Classe `list`

Artigo • 16/06/2023

A classe de lista da Biblioteca C++ Standard é um modelo de classe de contêineres de sequência que mantém seus elementos em uma organização linear e permite inserções e exclusões eficientes em qualquer local na sequência. A sequência é armazenada como uma lista vinculada bidirecional de elementos, cada um contendo um membro de algum tipo de *`Type`*.

Sintaxe

C++

```
template <class Type, class Allocator= allocator<Type>>
class list
```

Parâmetros

`Type`

O tipo de dados do elemento a ser armazenado na lista.

`Allocator`

O tipo que representa o objeto alocador armazenado que encapsula detalhes sobre a alocação e desalocação de memória da lista. Esse argumento é opcional e o valor padrão é `allocator<Type>`.

Comentários

A escolha do tipo de contêiner deve se basear, de modo geral, no tipo de pesquisa e inserção exigido pelo aplicativo. Vetores devem ser o contêiner preferencial para gerenciar uma sequência quando o acesso aleatório a qualquer elemento é alto e inserção e exclusões de elementos apenas são necessárias no final de uma sequência. O desempenho do contêiner de deque de classe é superior quando o acesso aleatório é necessário e as inserções e exclusões no início e final de uma sequência são essenciais.

As funções de membro da lista merge, reverse, unique, remove e remove_if foram otimizadas para funcionar com objetos da lista e oferecer uma alternativa de alto desempenho para os seus correspondentes genéricos.

A realocação da lista ocorre quando uma função de membro deve inserir ou apagar elementos da lista. Nesses casos, somente iteradores ou referências que apontem para partes apagadas da sequência controladas tornam-se inválidos.

Inclua o cabeçalho padrão da Biblioteca C++ Standard `<list>` para definir a lista do modelo de classe container e vários modelos de suporte.

Membros

Construtores

Expand table

Nome	Descrição
list	Constrói uma lista de um tamanho específico, ou com elementos de um valor específico, ou com um `allocator` específico, ou como uma cópia de alguma outra lista.

Typedefs

Expand table

Nome	Descrição
allocator_type	Um tipo que representa a classe `allocator` para um objeto de lista.
const_iterator	Um tipo que fornece um iterador bidirecional que pode ler um elemento **`const`** em uma lista.
const_pointer	Um tipo que fornece um ponteiro para um elemento **`const`** em uma lista.
const_reference	Um tipo que fornece uma referência para um elemento **`const`** armazenado em uma lista para leitura e execução de operações **`const`**.
const_reverse_iterator	Um tipo que fornece um iterador bidirecional que pode ler qualquer elemento **`const`** em uma lista.
difference_type	Um tipo que fornece a diferença entre dois iteradores que se referem a elementos na mesma lista.
iterator	Um tipo que fornece um iterador bidirecional que pode ler ou modificar qualquer elemento em uma lista.
pointer	Um tipo que fornece um ponteiro para um elemento em uma lista.

Nome	Descrição
reference	Um tipo que fornece uma referência para um elemento `const` armazenado em uma lista para leitura e execução de operações `const`.
reverse_iterator	Um tipo que fornece um iterador bidirecional que pode ler ou modificar um elemento em uma lista invertida.
size_type	Um tipo que conta o número de elementos em uma lista.
value_type	Um tipo que representa o tipo de dados armazenado em uma lista.

Funções

Expand table

Nome	Descrição
assign	Apaga os elementos de uma lista e copia um novo conjunto de elementos na lista de destino.
back	Retorna uma referência ao último elemento de uma lista.
begin	Retorna um iterador que trata o primeiro elemento em uma lista.
cbegin	Retorna um iterador const que trata o primeiro elemento em uma lista.
cend	Retorna um iterador const que trata o local após o último elemento em uma lista.
clear	Apaga todos os elementos de uma lista.
crbegin	Retorna um iterador const que trata o primeiro elemento em uma lista invertida.
crend	Retorna um iterador const que trata o local após o último elemento em uma lista invertida.
emplace	Insere um elemento construído no local em uma lista na posição especificada.
emplace_back	Adiciona um elemento construído no local ao final de uma lista.
emplace_front	Adiciona um elemento construído no local ao início de uma lista.
empty	Testa se uma lista está vazia.
end	Retorna um iterador que trata o local após o último elemento em uma lista.
erase	Remove um elemento ou um intervalo de elementos das posições especificadas.
front	Retorna uma referência ao primeiro elemento em uma lista.

Nome	Descrição
get_allocator	Retorna uma cópia do objeto `allocator` usada para construir uma lista.
insert	Insere um elemento ou um número de elementos ou um intervalo de elementos em uma lista, na posição especificada.
max_size	Retorna o tamanho máximo de uma lista.
merge	Remove os elementos da lista de argumentos, insere-os na lista de destino e organiza o conjunto novo e combinado de elementos em ordem crescente ou em alguma outra ordem especificada.
pop_back	Exclui o elemento no final de uma lista.
pop_front	Exclui o elemento no começo de uma lista.
push_back	Adiciona um elemento ao fim da lista.
push_front	Adiciona um elemento ao começo da lista.
rbegin	Retorna um iterador que trata o primeiro elemento em uma lista inversa.
remove	Apaga elementos em uma lista que correspondem a um valor especificado.
remove_if	Apaga os elementos da lista para a qual um predicado especificado foi atendido.
rend	Retorna um iterador que trata o local após o último elemento em uma lista invertida.
resize	Especifica um novo tamanho para uma lista.
reverse	Reverte a ordem na qual os elementos ocorrem em uma lista.
size	Retorna o número de elementos em uma lista.
sort	Organiza os elementos de uma lista em ordem crescente ou com respeito a alguma outra relação de ordem.
splice	Remove os elementos da lista de argumentos e os insere na lista de destino.
swap	Troca os elementos das duas listas.
unique	Remove elementos duplicados adjacentes ou elementos adjacentes que satisfazem algum predicado binário da lista.

Operadores

Expand table

Nome	Descrição
operator=	Substitui os elementos da lista por uma cópia de outra lista.

Requisitos

Cabeçalho: `<list>`

allocator_type

Um tipo que representa a classe de alocador para um objeto de lista.

C++

```
typedef Allocator allocator_type;
```

Comentários

`allocator_type` é um sinônimo do parâmetro de modelo *`Allocator`*.

Exemplo

Confira o exemplo de get_allocator.

assign

Apaga os elementos de uma lista e copia um novo conjunto de elementos na lista de destino.

C++

```
void assign(
    size_type Count,
    const Type& Val);

void assign
    initializer_list<Type> IList);

template <class InputIterator>
void assign(
    InputIterator First,
    InputIterator Last);
```

Parâmetros

First

Posição do elemento primeiro no intervalo de elementos ser copiado da lista de argumentos.

Last

Posição do primeiro elemento além do intervalo de elementos a ser copiado da lista de argumentos.

Count

O número de cópias de um elemento sendo inseridos na lista.

Val

O valor do elemento sendo inserido na lista.

IList

O initializer_list que contém os elementos a serem inseridos.

Comentários

Depois de apagar os elementos da lista de destino, atribua às inserções um intervalo especificado de elementos a partir da lista original ou de alguma outra lista na lista de destino ou insira cópias de um novo elemento de um valor especificado na lista de destino

Exemplo

C++

```
// list_assign.cpp
// compile with: /EHsc
#include <list>
#include <iostream>

int main()
{
    using namespace std;
    list<int> c1, c2;
    list<int>::const_iterator cIter;

    c1.push_back(10);
    c1.push_back(20);
    c1.push_back(30);
    c2.push_back(40);
    c2.push_back(50);
```

```
    c2.push_back(60);

    cout << "c1 =";
    for (auto c : c1)
        cout << " " << c;
    cout << endl;

    c1.assign(++c2.begin(), c2.end());
    cout << "c1 =";
    for (auto c : c1)
        cout << " " << c;
    cout << endl;

    c1.assign(7, 4);
    cout << "c1 =";
    for (auto c : c1)
        cout << " " << c;
    cout << endl;

    c1.assign({ 10, 20, 30, 40 });
    cout << "c1 =";
    for (auto c : c1)
        cout << " " << c;
    cout << endl;
}
```

Output

```
c1 = 10 20 30c1 = 50 60c1 = 4 4 4 4 4 4 4c1 = 10 20 30 40
```

back

Retorna uma referência ao último elemento de uma lista.

