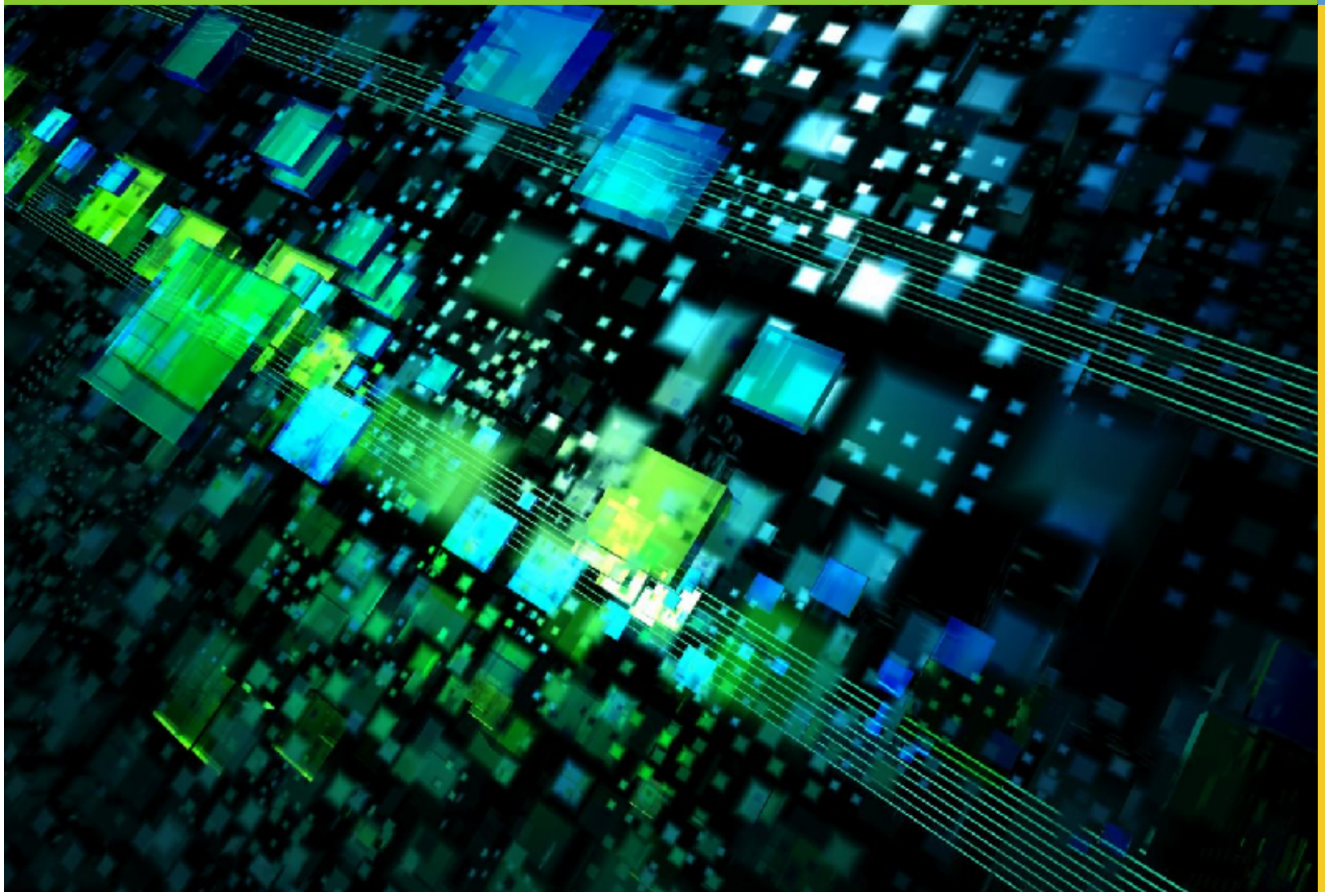
Programação de Computadores
Estruturas de Dados



Alan
de
Freitas

Templates

- São comuns situações em que operações muito parecidas podem ser executadas em diferentes tipos de dados
  - Isto é especialmente verdade entre os tipos de dados aritméticos
- Por exemplo, o algoritmo para encontrar o máximo ou mínimo em um conjunto de `int`s é muito similar ao algoritmo para os tipos de dado `double`, `short`, `float` ou `long`

Considere o código abaixo que encontra o maior elementos entre 3 `ints`

```
int maximo(int a, int b, int c){
    int max = a;
    if (b > max){
        max = b;
    }
    if (c > max){
        max = c;
    }
    return max;
}
```

Veja agora o código que faz o mesmo para dados do tipo double.

```java
int maximo(int a, int b, int c){
    int max = a;
    if (b > max){
        max = b;
    }
    if (c > max){
        max = c;
    }
    return max;
}
```

```java
double maximo(double a, double b, double c){
    double max = a;
    if (b > max){
        max = b;
    }
    if (c > max){
        max = c;
    }
    return max;
}
```

Com exceção do tipo de dado, os códigos são idênticos.

Não é difícil perceber que o mesmo código funcionaria para qualquer outro tipo de dado. Precisaríamos apenas alterar o tipo.

```
int maximo(int a, int b, int c){
    int max = a;
    if (b > max){
        max = b;
    }
    if (c > max){
        max = c;
    }
    return max;
}
```

```
double maximo(double a, double b, double c){
    double max = a;
    if (b > max){
        max = b;
    }
    if (c > max){
        max = c;
    }
    return max;
}
```

```
float maximo(float a, float b, float c){
    float max = a;
    if (b > max){
        max = b;
    }
    if (c > max){
        max = c;
    }
    return max;
}
```

```
char maximo(char a, char b, char c){
    char max = a;
    if (b > max){
        max = b;
    }
    if (c > max){
        max = c;
    }
    return max;
}
```

Já vimos neste curso vários algoritmos que poderiam ser generalizados para qualquer tipo de dado.

```
int maximo(int a, int b, int c){
    int max = a;
    if (b > max){
        max = b;
    }
    if (c > max){
        max = c;
    }
    return max;
}
```

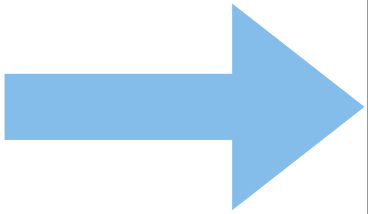
```
double maximo(double a, double b, double c){
    double max = a;
    if (b > max){
        max = b;
    }
    if (c > max){
        max = c;
    }
    return max;
}
```

```
float maximo(float a, float b, float c){
    float max = a;
    if (b > max){
        max = b;
    }
    if (c > max){
        max = c;
    }
    return max;
}
```

```
char maximo(char a, char b, char c){
    char max = a;
    if (b > max){
        max = b;
    }
    if (c > max){
        max = c;
    }
    return max;
}
```

Em C++, os templates nos permitem definir uma função que funciona para mais de um tipo de dado como no exemplo abaixo:

```
template <class T>
T maximo(T a, T b, T c){
    T max = a;
    if (b > max){
        max = b;
    }
    if (c > max){
        max = c;
    }
    return max;
}
```



```
double maximo(double a, double b, double c){
    double max = a;
    if (b > max){
        max = b;
    }
    if (c > max){
        max = c;
    }
    return max;
}
```

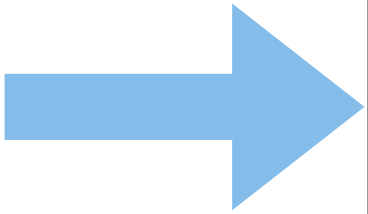
```
char maximo(char a, char b, char c){
    char max = a;
    if (b > max){
        max = b;
    }
    if (c > max){
        max = c;
    }
    return max;
}
```

```
float maximo(float a, float b, float c){
    float max = a;
    if (b > max){
        max = b;
    }
    if (c > max){
        max = c;
    }
    return max;
}
```

```
int maximo(int a, int b, int c){
    int max = a;
    if (b > max){
        max = b;
    }
    if (c > max){
        max = c;
    }
    return max;
}
```

Definir um template equivale a ter definido a função para qualquer tipo de dado possível.

```
template <class T>
T maximo(T a, T b, T c){
    T max = a;
    if (b > max){
        max = b;
    }
    if (c > max){
        max = c;
    }
    return max;
}
```



```
double maximo(double a, double b, double c){
    double max = a;
    if (b > max){
        max = b;
    }
    if (c > max){
        max = c;
    }
    return max;
}
```

```
char maximo(char a, char b, char c){
    char max = a;
    if (b > max){
        max = b;
    }
    if (c > max){
        max = c;
    }
    return max;
}
```

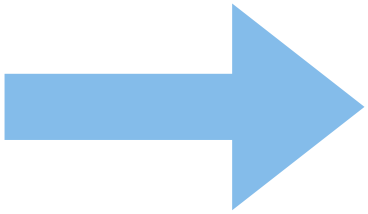
```
float maximo(float a, float b, float c){
    float max = a;
    if (b > max){
        max = b;
    }
    if (c > max){
        max = c;
    }
    return max;
}
```

```
int maximo(int a, int b, int c){
    int max = a;
    if (b > max){
        max = b;
    }
    if (c > max){
        max = c;
    }
    return max;
}
```

Por exemplo, se a função chamadora tem um comando `maximo(4,2,6)`, o compilador gera uma versão da função maximo que aceita `ints`.

