()".
- Ao sobrepor este operador, na forma de uma função, quando instanciamos a classe, ao invés de obter um objeto comum, obtemos um objeto que é a função que criamos.
- Vamos tomar um exemplo, usando o algoritmo genérico `accumulate`, da biblioteca `<numeric>`. Uma versão do `accumulate` permite definir a função que acumula, no lugar do tradicional '+'

Objetos Funções

- Declaração do accumulate

```
template <class InputIterator, class T, class BinaryOperation>
 T accumulate (InputIterator first, InputIterator last, T init,
               BinaryOperation binary_op);
```

- Este algoritmo genérico é implementado da seguinte forma:

```
template <class InputIterator, class T>
 T accumulate (InputIterator first, InputIterator last, T init
               BinaryOperation binary_op) {
  while(first != last) {
   init = binary_op(init,*first);
   ++first;
  }
  return init;
 }
```

Objetos Funções: Exemplo com função comum

Exemplo de accumulate com função comum

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

int mult(int x, int y) {return x * y;}

int main () {
 int v[] = {2,3,5,7,11};
 cout << ''Algoritmo genérico accumulate para produto'' << endl;

 int prod = accumulate(v, v+5, 1, mult);

 cout << prod << endl;
 return 0;
}
```

Objetos Funções

Exemplo de accumulate usando um objeto função

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

class multiply {
public:
  int operator() (int x, int y) {return x * y; }
};

int main () {
  int v[] = {2,3,5,7,11};
  cout << ''Algoritmo genérico accumulate para produto'' << endl;

  int prod = accumulate(v, v+5, 1, multiply());

  cout << prod << endl;
  return 0;
}
```

Objetos Funções

- Nesta segunda versão, no lugar da função `mult` utilizamos uma instância da classe `multiply` → `multiply()`
- Poderíamos ter instanciado antes:

```
class multiply {
public:
 int operator() (int x, int y) {return x * y; }
} objmult;
...
 int prod = accumulate(v, v+5, 1, objmult);
```

Objetos Funções

- A vantagem de utilizar classes e objetos funções podemos tratar mais tarde, mas de pronto, temos a flexibilidade de utilizar templates:

```
template <class T>
class multiply {
public:
 T operator() (const T& x, const T& y) {return x * y; }
} objmult;
...
 int prod = accumulate(v, v+5, 1, multiply<int>());
```

- O objeto da classe multiply passa a ser um algoritmo genérico que podemos utilizar no accumulate para qualquer tipo de sequência. (Desde que o tipo implemente o operador *).

Adaptadores (Adaptors)

- Um componente que modifica a interface de outro componente é chamado de *Adaptor*.
- Como exemplo temo o *reverse_iterator* que é um componente que adapta um iterator em um novo tipo. As operações '++' passam a ser '--' e vice-versa.
- Também como exemplo temos o container *stack* que é um componente que adapta o container *deque* para uma nova funcionalidade: *pilha*
- Os Containers Adaptors são:
  - `stack`: Container de Pilha (LIFO)
  - `queue`: Container de Fila (FIFO)
  - `priority_queue`: Container de Fila de prioridade.

Problema 1252 do URI Online Judge: Sort! Sort!! e Sort!!!

Hmm! Aqui você foi solicitado a fazer uma simples ordenação. A você serão dado **N** números e um inteiro positivo **M**. Você terá que ordenar estes **N** números em ordem ascendente de seu módulo **M**. Se houver um empate entre um número ímpar e um número par (para os quais o seu módulo **M** dá o mesmo valor) então o número impar irá preceder o número par. Se houver um empate entre dois números ímpares (para os quais o seu módulo **M** dá o mesmo valor), então o maior número ímpar irá preceder o menor número ímpar. Se houve um empate entre dois números pares (para os quais o seu módulo **M** dá o mesmo valor), então o menor número par irá preceder o maior número par. Para o resto de valores negativos siga a regra de linguagem de programação C: um número negativo nunca pode ter módulo maior do que zero. Por exemplo, -100 MOD 3 = -1, -100 MOD 4 = 0, etc.

Entrada:

A entrada contém vários casos de teste. Cada caso de teste inicia com dois inteiros **N** (0 ¡ **N** $\leqslant$ 10000) e **M** (0 ¡ **M** $\leqslant$ 10000) que denotam quantos números existirão neste conjunto. Cada uma das próximas **N** linhas conterá um número cada. Estes números deverão caber em um inteiro de 32 bits com sinal. A entrada é terminada por uma linha que conterá dois valores nulos (0) e não deve ser processada.

Saída:

A primeira linha de cada conjunto de saída irá contér os valores de **N** e **M**. As próximas **N** linhas irão contér **N** números, ordenados de acordo com as regras acima mencionadas. Imprima os dois últimos zeros da entrada para a saída padrão.

Resolvendo o problema: Algoritmo genérico `sort`

- Vamos usar
  a versão que nos dá a opção de indicar a função de comparação:

```
template <class RandomAccessIterator, class Compare>
  void sort (RandomAccessIterator first, RandomAccessIterator last, Compare comp);
```

- A função (ou função objeto) comp irá retornar um valor booleano que indica verdade se o primeiro elemento anteceder o segundo elemento na ordem.
- A função não pode alterar o conteúdo dos valores.
- Como a função somente pode receber os dois elementos, e precisamos de um valor inteiro para a comparação de módulo, entre os dois, então este valor inteiro será uma variável global.

Função de comparação para sort

```
#include <iostream> // cout
#include <algorithm> // sort
#include <vector> // vector
using namespace std;
int mod;

class less {
public:
 bool operator() (const int& x, const int& y) {
  bool res;
  res = x%mod < y%mod;
  if(x%mod == y%mod) {
   if(x%2 && y%2) res = y < x;
   else if(!(x%2) && !(y%2)) res = x < y;
   else
    if(x%2) res = true;
    else res = false;
  }
  return res;
 }
} objless;
```

Função de impressão do for_each e o main

```
void func(int i) {
  cout << i << endl;
}
int main () {
  vector<int> v;
  int n, k;
  cin >> n >> mod;
  while(n) {
    for(int i=0; i<n; i++) {
      cin >> k;
      v.push_back(k);
    }
    sort(v.begin(),v.end(),objless);
    cout << n << ' ' << mod << endl;
    for_each(v.begin(),v.end(),func);
    v.clear();
    cin >> n >> mod;
  }
  cout << n << ' ' << mod << endl;
  return 0;
}
```