C++

```
reference back();

const_reference back() const;
```

Valor de Devolução

O último elemento da lista. Se a lista estiver vazia, o valor de retorno será indefinido.

Comentários

Se o valor de retorno de `back` for atribuído a `const_reference`, o objeto de lista não poderá ser modificado. Se o valor retornado de `back` for atribuído à `reference`, o objeto de lista poderá ser modificado.

Quando compilado usando _ITERATOR_DEBUG_LEVEL definido como 1 ou 2, um erro de runtime ocorrerá se você tentar acessar um elemento em uma lista vazia. Consulte Iteradores Verificados para obter mais informações.

Exemplo

C++

```
// list_back.cpp
// compile with: /EHsc
#include <list>
#include <iostream>

int main( )
{
   using namespace std;
   list <int> c1;

   c1.push_back( 10 );
   c1.push_back( 11 );

   int& i = c1.back( );
   const int& ii = c1.front( );

   cout << "The last integer of c1 is " << i << endl;
   i--;
   cout << "The next-to-last integer of c1 is " << ii << endl;
}
```

Output

```
The last integer of c1 is 11
The next-to-last integer of c1 is 10
```

begin

Retorna um iterador que trata o primeiro elemento em uma lista.

C++

```
const_iterator begin() const;
```

```
iterator begin();
```

Valor de Devolução

Um iterador bidirecional que aborda o primeiro elemento na lista ou o local que substitui uma lista vazia.

Comentários

Se o valor retornado de `begin` foi atribuído a um `const_iterator`, os elementos no objeto da lista não poderão ser modificados. Se o valor retornado de `begin` foi atribuído a um `iterator`, os elementos no objeto da lista poderão ser modificados.

Exemplo

C++

```
// list_begin.cpp
// compile with: /EHsc
#include <list>
#include <iostream>

int main( )
{
   using namespace std;
   list <int> c1;
   list <int>::iterator c1_Iter;
   list <int>::const_iterator c1_cIter;

   c1.push_back( 1 );
   c1.push_back( 2 );

   c1_Iter = c1.begin( );
   cout << "The first element of c1 is " << *c1_Iter << endl;

*c1_Iter = 20;
   c1_Iter = c1.begin( );
   cout << "The first element of c1 is now " << *c1_Iter << endl;

   // The following line would be an error because iterator is const
   // *c1_cIter = 200;
}
```

Output

```
The first element of c1 is 1
The first element of c1 is now 20
```

cbegin

Retorna um iterador **const** que trata o primeiro elemento no intervalo.

C++

```
const_iterator cbegin() const;
```

Valor de Devolução

Um iterador de acesso bidirecional **const** que aponta o primeiro elemento do intervalo ou o local logo após o fim de um intervalo vazio (para um intervalo vazio, `cbegin() == cend()`).

Comentários

Com o valor de retorno `cbegin`, os elementos do intervalo não podem ser modificados.

Você pode usar essa função membro no lugar da função membro `begin()`, de modo a garantir que o valor de retorno seja `const_iterator`. Normalmente, é usada em conjunto com a palavra-chave de dedução de tipo auto, conforme mostrado no exemplo a seguir. No exemplo, considere `Container` como um contêiner modificável (não **const**) de qualquer tipo, que dá suporte para `begin()` e `cbegin()`.

C++

```
auto i1 = Container.begin();
// i1 is Container<T>::iterator
auto i2 = Container.cbegin();

// i2 is Container<T>::const_iterator
```

cend

Retorna um iterador **const** que trata o local logo após o último elemento em um intervalo.

C++

```
const_iterator cend() const;
```

Valor de Devolução

Um iterador de acesso bidirecional **`const`** que aponta para além do fim do intervalo.

Comentários

`cend` é usado para testar se um iterador passou do fim de seu intervalo.

Você pode usar essa função membro no lugar da função membro `end()`, de modo a garantir que o valor de retorno seja `const_iterator`. Normalmente, é usada em conjunto com a palavra-chave de dedução de tipo auto, conforme mostrado no exemplo a seguir. No exemplo, considere `Container` como um contêiner modificável (não **`const`**) de qualquer tipo, que dá suporte para `end()` e `cend()`.

C++

```
auto i1 = Container.end();
// i1 is Container<T>::iterator
auto i2 = Container.cend();

// i2 is Container<T>::const_iterator
```

O valor retornado por `cend` não deve ser desreferenciado.

clear

Apaga todos os elementos de uma lista.

C++

```
void clear();
```

Exemplo

C++

```cpp
// list_clear.cpp
// compile with: /EHsc
#include <list>
#include <iostream>

int main() {
   using namespace std;
   list <int> c1;

   c1.push_back( 10 );
   c1.push_back( 20 );
   c1.push_back( 30 );

   cout << "The size of the list is initially " << c1.size( ) << endl;
   c1.clear( );
   cout << "The size of list after clearing is " << c1.size( ) << endl;
}
```

Output

```
The size of the list is initially 3
The size of list after clearing is 0
```

`const_iterator`

Um tipo que fornece um iterador bidirecional que pode ler um elemento **`const`** em uma lista.

C++

```cpp
typedef implementation-defined const_iterator;
```

Comentários

Um tipo de `const_iterator` não pode ser usado para modificar o valor de um elemento.

Exemplo

Confira o exemplo de back.

`const_pointer`

Fornece um ponteiro para um elemento **`const`** em uma lista.

C++

```
typedef typename Allocator::const_pointer const_pointer;
```

Comentários

Um tipo de `const_pointer` não pode ser usado para modificar o valor de um elemento.

Na maioria dos casos, um iterator deve ser usado para acessar os elementos em um objeto de lista.

`const_reference`

Um tipo que fornece uma referência para um elemento **`const`** armazenado em uma lista para leitura e execução de operações **`const`**.

C++

```
typedef typename Allocator::const_reference const_reference;
```

Comentários

Um tipo de `const_reference` não pode ser usado para modificar o valor de um elemento.

Exemplo

C++

```
// list_const_ref.cpp
// compile with: /EHsc
#include <list>
#include <iostream>

int main( )
{
   using namespace std;
   list <int> c1;

   c1.push_back( 10 );
   c1.push_back( 20 );

   const list <int> c2 = c1;
   const int &i = c2.front( );
```

```
    const int &j = c2.back( );
    cout << "The first element is " << i << endl;
    cout << "The second element is " << j << endl;

    // The following line would cause an error because c2 is const
    // c2.push_back( 30 );
}
```

Output

```
The first element is 10
The second element is 20
```

`const_reverse_iterator`

Um tipo que fornece um iterador bidirecional que pode ler qualquer elemento **`const`** em uma lista.

C++

```
typedef std::reverse_iterator<const_iterator> const_reverse_iterator;
```

Comentários

Um tipo de `const_reverse_iterator` não pode modificar o valor de um elemento e é usado para percorrer a lista em ordem inversa.

Exemplo

Confira o exemplo de rbegin.

`crbegin`

Retorna um iterador const que trata o primeiro elemento em uma lista invertida.

C++

```
const_reverse_iterator rbegin() const;
```

Valor de Devolução

Um iterador bidirecional constante invertido que trata o primeiro elemento em uma lista invertida (ou tratando o que foi o último elemento no `list` não invertido).

Comentários

`crbegin` é usado com uma lista invertida assim como list::begin é usado com um `list`.

Com o valor de retorno de `crbegin`, o objeto não pode ser modificado. list::rbegin pode ser usado para iterar através de uma lista para trás.

Exemplo

C++

```
// list_crbegin.cpp
// compile with: /EHsc
#include <list>
#include <iostream>

int main( )
{
   using namespace std;
   list <int> c1;
   list <int>::const_reverse_iterator c1_crIter;

   c1.push_back( 10 );
   c1.push_back( 20 );
   c1.push_back( 30 );
   c1_crIter = c1.crbegin( );
   cout << "The last element in the list is " << *c1_crIter << "." << endl;
}
```

Output

```
The last element in the list is 30.
```

crend

Retorna um iterador const que trata o local após o último elemento em uma lista invertida.