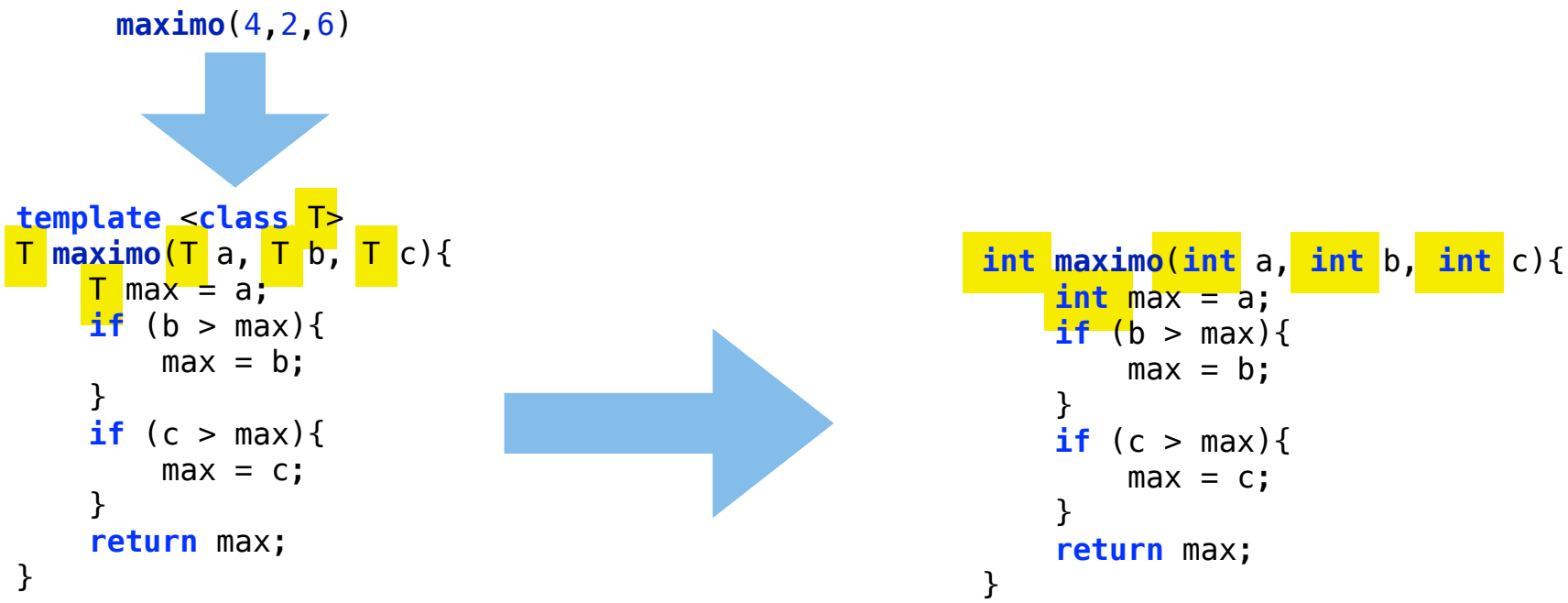
```
maximo(4,2,6)
```

```
template <class T>
T maximo(T a, T b, T c){
    T max = a;
    if (b > max){
        max = b;
    }
    if (c > max){
        max = c;
    }
    return max;
}
```



```
int maximo(int a, int b, int c){
    int max = a;
    if (b > max){
        max = b;
    }
    if (c > max){
        max = c;
    }
    return max;
}
```

Por exemplo, se a função chamadora tem um comando `maximo(4,2,6)`, o compilador gera uma versão da função maximo que aceita `ints`.



Templates

- Representam uma coleção de definições
- Definições são geradas sob demanda pelo compilador
- Recurso poderoso de reutilização de código em C++
- Com eles, conseguimos fazer o que chamamos de **programação genérica**
- Retornaremos a este tópico mais vezes neste curso

Standard Template Library

- Contribuição mais relevante e mais representativa de programação genérica em C++ é a STL (Biblioteca Padrão de Templates)
  - Parte do padrão C++ (aprovado em 1997/1998)
  - Estende o núcleo de C++ fornecendo componentes gerais

Standard Template Library

- Como alguns exemplos, a STL fornece
  - O tipo de dado `string`
  - Diferentes estruturas de dados para armazenamento de dados
  - Classes para entrada/saída
  - Algoritmos utilizados frequentemente

Standard Template Library

A STL se baseia em três partes fundamentais:

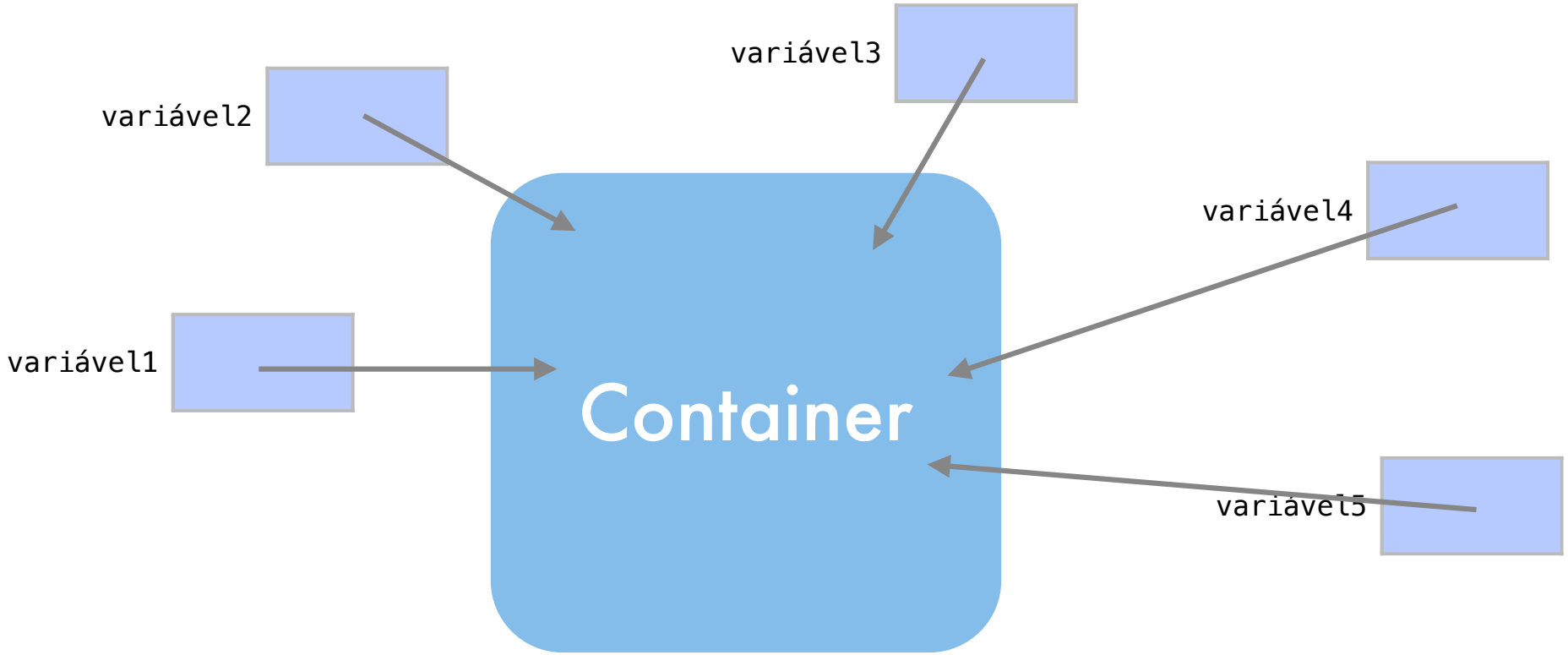
Containers	Gerenciam coleções de objetos
Iteradores	Percorrem elementos das coleções de objetos
Algoritmos	Processam elementos da coleção

Containers

As estruturas de dados são representadas com a STL através de containers.