C++

```
const_reverse_iterator rend() const;
```

Valor de Devolução

Um iterador const bidirecional inverso que aborda a localização que vem após o último elemento em um list invertida (o local que precedeu o primeiro elemento no `list` não invertido).

Comentários

`crend` é usado com uma lista invertida assim como list::end é usado com um `list`.

Com o valor de retorno `crend`, o objeto `list` não pode ser modificado.

`crend` pode ser usado para testar se um iterador inverso alcançou o final de sua `list`.

O valor retornado por `crend` não deve ser desreferenciado.

Exemplo

C++

```
// list_crend.cpp
// compile with: /EHsc
#include <list>
#include <iostream>

int main( )
{
   using namespace std;
   list <int> c1;
   list <int>::const_reverse_iterator c1_crIter;

   c1.push_back( 10 );
   c1.push_back( 20 );
   c1.push_back( 30 );

   c1_crIter = c1.crend( );
   c1_crIter --;  // Decrementing a reverse iterator moves it forward in
                  // the list (to point to the first element here)
   cout << "The first element in the list is: " << *c1_crIter << endl;
}
```

Output

```
The first element in the list is: 10
```

difference_type

Um tipo de inteiro marcado que pode ser usado para representar o número de elementos de uma lista em um intervalo entre os elementos apontados por iteradores.

C++

```
typedef typename Allocator::difference_type difference_type;
```

Comentários

`difference_type` é o tipo retornado ao subtrair ou incrementar por meio de iteradores do contêiner. `difference_type` geralmente é usado para representar o número de elementos no intervalo [`first`, `last`) entre os iteradores `first` e `last`, inclui o elemento apontado por `first` e o intervalo de elementos até, mas sem incluir, o elemento apontado por `last`.

Observe que, embora `difference_type` esteja disponível para todos os iteradores que atendem aos requisitos de um iterador de entrada, que inclui a classe de iteradores bidirecionais com suporte de contêineres reversíveis como conjunto, a subtração entre iteradores só terá suporte de iteradores de acesso aleatório fornecidos por um contêiner de acesso aleatório, como umaClasse vector.

Exemplo

C++

```
// list_diff_type.cpp
// compile with: /EHsc
#include <iostream>
#include <list>
#include <algorithm>

int main( )
{
   using namespace std;

   list <int> c1;
   list <int>::iterator   c1_Iter, c2_Iter;

   c1.push_back( 30 );
   c1.push_back( 20 );
   c1.push_back( 30 );
   c1.push_back( 10 );
   c1.push_back( 30 );
```

```
    c1.push_back( 20 );

    c1_Iter = c1.begin( );
    c2_Iter = c1.end( );

     list <int>::difference_type df_typ1, df_typ2, df_typ3;

    df_typ1 = count( c1_Iter, c2_Iter, 10 );
    df_typ2 = count( c1_Iter, c2_Iter, 20 );
    df_typ3 = count( c1_Iter, c2_Iter, 30 );
    cout << "The number '10' is in c1 collection " << df_typ1 << " times.\n";
    cout << "The number '20' is in c1 collection " << df_typ2 << " times.\n";
    cout << "The number '30' is in c1 collection " << df_typ3 << " times.\n";
}
```

Output

```
The number '10' is in c1 collection 1 times.
The number '20' is in c1 collection 2 times.
The number '30' is in c1 collection 3 times.
```

emplace

Insere um elemento construído no local em uma lista na posição especificada.

C++

```
void emplace(iterator Where, Type&& val);
```

Parâmetros

Where

A posição na list de destino em que o primeiro elemento é inserido.

val

O elemento adicionado ao final da `list`.

Comentários

Se uma exceção for gerada, a `list` permanecerá inalterada e a exceção será emitida novamente.

Exemplo

C++

```
// list_emplace.cpp
// compile with: /EHsc
#include <list>
#include <iostream>
#include <string>

int main( )
{
   using namespace std;
   list <string> c2;
   string str("a");

   c2.emplace(c2.begin(), move( str ) );
   cout << "Moved first element: " << c2.back( ) << endl;
}
```

Output

```
Moved first element: a
```

emplace_back

Adiciona um elemento construído no local ao final de uma lista.

C++

```
void emplace_back(Type&& val);
```

Parâmetros

val

O elemento adicionado ao final da list.

Comentários

Se uma exceção for gerada, a `list` permanecerá inalterada e a exceção será emitida novamente.

Exemplo

C++

```cpp
// list_emplace_back.cpp
// compile with: /EHsc
#include <list>
#include <iostream>
#include <string>

int main( )
{
   using namespace std;
   list <string> c2;
   string str("a");

   c2.emplace_back( move( str ) );
   cout << "Moved first element: " << c2.back( ) << endl;
}
```

Output

```
Moved first element: a
```

emplace_front

Adiciona um elemento construído no local ao início de uma lista.

C++

```cpp
void emplace_front(Type&& val);
```

Parâmetros

val

O elemento adicionado ao início da list.

Comentários

Se uma exceção for gerada, a `list` permanecerá inalterada e a exceção será emitida novamente.

Exemplo

C++

```
// list_emplace_front.cpp
// compile with: /EHsc
#include <list>
#include <iostream>
#include <string>

int main( )
{
   using namespace std;
   list <string> c2;
   string str("a");

   c2.emplace_front( move( str ) );
   cout << "Moved first element: " << c2.front( ) << endl;
}
```

Output

```
Moved first element: a
```

empty

Testa se uma lista está vazia.

C++

```
bool empty() const;
```

Valor de Devolução

`true` se a lista estiver vazia; **`false`** se a lista não estiver vazia.

Exemplo

C++

```
// list_empty.cpp
// compile with: /EHsc
#include <list>
#include <iostream>

int main( )
{
   using namespace std;
   list <int> c1;
```

```
   c1.push_back( 10 );
   if ( c1.empty( ) )
      cout << "The list is empty." << endl;
   else
      cout << "The list is not empty." << endl;
}
```

Output

```
The list is not empty.
```

end

Retorna um iterador que trata o local após o último elemento em uma lista.

C++

```
const_iterator end() const;
iterator end();
```

Valor de Devolução

Retorna um iterador bidirecional que trata o local após o último elemento em uma lista. Se a lista estiver vazia, então, `list::end == list::begin`.

Comentários

`end` é usado para testar se um iterador chegou ao final de sua lista.

Exemplo

C++

```
// list_end.cpp
// compile with: /EHsc
#include <list>
#include <iostream>

int main( )
{
   using namespace std;
   list <int> c1;
   list <int>::iterator c1_Iter;
```

```
    c1.push_back( 10 );
    c1.push_back( 20 );
    c1.push_back( 30 );

    c1_Iter = c1.end( );
    c1_Iter--;
    cout << "The last integer of c1 is " << *c1_Iter << endl;

    c1_Iter--;
*c1_Iter = 400;
    cout << "The new next-to-last integer of c1 is "
         << *c1_Iter << endl;

    // If a const iterator had been declared instead with the line:
    // list <int>::const_iterator c1_Iter;
    // an error would have resulted when inserting the 400

    cout << "The list is now:";
    for ( c1_Iter = c1.begin( ); c1_Iter != c1.end( ); c1_Iter++ )
       cout << " " << *c1_Iter;
}
```

Output

```
The last integer of c1 is 30
The new next-to-last integer of c1 is 400
The list is now: 10 400 30
```

erase

Remove um elemento ou um intervalo de elementos das posições especificadas.

C++

```
iterator erase(iterator Where);
iterator erase(iterator first, iterator last);
```

Parâmetros

Where

Posição do elemento a ser removido da lista.

first

Posição do primeiro elemento removido da lista.

last

Posição além do último elemento removido da lista.

Valor de Devolução

Um iterador bidirecional que designa o primeiro elemento restante além de quaisquer elementos removidos ou um ponteiro para o fim da lista se não houver tal elemento.

Comentários

Nenhuma realocação ocorre, por isso, referências e iteradores tornam-se inválidos somente para os elementos apagados.

`erase` nunca gera uma exceção.

Exemplo

C++

```
// list_erase.cpp
// compile with: /EHsc
#include <list>
#include <iostream>

int main( )
{
   using namespace std;
   list <int> c1;
   list <int>::iterator Iter;

   c1.push_back( 10 );
   c1.push_back( 20 );
   c1.push_back( 30 );
   c1.push_back( 40 );
   c1.push_back( 50 );
   cout << "The initial list is:";
   for ( Iter = c1.begin( ); Iter != c1.end( ); Iter++ )
      cout << " " << *Iter;
   cout << endl;

   c1.erase( c1.begin( ) );
   cout << "After erasing the first element, the list becomes:";
   for ( Iter = c1.begin( ); Iter != c1.end( ); Iter++ )
      cout << " " << *Iter;
   cout << endl;
   Iter = c1.begin( );
   Iter++;
   c1.erase( Iter, c1.end( ) );
```

```
    cout << "After erasing all elements but the first, the list becomes: ";
    for (Iter = c1.begin( ); Iter != c1.end( ); Iter++ )
       cout << " " << *Iter;
    cout << endl;
}
```

Output

```
The initial list is: 10 20 30 40 50
After erasing the first element, the list becomes: 20 30 40 50
After erasing all elements but the first, the list becomes:  20
```

`front`

Retorna uma referência ao primeiro elemento em uma lista.

C++

```
reference front();
const_reference front() const;
```

Valor de Devolução

Se a lista estiver vazia, o retorno será indefinido.

Comentários

Se o valor de retorno de `front` for atribuído a `const_reference`, o objeto de lista não poderá ser modificado. Se o valor retornado de `front` for atribuído à `reference`, o objeto de lista poderá ser modificado.

Quando compilado usando _ITERATOR_DEBUG_LEVEL definido como 1 ou 2, um erro de runtime ocorrerá se você tentar acessar um elemento em uma lista vazia. Consulte Iteradores Verificados para obter mais informações.

Exemplo

C++

```
// list_front.cpp
// compile with: /EHsc
#include <list>
#include <iostream>
```

```
int main() {
   using namespace std;
   list <int> c1;

   c1.push_back( 10 );

   int& i = c1.front();
   const int& ii = c1.front();

   cout << "The first integer of c1 is " << i << endl;
   i++;
   cout << "The first integer of c1 is " << ii << endl;
}
```

Output

```
The first integer of c1 is 10
The first integer of c1 is 11
```

get_allocator

Retorna uma cópia do objeto alocador usado para construir uma lista.

C++

```
Allocator get_allocator() const;
```

Valor de Devolução

O alocador usado pela lista.

Comentários

Alocadores para a classe de lista especificam como a classe gerencia o armazenamento. Os alocadores padrão fornecidos com as classes de contêiner da Biblioteca Padrão C++ são suficientes para a maioria das necessidades de programação. Gravando e usando sua própria classe de alocador é um tópico avançado do C++.

Exemplo

C++

```
// list_get_allocator.cpp
// compile with: /EHsc
#include <list>
#include <iostream>

int main( )
{
   using namespace std;
   // The following lines declare objects
   // that use the default allocator.
   list <int> c1;
   list <int, allocator<int> > c2 = list <int, allocator<int> >(
allocator<int>( ) );

   // c3 will use the same allocator class as c1
   list <int> c3( c1.get_allocator( ) );

   list<int>::allocator_type xlst = c1.get_allocator( );
   // You can now call functions on the allocator class used by c1
}
```

insert

Insere um elemento ou um número de elementos ou um intervalo de elementos em uma lista, na posição especificada.

C++

```
iterator insert(iterator Where, const Type& Val);
iterator insert(iterator Where, Type&& Val);

void insert(iterator Where, size_type Count, const Type& Val);
iterator insert(iterator Where, initializer_list<Type> IList);

template <class InputIterator>
void insert(iterator Where, InputIterator First, InputIterator Last);
```

Parâmetros

Where

A posição na lista de destino em que o primeiro elemento é inserido.

Val

O valor do elemento sendo inserido na lista.

Count

O número de elementos sendo inseridos na lista.

First

A posição do primeiro elemento no intervalo de elementos na lista de argumentos a ser copiada.

Last

A posição do primeiro elemento além do intervalo de elementos na lista de argumentos a ser copiada.

Valor de Devolução

As duas primeiras funções de entrada retornam um iterador que aponta para a posição onde o novo elemento foi inserido na lista.

Exemplo

C++

```
// list_class_insert.cpp
// compile with: /EHsc
#include <list>
#include <iostream>
#include <string>

int main()
{
    using namespace std;
    list <int> c1, c2;
    list <int>::iterator Iter;

    c1.push_back(10);
    c1.push_back(20);
    c1.push_back(30);
    c2.push_back(40);
    c2.push_back(50);
    c2.push_back(60);

    cout << "c1 =";
    for (auto c : c1)
        cout << " " << c;
    cout << endl;

    Iter = c1.begin();
    Iter++;
    c1.insert(Iter, 100);
    cout << "c1 =";
```

```
    for (auto c : c1)
        cout << " " << c;
    cout << endl;

    Iter = c1.begin();
    Iter++;
    Iter++;
    c1.insert(Iter, 2, 200);

    cout << "c1 =";
    for(auto c : c1)
        cout << " " << c;
    cout << endl;

    c1.insert(++c1.begin(), c2.begin(), --c2.end());

    cout << "c1 =";
    for (auto c : c1)
        cout << " " << c;
    cout << endl;

    // initialize a list of strings by moving
    list < string > c3;
    string str("a");

    c3.insert(c3.begin(), move(str));
    cout << "Moved first element: " << c3.front() << endl;

    // Assign with an initializer_list
    list <int> c4{ {1, 2, 3, 4} };
    c4.insert(c4.begin(), { 5, 6, 7, 8 });

    cout << "c4 =";
    for (auto c : c4)
        cout << " " << c;
    cout << endl;
}
```

iterator

Um tipo que fornece um iterador bidirecional que pode ler ou modificar qualquer elemento em uma lista.

C++

```
typedef implementation-defined iterator;
```

Comentários

Um tipo de `iterator` pode ser usado para modificar o valor de um elemento.

Exemplo

Confira o exemplo de begin.

`list`

Constrói uma lista de um tamanho específico, ou com elementos de um valor específico, ou com um alocador específico, ou como uma cópia de toda ou parte de alguma outra lista.

C++

```
list();
explicit list(const Allocator& Al);
explicit list(size_type Count);
list(size_type Count, const Type& Val);
list(size_type Count, const Type& Val, const Allocator& Al);

list(const list& Right);
list(list&& Right);
list(initializer_list<Type> IList, const Allocator& Al);

template <class InputIterator>
list(InputIterator First, InputIterator Last);

template <class InputIterator>
list(InputIterator First, InputIterator Last, const Allocator& Al);
```

Parâmetros

`Al`

A classe de alocador a ser usada com esse objeto.

`Count`

O número de elementos na lista construída.

`Val`

O valor dos elementos na lista.

`Right`

A lista da qual a lista construída é uma cópia.

First

A posição do primeiro elemento no intervalo de elementos a serem copiados.

Last

A posição do primeiro elemento além do intervalo de elementos a serem copiados.

IList

O initializer_list que contém os elementos a serem copiados.

Comentários

Todos os construtores armazenam um objeto alocador (*Al*) e iniciam a lista.

get_allocator retorna uma cópia do objeto alocador usado para construir uma lista.

Os primeiros dois construtores especificam uma lista inicial vazia, o segundo especifica o tipo de alocador (*Al*) a ser usado.

O terceiro construtor especifica uma repetição de um número especificado (*Count*) de elementos do valor padrão para a classe `Type`.

O quarto e o quinto construtor especificam uma repetição de elementos *Count* de valor *Val*.

O sexto construtor especifica uma cópia da lista *Right*.

O sétimo construtor move a lista *Right*.

O oitavo construtor usa initializer_list para especificar os elementos.

Os dois construtores seguintes copiam o intervalo `[First, Last)` de uma lista.

Nenhum dos construtores executa realocações provisórias.