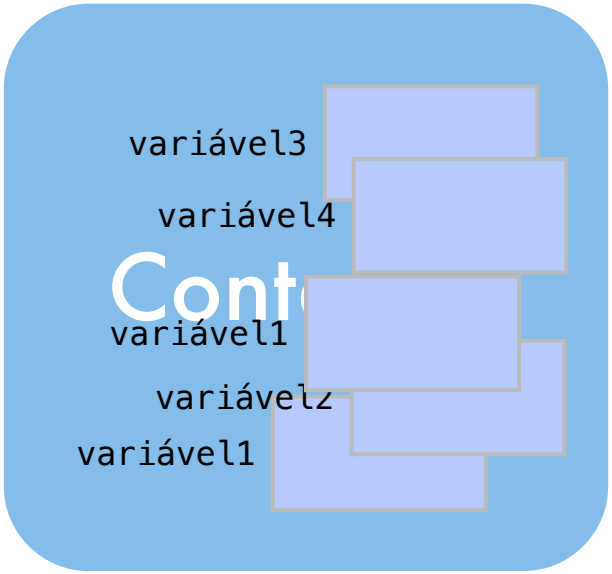
Containers

Como a analogia sugere, dentro de um container, podemos guardar várias variáveis de qualquer tipo.



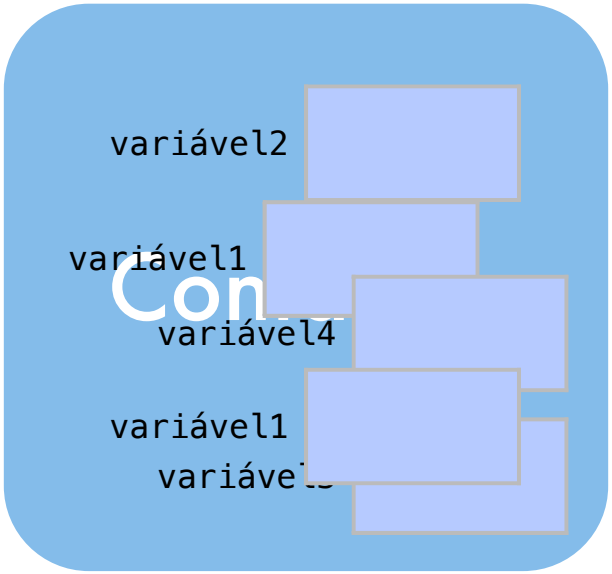
Containers

Porém, cada container tem suas próprias características pois utiliza internamente diferentes estruturas de dados para guardar estas variáveis.



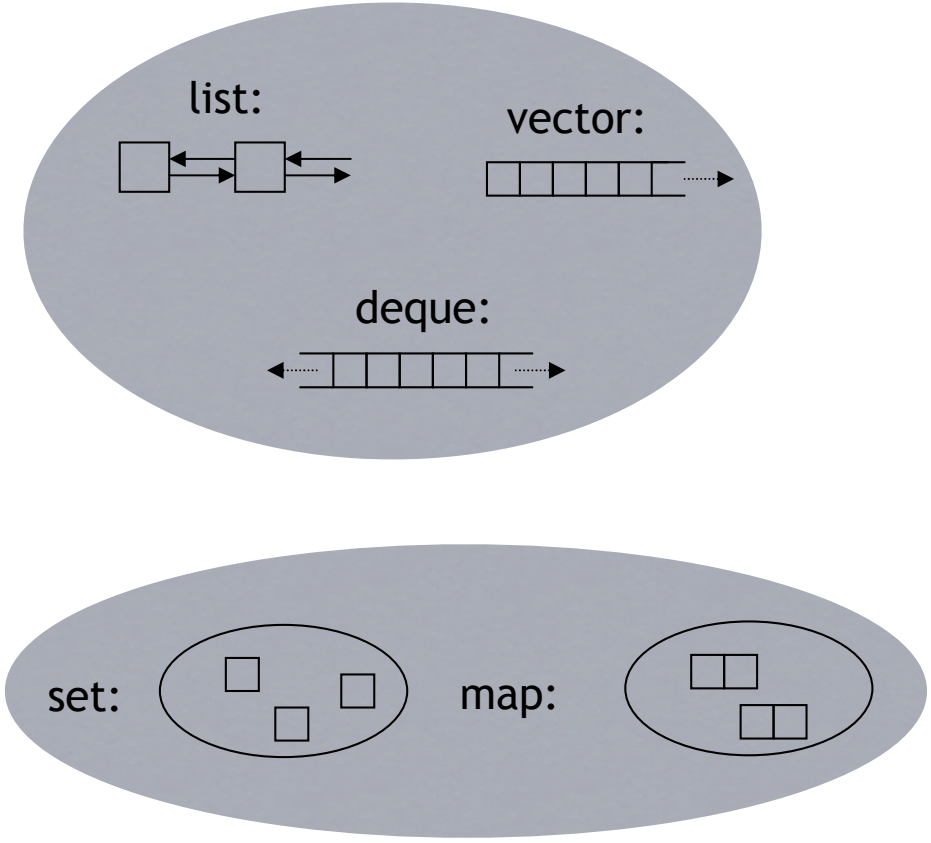
Containers

Ou seja, cada tipo de container possui uma estratégia diferente para organização interna de seus dados. Por isso, cada container possui vantagens e desvantagens em diferentes situações.



Containers

- 2 tipos básicos:
  - Containers sequenciais: representa listas onde cada elemento tem uma posição específica
  - Containers associativos: representa conjuntos onde a posição interna de um elemento depende de seu valor



Containers

- Os containers são representados através de objetos que utilizam templates
- Os templates permitem que o container seja utilizado em qualquer tipo de dado
- Grosso modo, os objetos são recursos similares aos `structs`, porém, além de ter suas próprias variáveis, os objetos possuem suas próprias funções

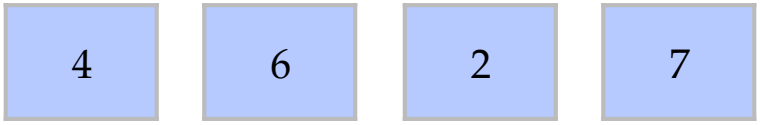
Funções-membro comuns aos contêiners

Se está vazio `empty()`	Tamanho `size()`	Compara `<, >, >=, <=, ==, !=`
Troca `swap()`	Tamanho máximo `max_size()`	Início `begin()`
Fim `end()`	Início (reverse) `rbegin()`	Fim (reverse) `rend()`
Apaga elementos `erase(i)`	Limpa container `clear()`	