Exemplo

C++

```
// list_class_list.cpp
// compile with: /EHsc
#include <list>
#include <iostream>

int main()
{
    using namespace std;
```

```
// Create an empty list c0
list <int> c0;

// Create a list c1 with 3 elements of default value 0
list <int> c1(3);

// Create a list c2 with 5 elements of value 2
list <int> c2(5, 2);

// Create a list c3 with 3 elements of value 1 and with the
// allocator of list c2
list <int> c3(3, 1, c2.get_allocator());

// Create a copy, list c4, of list c2
list <int> c4(c2);

// Create a list c5 by copying the range c4[ first,  last)
list <int>::iterator c4_Iter = c4.begin();
c4_Iter++;
c4_Iter++;
list <int> c5(c4.begin(), c4_Iter);

// Create a list c6 by copying the range c4[ first,  last) and with
// the allocator of list c2
c4_Iter = c4.begin();
c4_Iter++;
c4_Iter++;
c4_Iter++;
list <int> c6(c4.begin(), c4_Iter, c2.get_allocator());

cout << "c1 =";
for (auto c : c1)
    cout << " " << c;
cout << endl;

cout << "c2 =";
for (auto c : c2)
    cout << " " << c;
cout << endl;

cout << "c3 =";
for (auto c : c3)
    cout << " " << c;
cout << endl;

cout << "c4 =";
for (auto c : c4)
    cout << " " << c;
cout << endl;

cout << "c5 =";
for (auto c : c5)
    cout << " " << c;
cout << endl;
```

```
    cout << "c6 =";
    for (auto c : c6)
        cout << " " << c;
    cout << endl;

    // Move list c6 to list c7
    list <int> c7(move(c6));
    cout << "c7 =";
    for (auto c : c7)
        cout << " " << c;
    cout << endl;

    // Construct with initializer_list
    list<int> c8({ 1, 2, 3, 4 });
    cout << "c8 =";
    for (auto c : c8)
        cout << " " << c;
    cout << endl;
}
```

Output

```
c1 = 0 0 0c2 = 2 2 2 2 2c3 = 1 1 1c4 = 2 2 2 2 2c5 = 2 2c6 = 2 2 2c7 = 2 2
2c8 = 1 2 3 4
```

`max_size`

Retorna o tamanho máximo de uma lista.

C++

```
size_type max_size() const;
```

Valor de Devolução

O comprimento máximo possível da lista.

Exemplo

C++

```
// list_max_size.cpp
// compile with: /EHsc
#include <list>
#include <iostream>
```

```
int main( )
{
   using namespace std;
   list <int> c1;
   list <int>::size_type i;

   i = c1.max_size( );
   cout << "Maximum possible length of the list is " << i << "." << endl;
}
```

merge

Remove os elementos da lista de argumentos, insere-os na lista de destino e organiza o conjunto novo e combinado de elementos em ordem crescente ou em alguma outra ordem especificada.

C++

```
void merge(list<Type, Allocator>& right);

template <class Traits>
void merge(list<Type, Allocator>& right, Traits comp);
```

Parâmetros

right

A lista de argumentos a ser mesclada com a lista de destino.

comp

O operador de comparação usado para ordenar os elementos da lista de destino.

Comentários

A lista de argumentos *right* é mesclada com a lista de destino.

Listas de argumento e de destino devem ser solicitadas com a mesma relação de comparação pela qual a sequência resultante deve ser organizada. A ordem padrão para a primeira função de membro é ordem crescente. A segunda função de membro impõe a operação de comparação especificada pelo usuário *comp* da classe `Traits`.

Exemplo

C++

```
// list_merge.cpp
// compile with: /EHsc
#include <list>
#include <iostream>

int main( )
{
   using namespace std;
   list <int> c1, c2, c3;
   list <int>::iterator c1_Iter, c2_Iter, c3_Iter;

   c1.push_back( 3 );
   c1.push_back( 6 );
   c2.push_back( 2 );
   c2.push_back( 4 );
   c3.push_back( 5 );
   c3.push_back( 1 );

   cout << "c1 =";
   for ( c1_Iter = c1.begin( ); c1_Iter != c1.end( ); c1_Iter++ )
      cout << " " << *c1_Iter;
   cout << endl;

   cout << "c2 =";
   for ( c2_Iter = c2.begin( ); c2_Iter != c2.end( ); c2_Iter++ )
      cout << " " << *c2_Iter;
   cout << endl;

   c2.merge( c1 );  // Merge c1 into c2 in (default) ascending order
   c2.sort( greater<int>( ) );
   cout << "After merging c1 with c2 and sorting with >: c2 =";
   for ( c2_Iter = c2.begin( ); c2_Iter != c2.end( ); c2_Iter++ )
      cout << " " << *c2_Iter;
   cout << endl;

   cout << "c3 =";
   for ( c3_Iter = c3.begin( ); c3_Iter != c3.end( ); c3_Iter++ )
      cout << " " << *c3_Iter;
   cout << endl;

   c2.merge( c3, greater<int>( ) );
   cout << "After merging c3 with c2 according to the '>' comparison rela-
tion: c2 =";
   for ( c2_Iter = c2.begin( ); c2_Iter != c2.end( ); c2_Iter++ )
      cout << " " << *c2_Iter;
   cout << endl;
}
```

Output

```
c1 = 3 6
c2 = 2 4
```

```
After merging c1 with c2 and sorting with >: c2 = 6 4 3 2
c3 = 5 1
After merging c3 with c2 according to the '>' comparison relation: c2 = 6 5
4 3 2 1
```

operator=

Substitui os elementos da lista por uma cópia de outra lista.

C++

```
list& operator=(const list& right);
list& operator=(list&& right);
```

Parâmetros

right

O list que está sendo copiado no `list`.

Comentários

Após apagar os elementos existentes em uma `list`, o operador copia ou move o conteúdo da *right* para a `list`.

Exemplo

C++

```
// list_operator_as.cpp
// compile with: /EHsc
#include <list>
#include <iostream>

int main( )
{
   using namespace std;
   list<int> v1, v2, v3;
   list<int>::iterator iter;

   v1.push_back(10);
   v1.push_back(20);
   v1.push_back(30);
   v1.push_back(40);
   v1.push_back(50);
```

```
    cout << "v1 = " ;
    for (iter = v1.begin(); iter != v1.end(); iter++)
       cout << *iter << " ";
    cout << endl;

    v2 = v1;
    cout << "v2 = ";
    for (iter = v2.begin(); iter != v2.end(); iter++)
       cout << *iter << " ";
    cout << endl;

// move v1 into v2
    v2.clear();
    v2 = forward< list<int> >(v1);
    cout << "v2 = ";
    for (iter = v2.begin(); iter != v2.end(); iter++)
       cout << *iter << " ";
    cout << endl;
}
```

pointer

Fornece um ponteiro para um elemento em uma lista.

C++

```
typedef typename Allocator::pointer pointer;
```

Comentários

Um tipo de `pointer` pode ser usado para modificar o valor de um elemento.

Na maioria dos casos, um iterator deve ser usado para acessar os elementos em um objeto de lista.

pop_back

Exclui o elemento no final de uma lista.

C++

```
void pop_back();
```

Comentários

O último elemento não deve estar vazio. `pop_back` nunca gera uma exceção.

Exemplo

C++

```
// list_pop_back.cpp
// compile with: /EHsc
#include <list>
#include <iostream>

int main( )
{
   using namespace std;
   list <int> c1;

   c1.push_back( 1 );
   c1.push_back( 2 );
   cout << "The first element is: " << c1.front( ) << endl;
   cout << "The last element is: " << c1.back( ) << endl;

   c1.pop_back( );
   cout << "After deleting the element at the end of the list, "
        "the last element is: " << c1.back( ) << endl;
}
```

Output

```
The first element is: 1
The last element is: 2
After deleting the element at the end of the list, the last element is: 1
```

pop_front

Exclui o elemento no começo de uma lista.

C++

```
void pop_front();
```

Comentários

O primeiro elemento não deve estar vazio. `pop_front` nunca gera uma exceção.

Exemplo

C++

```
// list_pop_front.cpp
// compile with: /EHsc
#include <list>
#include <iostream>

int main( )
{
   using namespace std;
   list <int> c1;

   c1.push_back( 1 );
   c1.push_back( 2 );
   cout << "The first element is: " << c1.front( ) << endl;
   cout << "The second element is: " << c1.back( ) << endl;

   c1.pop_front( );
   cout << "After deleting the element at the beginning of the list, "
        "the first element is: " << c1.front( ) << endl;
}
```

Output

```
The first element is: 1
The second element is: 2
After deleting the element at the beginning of the list, the first element
is: 2
```

push_back

Adiciona um elemento ao fim da lista.

C++

```
void push_back(const Type& val);
void push_back(Type&& val);
```

Parâmetros

val

O elemento adicionado ao final da lista.

Comentários

Se uma exceção for gerada, a lista permanecerá inalterada e a exceção será gerada novamente.

Exemplo

C++

```
// list_push_back.cpp
// compile with: /EHsc
#include <list>
#include <iostream>
#include <string>

int main( )
{
   using namespace std;
   list <int> c1;

   c1.push_back( 1 );
   if ( c1.size( ) != 0 )
      cout << "Last element: " << c1.back( ) << endl;

   c1.push_back( 2 );
   if ( c1.size( ) != 0 )
      cout << "New last element: " << c1.back( ) << endl;

// move initialize a list of strings
   list <string> c2;
   string str("a");

   c2.push_back( move( str ) );
   cout << "Moved first element: " << c2.back( ) << endl;
}
```

Output

```
Last element: 1
New last element: 2
Moved first element: a
```

push_front

Adiciona um elemento ao começo da lista.

C++

```
void push_front(const Type& val);
void push_front(Type&& val);
```

Parâmetros

val

O elemento adicionado ao início da lista.

Comentários

Se uma exceção for gerada, a lista permanecerá inalterada e a exceção será gerada novamente.

Exemplo

C++

```
// list_push_front.cpp
// compile with: /EHsc
#include <list>
#include <iostream>
#include <string>

int main( )
{
   using namespace std;
   list <int> c1;

   c1.push_front( 1 );
   if ( c1.size( ) != 0 )
      cout << "First element: " << c1.front( ) << endl;

   c1.push_front( 2 );
   if ( c1.size( ) != 0 )
      cout << "New first element: " << c1.front( ) << endl;

// move initialize a list of strings
   list <string> c2;
   string str("a");

   c2.push_front( move( str ) );
   cout << "Moved first element: " << c2.front( ) << endl;
}
```

Output

```
First element: 1
New first element: 2
Moved first element: a
```

rbegin

Retorna um iterador que aborda o primeiro elemento em uma lista invertida.

C++

```
const_reverse_iterator rbegin() const;
reverse_iterator rbegin();
```

Valor de Devolução

Um iterador bidirecional invertido que aborda o primeiro elemento em uma lista invertida (ou que aborda o último elemento na lista não invertida).

Comentários

`rbegin` é usado com uma lista invertida assim como begin é usado com uma lista.

Se o valor de retorno de `rbegin` for atribuído a `const_reverse_iterator`, o objeto de lista não poderá ser modificado. Se o valor retornado de `rbegin` for atribuído à `reverse_iterator`, o objeto de lista poderá ser modificado.

`rbegin` pode ser usado para iterar através de uma lista para trás.

Exemplo

C++

```
// list_rbegin.cpp
// compile with: /EHsc
#include <list>
#include <iostream>

int main( )
{
   using namespace std;
   list <int> c1;
   list <int>::iterator c1_Iter;
   list <int>::reverse_iterator c1_rIter;
```

```
   // If the following line replaced the line above, *c1_rIter = 40;
   // (below) would be an error
   //list <int>::const_reverse_iterator c1_rIter;

   c1.push_back( 10 );
   c1.push_back( 20 );
   c1.push_back( 30 );
   c1_rIter = c1.rbegin( );
   cout << "The last element in the list is " << *c1_rIter << "." << endl;

   cout << "The list is:";
   for ( c1_Iter = c1.begin( ); c1_Iter != c1.end( ); c1_Iter++ )
      cout << " " << *c1_Iter;
   cout << endl;

   // rbegin can be used to start an iteration through a list in
   // reverse order
   cout << "The reversed list is:";
   for ( c1_rIter = c1.rbegin( ); c1_rIter != c1.rend( ); c1_rIter++ )
      cout << " " << *c1_rIter;
   cout << endl;

   c1_rIter = c1.rbegin( );
*c1_rIter = 40;
   cout << "The last element in the list is now " << *c1_rIter << "." <<
endl;
}
```

Output

```
The last element in the list is 30.
The list is: 10 20 30
The reversed list is: 30 20 10
The last element in the list is now 40.
```

reference

Um tipo que fornece uma referência a um elemento armazenado em uma lista.

C++

```
typedef typename Allocator::reference reference;
```

Exemplo

C++

```cpp
// list_ref.cpp
// compile with: /EHsc
#include <list>
#include <iostream>

int main( )
{
   using namespace std;
   list <int> c1;

   c1.push_back( 10 );
   c1.push_back( 20 );

   int &i = c1.front( );
   int &j = c1.back( );
   cout << "The first element is " << i << endl;
   cout << "The second element is " << j << endl;
}
```

Output

```
The first element is 10
The second element is 20
```

remove

Apaga elementos em uma lista que correspondem a um valor especificado.

C++

```cpp
void remove(const Type& val);
```

Parâmetros

val

O valor que, se mantido por um elemento, resultará na remoção de tal elemento da lista.

Comentários

A ordem dos elementos restantes não é afetada.

Exemplo

C++

```
// list_remove.cpp
// compile with: /EHsc
#include <list>
#include <iostream>

int main( )
{
   using namespace std;
   list <int> c1;
   list <int>::iterator c1_Iter, c2_Iter;

   c1.push_back( 5 );
   c1.push_back( 100 );
   c1.push_back( 5 );
   c1.push_back( 200 );
   c1.push_back( 5 );
   c1.push_back( 300 );

   cout << "The initial list is c1 =";
   for ( c1_Iter = c1.begin( ); c1_Iter != c1.end( ); c1_Iter++ )
      cout << " " << *c1_Iter;
   cout << endl;

   list <int> c2 = c1;
   c2.remove( 5 );
   cout << "After removing elements with value 5, the list becomes c2 =";
   for ( c2_Iter = c2.begin( ); c2_Iter != c2.end( ); c2_Iter++ )
      cout << " " << *c2_Iter;
   cout << endl;
}
```

Output

```
The initial list is c1 = 5 100 5 200 5 300
After removing elements with value 5, the list becomes c2 = 100 200 300
```

remove_if

Apaga os elementos da lista para a qual um predicado especificado foi atendido.

C++

```
template <class Predicate>
void remove_if(Predicate pred)
```

Parâmetros

pred

O predicado unário que, se atendido por um elemento, resultará na exclusão de tal elemento da lista.

Exemplo

C++

```
// list_remove_if.cpp
// compile with: /EHsc
#include <list>
#include <iostream>

template <class T> class is_odd : public std::unary_function<T, bool>
{
public:
   bool operator( ) ( T& val )
   {
   return ( val % 2 ) == 1;
   }
};

int main( )
{
   using namespace std;
   list <int> c1;
   list <int>::iterator c1_Iter, c2_Iter;

   c1.push_back( 3 );
   c1.push_back( 4 );
   c1.push_back( 5 );
   c1.push_back( 6 );
   c1.push_back( 7 );
   c1.push_back( 8 );

   cout << "The initial list is c1 =";
   for ( c1_Iter = c1.begin( ); c1_Iter != c1.end( ); c1_Iter++ )
      cout << " " << *c1_Iter;
   cout << endl;

   list <int> c2 = c1;
   c2.remove_if( is_odd<int>( ) );

   cout << "After removing the odd elements, "
        << "the list becomes c2 =";
   for ( c2_Iter = c2.begin( ); c2_Iter != c2.end( ); c2_Iter++ )
      cout << " " << *c2_Iter;
```

```
    cout << endl;
}
```

Output

```
The initial list is c1 = 3 4 5 6 7 8
After removing the odd elements, the list becomes c2 = 4 6 8
```

rend

Retorna um iterador que aborda o local após o último elemento em uma lista invertida.

C++

```
const_reverse_iterator rend() const;
reverse_iterator rend();
```

Valor de Devolução

Um iterador bidirecional inverso que aborda a localização que vem após o último elemento em uma lista invertida (o local que precedeu o primeiro elemento na lista não invertida).

Comentários

`rend` é usado com uma lista invertida assim como end é usado com uma lista.

Se o valor de retorno de `rend` for atribuído a `const_reverse_iterator`, o objeto de lista não poderá ser modificado. Se o valor retornado de `rend` for atribuído à `reverse_iterator`, o objeto de lista poderá ser modificado.