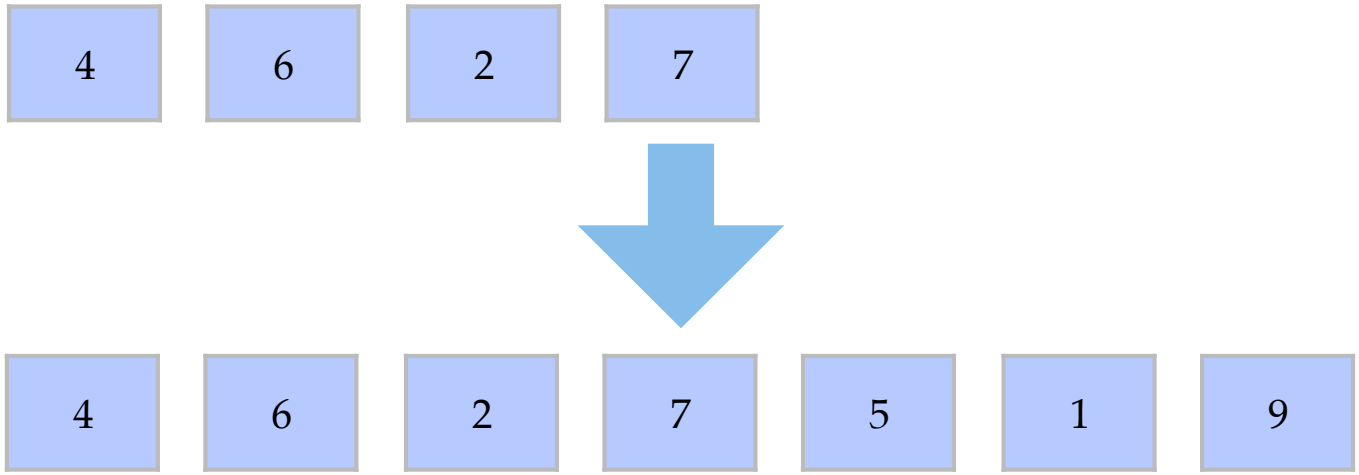
Vector

- O vector é talvez o tipo de container de sequência mais utilizado
- Este container tem funções para acessar, remover ou inserir elementos no fim da sequência de elementos de maneira eficiente
- Exige a inclusão do cabeçalho `<vector>`

Imagine a sequência de elementos abaixo. Imagine que o `vector` com esta sequência se chama `c`.

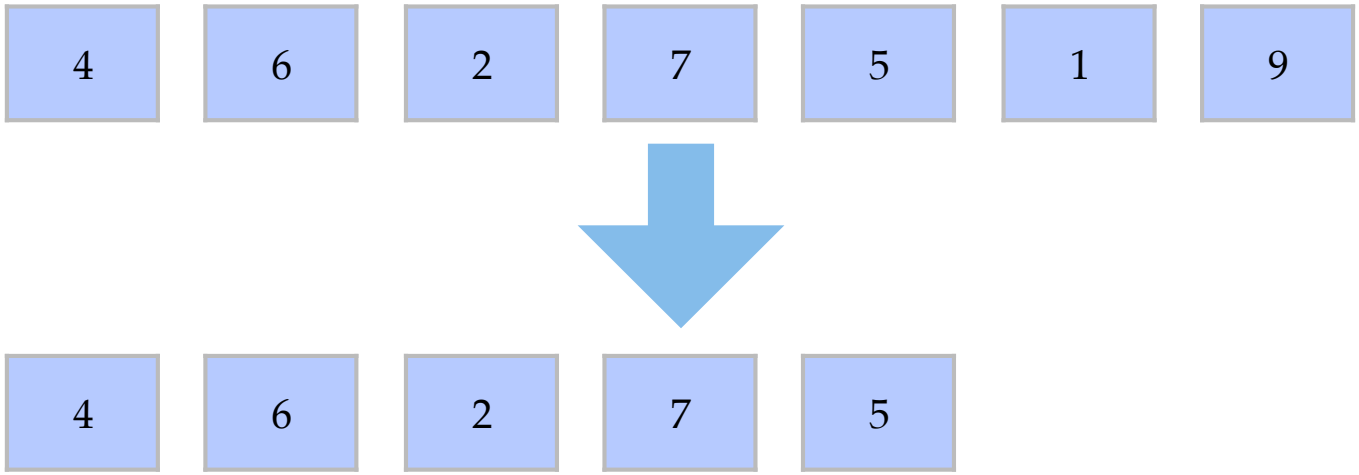


```
c.push_back(5);
c.push_back(1);
c.push_back(9);
```



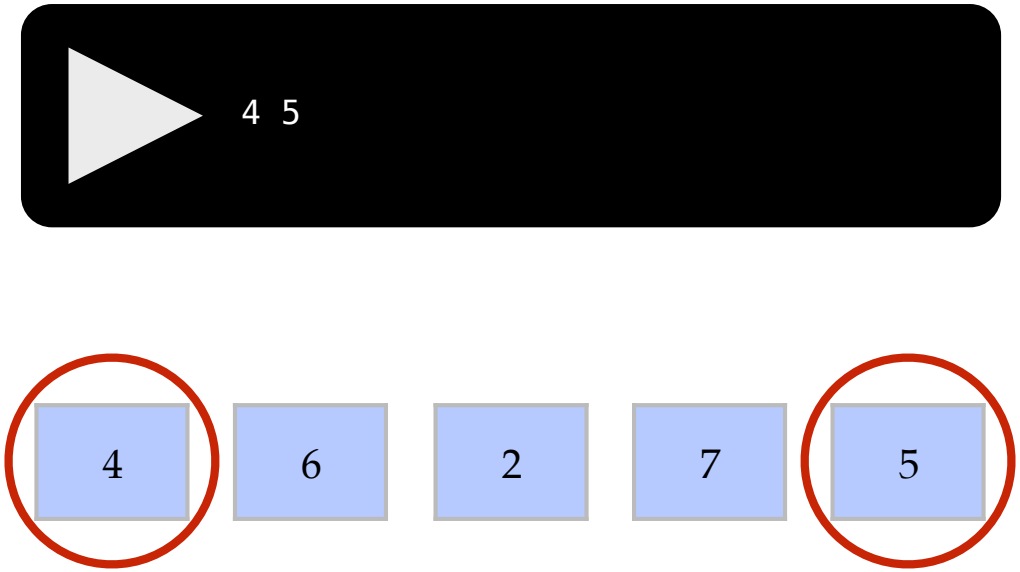
Novos elementos podem ser inseridos ao final de c com a função push_back.

```
c.pop_back();
c.pop_back();
```



Elementos podem ser removidos do final de `c` com a função `pop_back`.

```
cout << c.front() << " " << c.back() << endl;
```



O primeiro e último elementos de `c` podem ser acessados com as funções `front` e `back`.

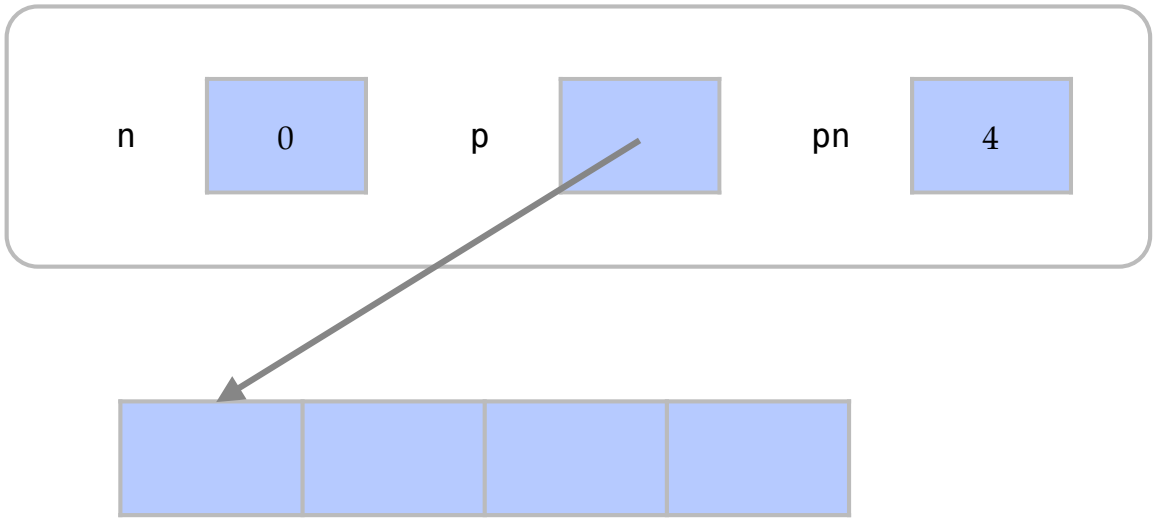
Vector - Como funciona

- Apesar de todas as simplificações oferecidas pelos templates, os containers são complexos internamente
- Todo container é representado internamente através de recursos que já conhecemos, como ponteiros e arranjos
- Este recursos são organizados para formarem uma estrutura de dados para organizar os elementos

```
vector<int> c;
```