`rend` pode ser usado para testar se um iterador inverso alcançou o final de sua lista.

O valor retornado por `rend` não deve ser desreferenciado.

Exemplo

C++

```
// list_rend.cpp
// compile with: /EHsc
#include <list>
```

```
#include <iostream>

int main( )
{
   using namespace std;
   list <int> c1;
   list <int>::iterator c1_Iter;
   list <int>::reverse_iterator c1_rIter;

   // If the following line had replaced the line above, an error would
   // have resulted in the line modifying an element (commented below)
   // because the iterator would have been const
   // list <int>::const_reverse_iterator c1_rIter;

   c1.push_back( 10 );
   c1.push_back( 20 );
   c1.push_back( 30 );

   c1_rIter = c1.rend( );
   c1_rIter --;  // Decrementing a reverse iterator moves it forward in
                 // the list (to point to the first element here)
   cout << "The first element in the list is: " << *c1_rIter << endl;

   cout << "The list is:";
   for ( c1_Iter = c1.begin( ); c1_Iter != c1.end( ); c1_Iter++ )
      cout << " " << *c1_Iter;
   cout << endl;

   // rend can be used to test if an iteration is through all of the
   // elements of a reversed list
   cout << "The reversed list is:";
   for ( c1_rIter = c1.rbegin( ); c1_rIter != c1.rend( ); c1_rIter++ )
      cout << " " << *c1_rIter;
   cout << endl;

   c1_rIter = c1.rend( );
   c1_rIter--;  // Decrementing the reverse iterator moves it backward
                // in the reversed list (to the last element here)

*c1_rIter = 40;  // This modification of the last element would have
                    // caused an error if a const_reverse iterator had
                    // been declared (as noted above)

   cout << "The modified reversed list is:";
   for ( c1_rIter = c1.rbegin( ); c1_rIter != c1.rend( ); c1_rIter++ )
      cout << " " << *c1_rIter;
   cout << endl;
}
```

Output

```
The first element in the list is: 10
The list is: 10 20 30
```

```
The reversed list is: 30 20 10
The modified reversed list is: 30 20 40
```

resize

Especifica um novo tamanho para uma lista.

C++

```
void resize(size_type _Newsize);
void resize(size_type _Newsize, Type val);
```

Parâmetros

_Newsize

O novo tamanho da lista.

val

O valor dos novos elementos a serem adicionados à lista, caso o novo tamanho seja maior que o tamanho original. Se o valor for omitido, os novos elementos receberão o valor padrão para a classe.

Comentários

Se o tamanho da lista for menor que o tamanho solicitado, *_Newsize*, elementos serão adicionados ao vetor até ele atingir o tamanho solicitado.

Se o tamanho da lista for maior que o tamanho solicitado, os elementos mais próximos do final da lista serão excluídos até a lista atingir o tamanho *_Newsize*.

Se o tamanho atual da lista for igual ao tamanho solicitado, nenhuma ação será realizada.

size reflete o tamanho atual da lista.

Exemplo

C++

```
// list_resize.cpp
// compile with: /EHsc
#include <list>
```

```
#include <iostream>

int main( )
{
   using namespace std;
   list <int> c1;

   c1.push_back( 10 );
   c1.push_back( 20 );
   c1.push_back( 30 );

   c1.resize( 4,40 );
   cout << "The size of c1 is " << c1.size( ) << endl;
   cout << "The value of the last element is " << c1.back( ) << endl;

   c1.resize( 5 );
   cout << "The size of c1 is now " << c1.size( ) << endl;
   cout << "The value of the last element is now " << c1.back( ) << endl;

   c1.resize( 2 );
   cout << "The reduced size of c1 is: " << c1.size( ) << endl;
   cout << "The value of the last element is now " << c1.back( ) << endl;
}
```

Output

```
The size of c1 is 4
The value of the last element is 40
The size of c1 is now 5
The value of the last element is now 0
The reduced size of c1 is: 2
The value of the last element is now 20
```

reverse

Reverte a ordem na qual os elementos ocorrem em uma lista.

C++

```
void reverse();
```

Exemplo

C++

```
// list_reverse.cpp
// compile with: /EHsc
```

```cpp
#include <list>
#include <iostream>

int main( )
{
   using namespace std;
   list <int> c1;
   list <int>::iterator c1_Iter;

   c1.push_back( 10 );
   c1.push_back( 20 );
   c1.push_back( 30 );

   cout << "c1 =";
   for ( c1_Iter = c1.begin( ); c1_Iter != c1.end( ); c1_Iter++ )
      cout << " " << *c1_Iter;
   cout << endl;

   c1.reverse( );
   cout << "Reversed c1 =";
   for ( c1_Iter = c1.begin( ); c1_Iter != c1.end( ); c1_Iter++ )
      cout << " " << *c1_Iter;
   cout << endl;
}
```

Output

```
c1 = 10 20 30
Reversed c1 = 30 20 10
```

reverse_iterator

Um tipo que fornece um iterador bidirecional que pode ler ou modificar um elemento em uma lista invertida.

C++

```cpp
typedef std::reverse_iterator<iterator> reverse_iterator;
```

Comentários

Um tipo de `reverse_iterator` é usado para iterar a lista em ordem inversa.

Exemplo

Confira o exemplo de rbegin.

size

Retorna o número de elementos em uma lista.

C++

```
size_type size() const;
```

Valor de Devolução

O comprimento atual da lista.

Exemplo

C++

```
// list_size.cpp
// compile with: /EHsc
#include <list>
#include <iostream>

int main( )
{
   using namespace std;
   list <int> c1;
   list <int>::size_type i;

   c1.push_back( 5 );
   i = c1.size( );
   cout << "List length is " << i << "." << endl;

   c1.push_back( 7 );
   i = c1.size( );
   cout << "List length is now " << i << "." << endl;
}
```

Output

```
List length is 1.
List length is now 2.
```

size_type

Um tipo que conta o número de elementos em uma lista.

C++

```
typedef typename Allocator::size_type size_type;
```

Exemplo

Confira o exemplo de size.

sort

Organiza os elementos de uma lista em ordem crescente ou com respeito a alguma outra ordem especificada pelo usuário.

C++

```
void sort();

template <class Traits>
    void sort(Traits comp);
```

Parâmetros

comp

O operador de comparação usado para ordenar elementos sucessivos.

Comentários

A primeira função de membro coloca os elementos em ordem crescente por padrão.

A função de membro de modelo ordena os elementos de acordo com a operação de comparação *comp* especificada pelo usuário da classe `Traits`.

Exemplo

C++

```
// list_sort.cpp
// compile with: /EHsc
#include <list>
#include <iostream>

int main( )
```

```
{
   using namespace std;
   list <int> c1;
   list <int>::iterator c1_Iter;

   c1.push_back( 20 );
   c1.push_back( 10 );
   c1.push_back( 30 );

   cout << "Before sorting: c1 =";
   for ( c1_Iter = c1.begin( ); c1_Iter != c1.end( ); c1_Iter++ )
      cout << " " << *c1_Iter;
   cout << endl;

   c1.sort( );
   cout << "After sorting c1 =";
   for ( c1_Iter = c1.begin( ); c1_Iter != c1.end( ); c1_Iter++ )
      cout << " " << *c1_Iter;
   cout << endl;

   c1.sort( greater<int>( ) );
   cout << "After sorting with 'greater than' operation, c1 =";
   for ( c1_Iter = c1.begin( ); c1_Iter != c1.end( ); c1_Iter++ )
      cout << " " << *c1_Iter;
   cout << endl;
}
```

Output

```
Before sorting: c1 = 20 10 30
After sorting c1 = 10 20 30
After sorting with 'greater than' operation, c1 = 30 20 10
```

splice

Remove elementos de uma lista de origem e os insere em uma lista de destino.

C++

```
// insert the entire source list
void splice(const_iterator Where, list<Type, Allocator>& Source);
void splice(const_iterator Where, list<Type, Allocator>&& Source);

// insert one element of the source list
void splice(const_iterator Where, list<Type, Allocator>& Source,
const_iterator Iter);
void splice(const_iterator Where, list<Type, Allocator>&& Source,
const_iterator Iter);

// insert a range of elements from the source list
```

```
void splice(const_iterator Where, list<Type, Allocator>& Source,
const_iterator First, const_iterator Last);
void splice(const_iterator Where, list<Type, Allocator>&& Source,
const_iterator First, const_iterator Last);
```

Parâmetros

Where

A posição na lista de destino antes da inserção.