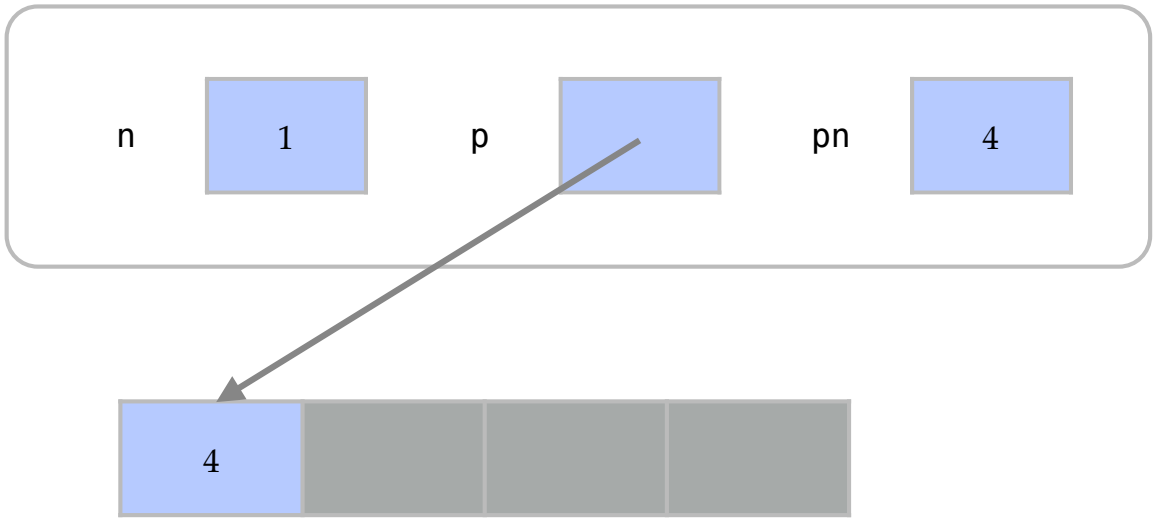
Com este comando, podemos criar um `vector` vazio. Internamente, o vector tem um ponteiro um dois números inteiros.

```
vector<int> c;
```



Um número inteiro guarda o número de elementos no container enquanto o ponteiro aponta para um arranjo de elementos na memória. O segundo número inteiro guarda o tamanho deste arranjo.

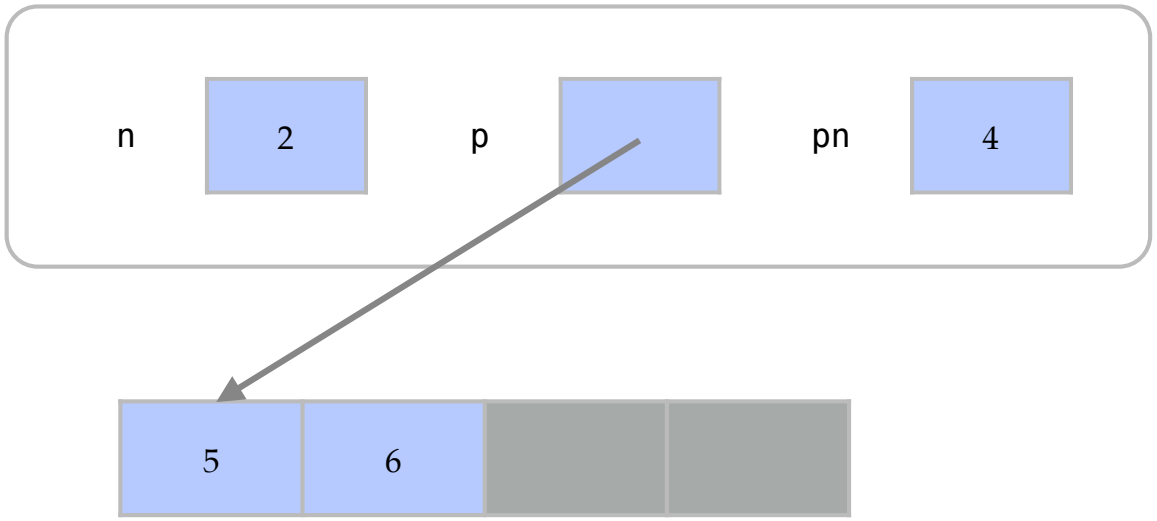
```
vector<int> c;
```




```
c.push_back(4);
```

Quando inserimos um elemento no fim do container, este é inserido na posição n do arranjo e n é incrementado.

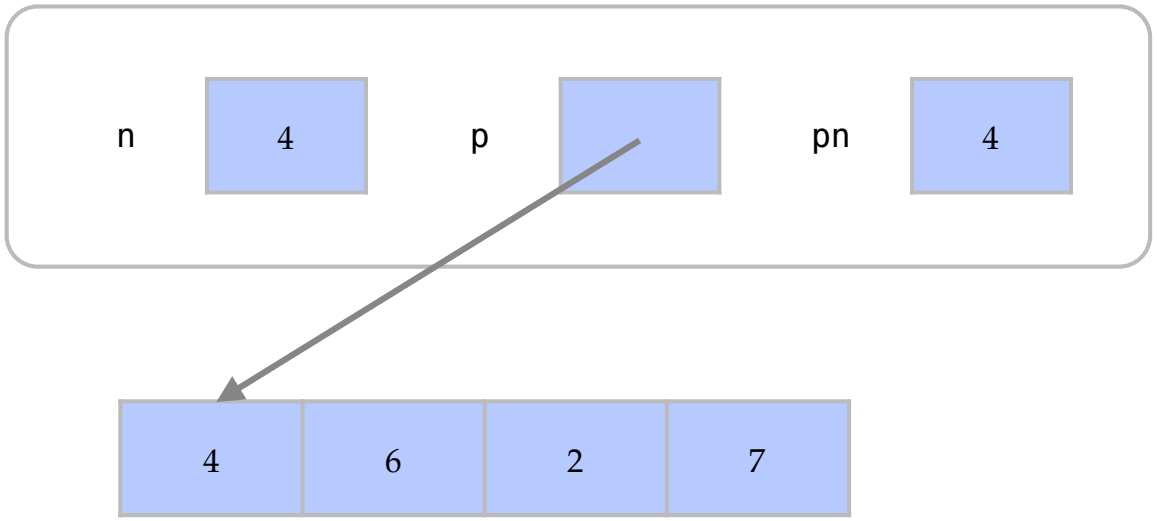
```
vector<int> c;
```




```
c.push_back(6);
```

Como o arranjo já tem espaço para mais que *n* elementos, mais elementos podem ser inseridos sem necessidade de se alocar mais memória.