Source

A lista de origem a ser inserida na lista de destino.

Iter

O elemento a ser inserido da lista de origem.

First

O primeiro elemento no intervalo a ser inserido da lista de origem.

Last

A primeira posição além do último elemento no intervalo a ser inserido da lista de origem.

Comentários

O primeiro par de funções de membro insere todos os elementos na lista de origem na lista de destino antes da posição indicada por *Where* e remove todos os elementos da lista de origem. (`&Source` não deve ser igual a **`this`**.)

O segundo par de funções de membro insere o elemento indicado por *Iter* antes da posição na lista de destino indicada por *Where* e remove *Iter* da lista de origem. (Se `Where == Iter || Where == ++Iter`, nenhuma alteração ocorrerá.)

O terceiro par de funções de membro insere o intervalo designado por `[First, Last)` antes do elemento na lista de destino indicada por *Where* e remove esse intervalo de elementos da lista de origem. (Se `&Source == this`, o intervalo `[First, Last)` não deve incluir o elemento apontado por *Where*.)

Se a união no intervalo inserir elementos `N` e `&Source != this` um objeto da classe iterator será incrementado `N` vezes.

Em todos os iteradores de casos, ponteiros ou referências a elementos de união permanecem válidos e são transferidos para o contêiner de destino.

Exemplo

C++

```
// list_splice.cpp
// compile with: /EHsc /W4
#include <list>
#include <iostream>

using namespace std;

template <typename S> void print(const S& s) {
    cout << s.size() << " elements: ";

    for (const auto& p : s) {
        cout << "(" << p << ") ";
    }

    cout << endl;
}

int main()
{
    list<int> c1{10,11};
    list<int> c2{20,21,22};
    list<int> c3{30,31};
    list<int> c4{40,41,42,43};

    list<int>::iterator where_iter;
    list<int>::iterator first_iter;
    list<int>::iterator last_iter;

    cout << "Beginning state of lists:" << endl;
    cout << "c1 = ";
    print(c1);
    cout << "c2 = ";
    print(c2);
    cout << "c3 = ";
    print(c3);
    cout << "c4 = ";
    print(c4);

    where_iter = c2.begin();
    ++where_iter; // start at second element
    c2.splice(where_iter, c1);
    cout << "After splicing c1 into c2:" << endl;
    cout << "c1 = ";
    print(c1);
    cout << "c2 = ";
```

```
    print(c2);

    first_iter = c3.begin();
    c2.splice(where_iter, c3, first_iter);
    cout << "After splicing the first element of c3 into c2:" << endl;
    cout << "c3 = ";
    print(c3);
    cout << "c2 = ";
    print(c2);

    first_iter = c4.begin();
    last_iter = c4.end();
    // set up to get the middle elements
    ++first_iter;
    --last_iter;
    c2.splice(where_iter, c4, first_iter, last_iter);
    cout << "After splicing a range of c4 into c2:" << endl;
    cout << "c4 = ";
    print(c4);
    cout << "c2 = ";
    print(c2);
}
```

Output

```
Beginning state of lists:c1 = 2 elements: (10) (11)c2 = 3 elements: (20)
(21) (22)c3 = 2 elements: (30) (31)c4 = 4 elements: (40) (41) (42) (43)After
splicing c1 into c2:c1 = 0 elements:c2 = 5 elements: (20) (10) (11) (21)
(22)After splicing the first element of c3 into c2:c3 = 1 elements: (31)c2 =
6 elements: (20) (10) (11) (30) (21) (22)After splicing a range of c4 into
c2:c4 = 2 elements: (40) (43)c2 = 8 elements: (20) (10) (11) (30) (41) (42)
(21) (22)
```

swap

Troca os elementos das duas listas.

C++

```
void swap(list<Type, Allocator>& right);
friend void swap(list<Type, Allocator>& left, list<Type, Allocator>& right)
```

Parâmetros

right

A lista que fornece os elementos a serem trocados ou a lista cujos elementos deverão ser trocados com aqueles da lista *left*.

left

A lista cujos elementos serão trocados por aqueles da lista *right*.

Exemplo

C++

```
// list_swap.cpp
// compile with: /EHsc
#include <list>
#include <iostream>

int main( )
{
   using namespace std;
   list <int> c1, c2, c3;
   list <int>::iterator c1_Iter;

   c1.push_back( 1 );
   c1.push_back( 2 );
   c1.push_back( 3 );
   c2.push_back( 10 );
   c2.push_back( 20 );
   c3.push_back( 100 );

   cout << "The original list c1 is:";
   for ( c1_Iter = c1.begin( ); c1_Iter != c1.end( ); c1_Iter++ )
      cout << " " << *c1_Iter;
   cout << endl;

   c1.swap( c2 );

   cout << "After swapping with c2, list c1 is:";
   for ( c1_Iter = c1.begin( ); c1_Iter != c1.end( ); c1_Iter++ )
      cout << " " << *c1_Iter;
   cout << endl;

   swap( c1,c3 );

   cout << "After swapping with c3, list c1 is:";
   for ( c1_Iter = c1.begin( ); c1_Iter != c1.end( ); c1_Iter++ )
      cout << " " << *c1_Iter;
   cout << endl;
}
```

Output

```
The original list c1 is: 1 2 3
After swapping with c2, list c1 is: 10 20
After swapping with c3, list c1 is: 100
```

`unique`

Remove elementos duplicados adjacentes ou elementos adjacentes que satisfazem algum predicado binário da lista.

C++

```
void unique();

template <class BinaryPredicate>
void unique(BinaryPredicate pred);
```

Parâmetros

`pred`
O predicado binário usado para comparar elementos sucessivos.

Comentários

Essa função pressupõe que a lista é classificada, para que todos os elementos duplicados sejam adjacentes. Duplicatas que não sejam adjacentes não serão excluídas.

A primeira função de membro remove todos os elementos que compara igual ao seu elemento anterior.

A segunda função de membro remove todos os elementos que atendem à função de predicado *`pred`* quando comparada com o elemento anterior. É possível usar qualquer um dos objetos de função binários declarados no cabeçalho `<functional>` para o argumento *`pred`* ou criar seus próprios objetos.

Exemplo

C++

```
// list_unique.cpp
// compile with: /EHsc
#include <list>
#include <iostream>

int main( )
{
   using namespace std;
   list <int> c1;
   list <int>::iterator c1_Iter, c2_Iter,c3_Iter;
```

```
    not_equal_to<int> mypred;

    c1.push_back( -10 );
    c1.push_back( 10 );
    c1.push_back( 10 );
    c1.push_back( 20 );
    c1.push_back( 20 );
    c1.push_back( -10 );

    cout << "The initial list is c1 =";
    for ( c1_Iter = c1.begin( ); c1_Iter != c1.end( ); c1_Iter++ )
       cout << " " << *c1_Iter;
    cout << endl;

    list <int> c2 = c1;
    c2.unique( );
    cout << "After removing successive duplicate elements, c2 =";
    for ( c2_Iter = c2.begin( ); c2_Iter != c2.end( ); c2_Iter++ )
       cout << " " << *c2_Iter;
    cout << endl;

    list <int> c3 = c2;
    c3.unique( mypred );
    cout << "After removing successive unequal elements, c3 =";
    for ( c3_Iter = c3.begin( ); c3_Iter != c3.end( ); c3_Iter++ )
       cout << " " << *c3_Iter;
    cout << endl;
}
```

Output

```
The initial list is c1 = -10 10 10 20 20 -10
After removing successive duplicate elements, c2 = -10 10 20 -10
After removing successive unequal elements, c3 = -10 -10
```

value_type

Um tipo que representa o tipo de dados armazenado em uma lista.

C++

```
typedef typename Allocator::value_type value_type;
```

Comentários

`value_type` é um sinônimo do parâmetro de modelo *`Type`*.

Exemplo

C++

```
// list_value_type.cpp
// compile with: /EHsc
#include <list>
#include <iostream>

int main( )
{
   using namespace std;
   list<int>::value_type AnInt;
   AnInt = 44;
   cout << AnInt << endl;
}
```

Output

```
44
```