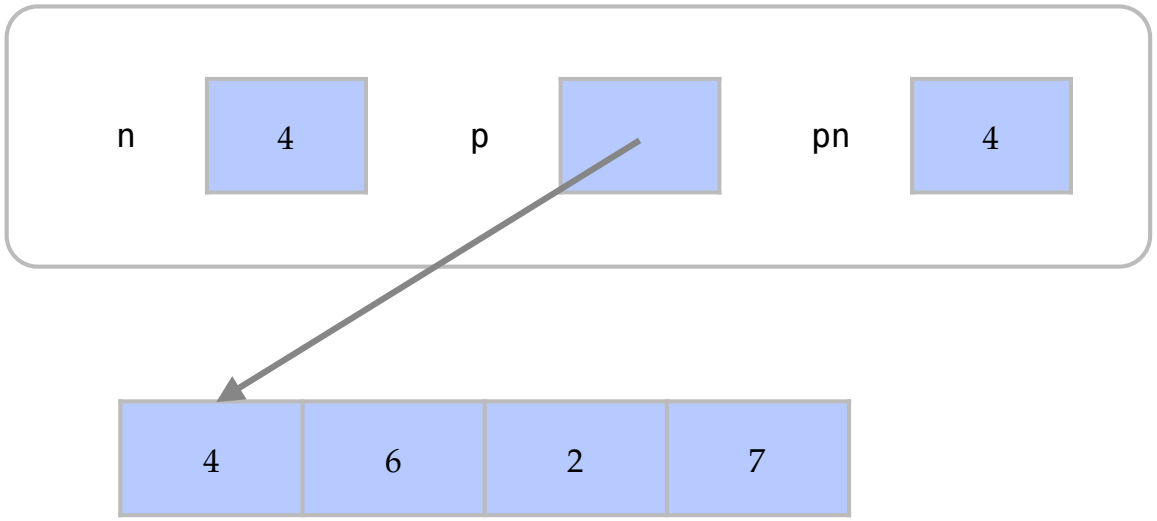
```
vector<int> c;
```




```
c.push_back(2);
c.push_back(7);
```

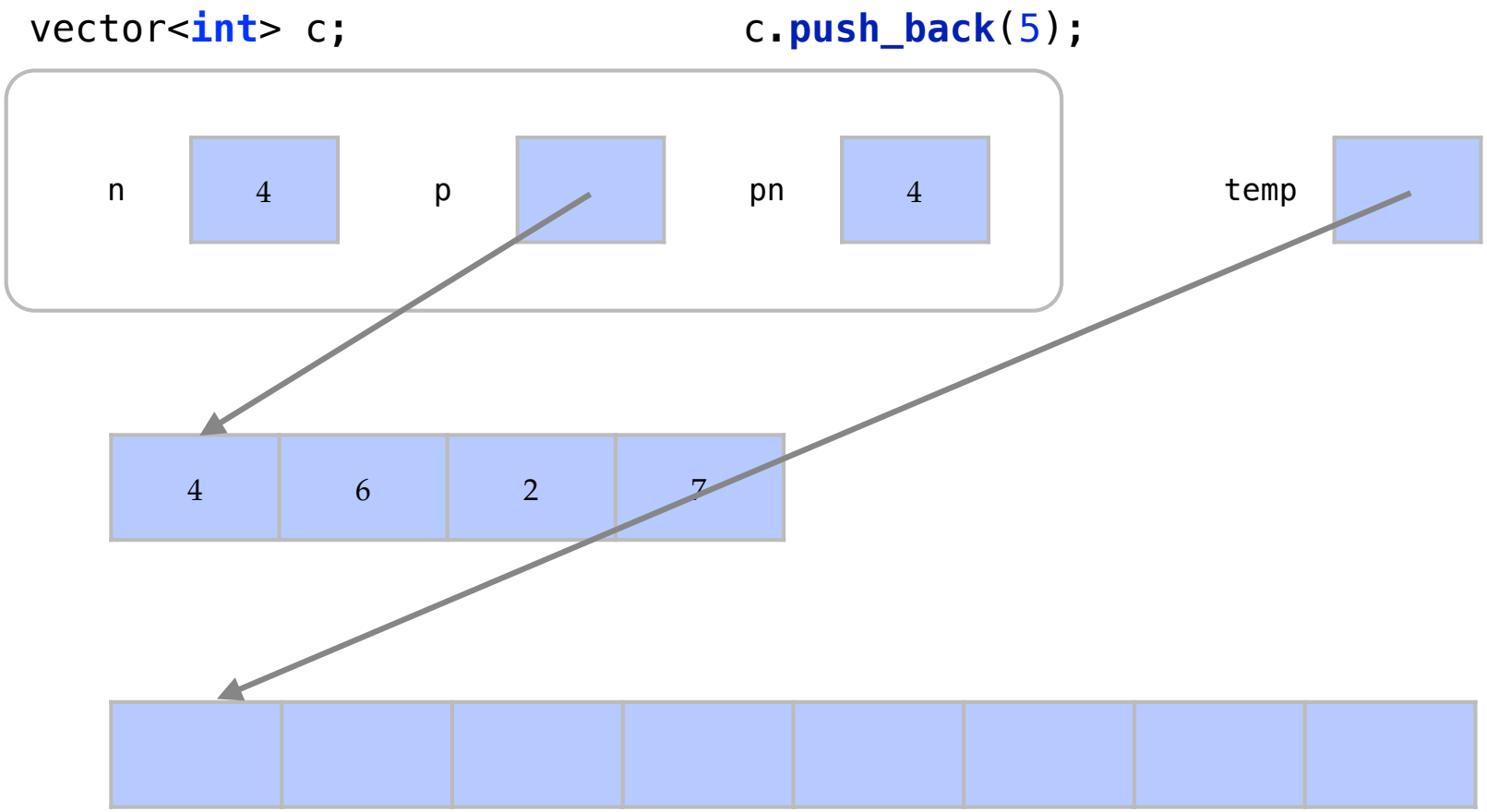
Eventualmente, o número de elementos será igual ao tamanho do arranjo. Neste caso, não podemos colocar mais elementos sem alocar mais memória.

```
vector<int> c;
```

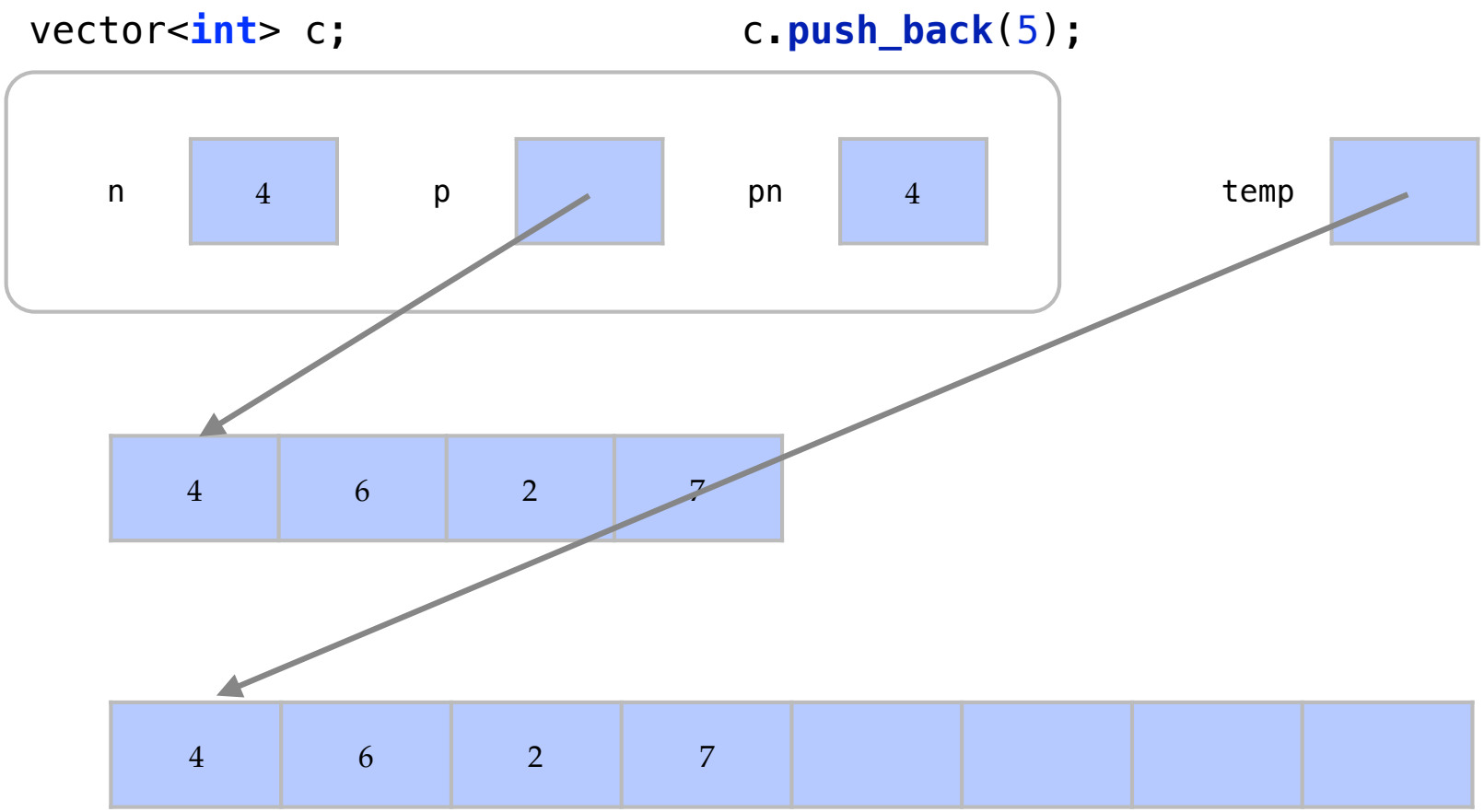



```
c.push_back(5);
```

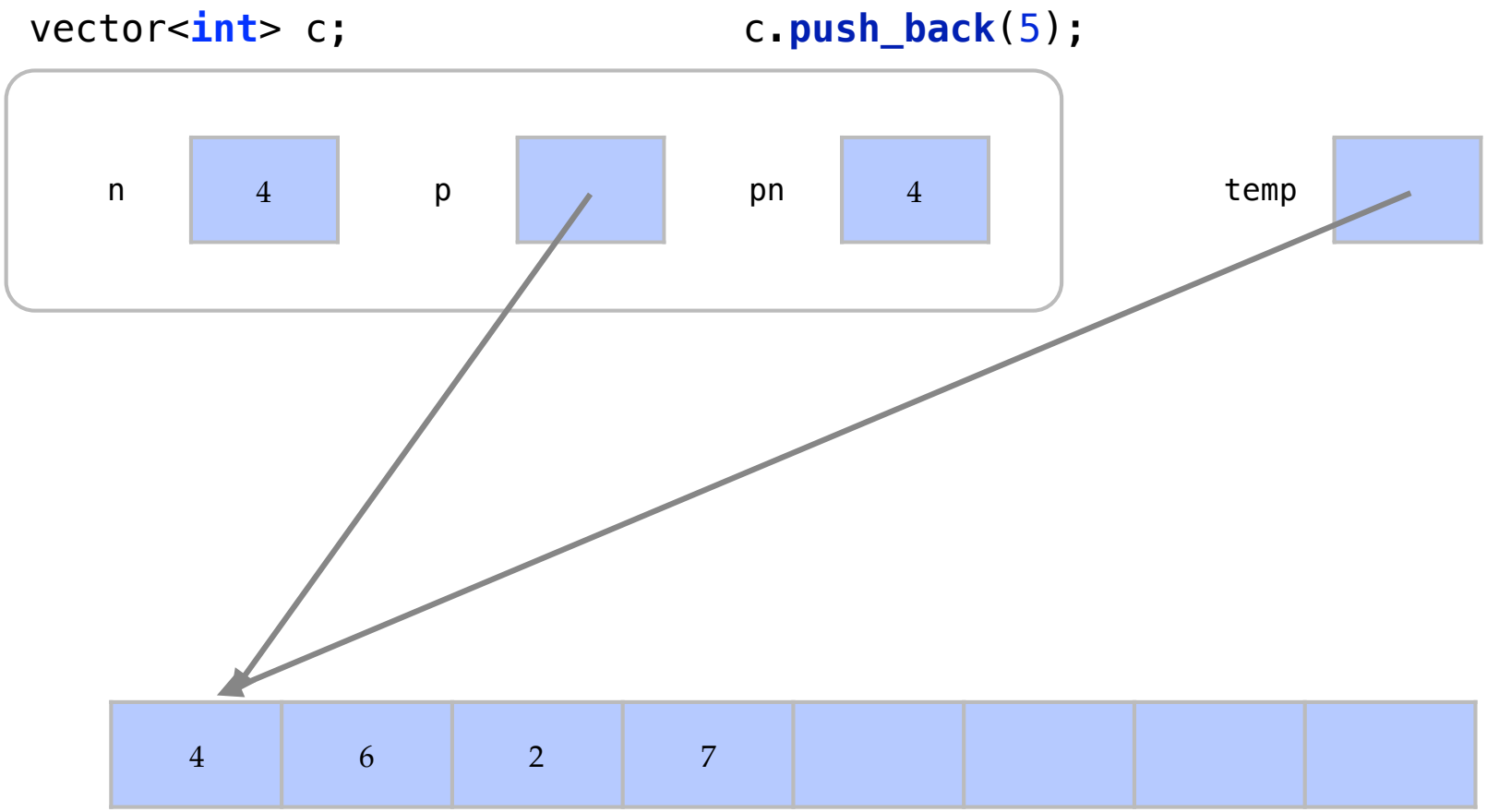
Queremos agora inserir o elemento 5 ao container.
Para isto, ocorrem os seguintes passos...



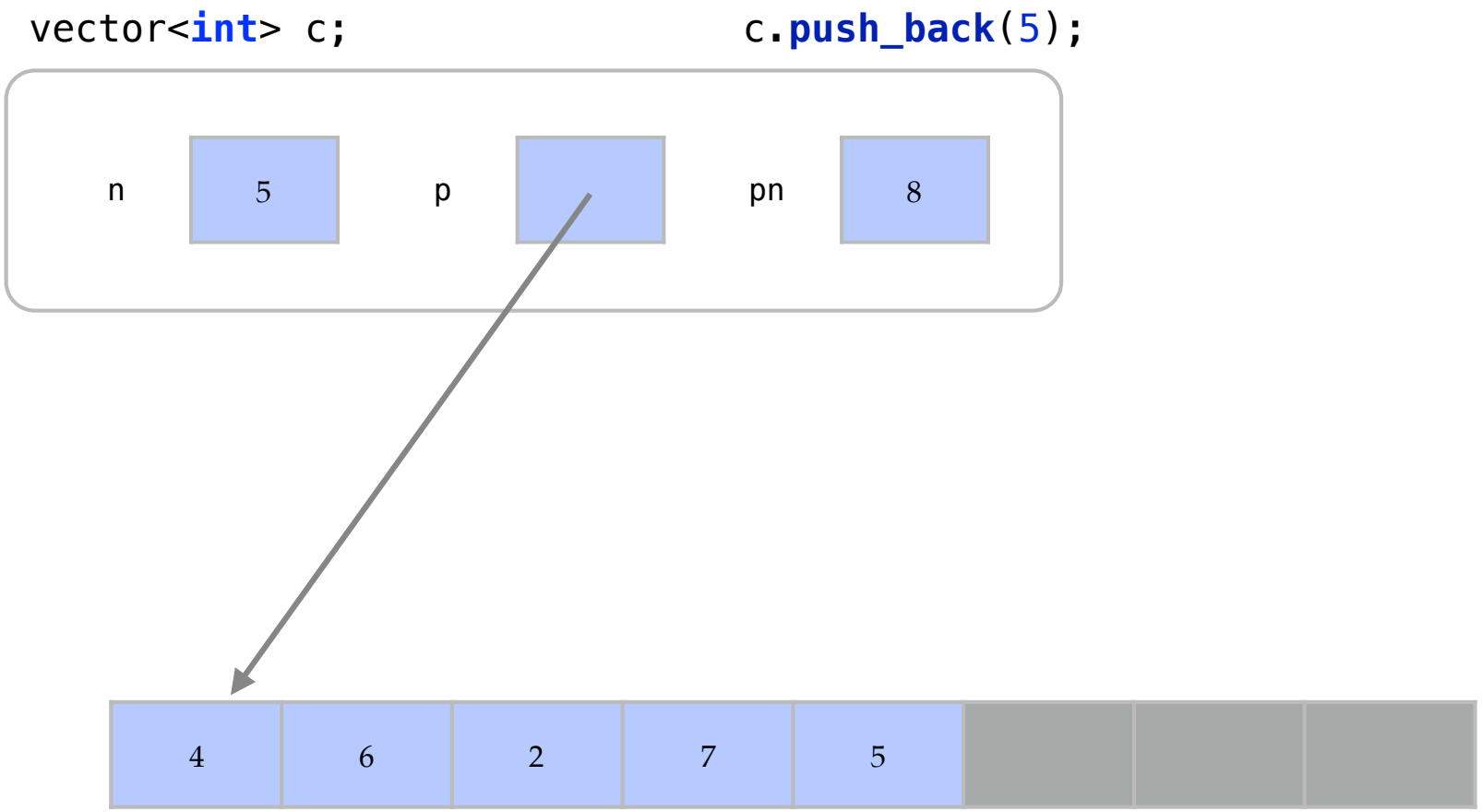
Alocamos espaço para um arranjo com o dobro de elementos e fazemos um ponteiro temporário apontar para ele.



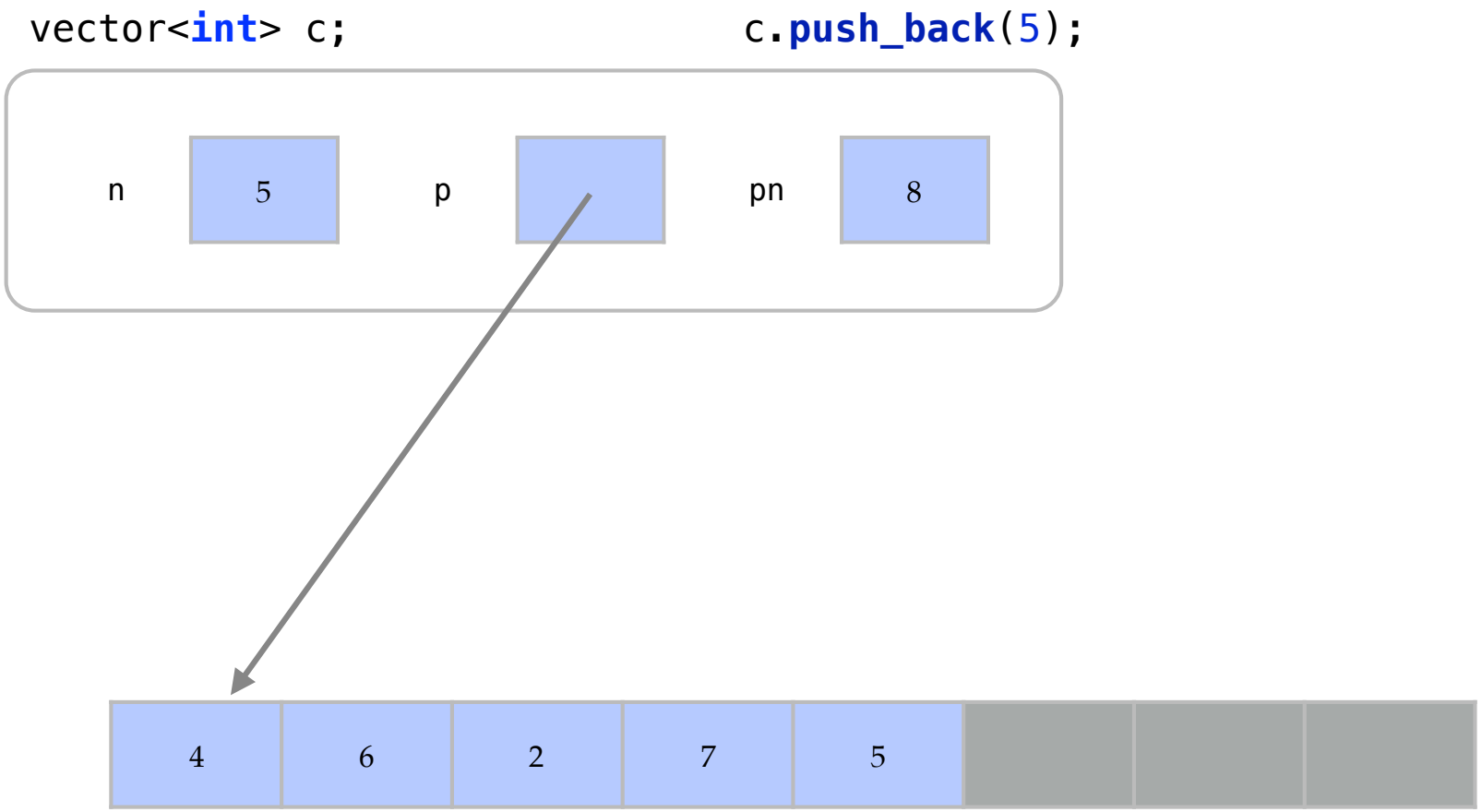
Copiamos todos os elementos de `p` para `temp`. Este passo tem um custo $O(n)$.



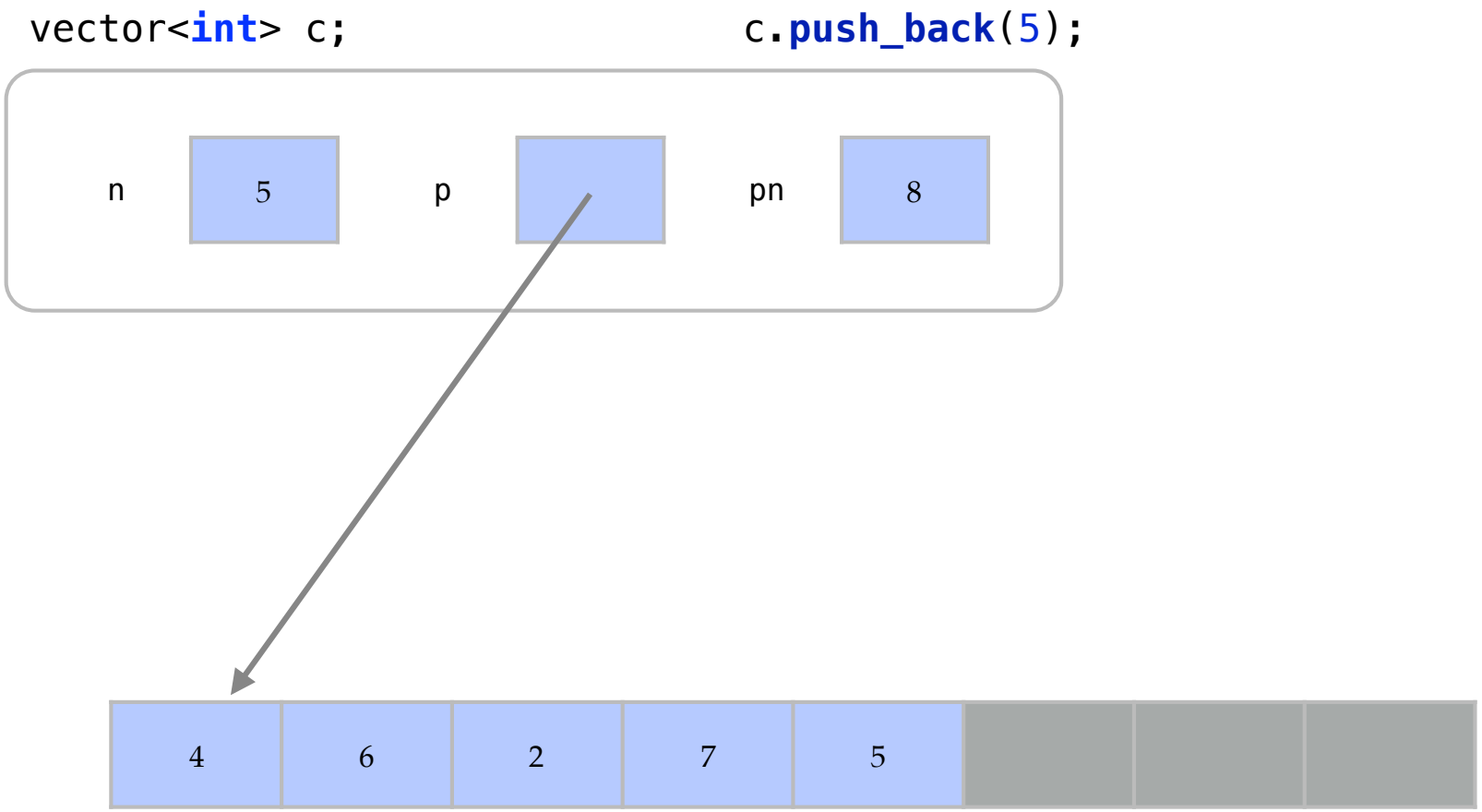
Desalocamos a memória apontada por p e fazemos com que p aponte para o mesmo arranjo de temp.



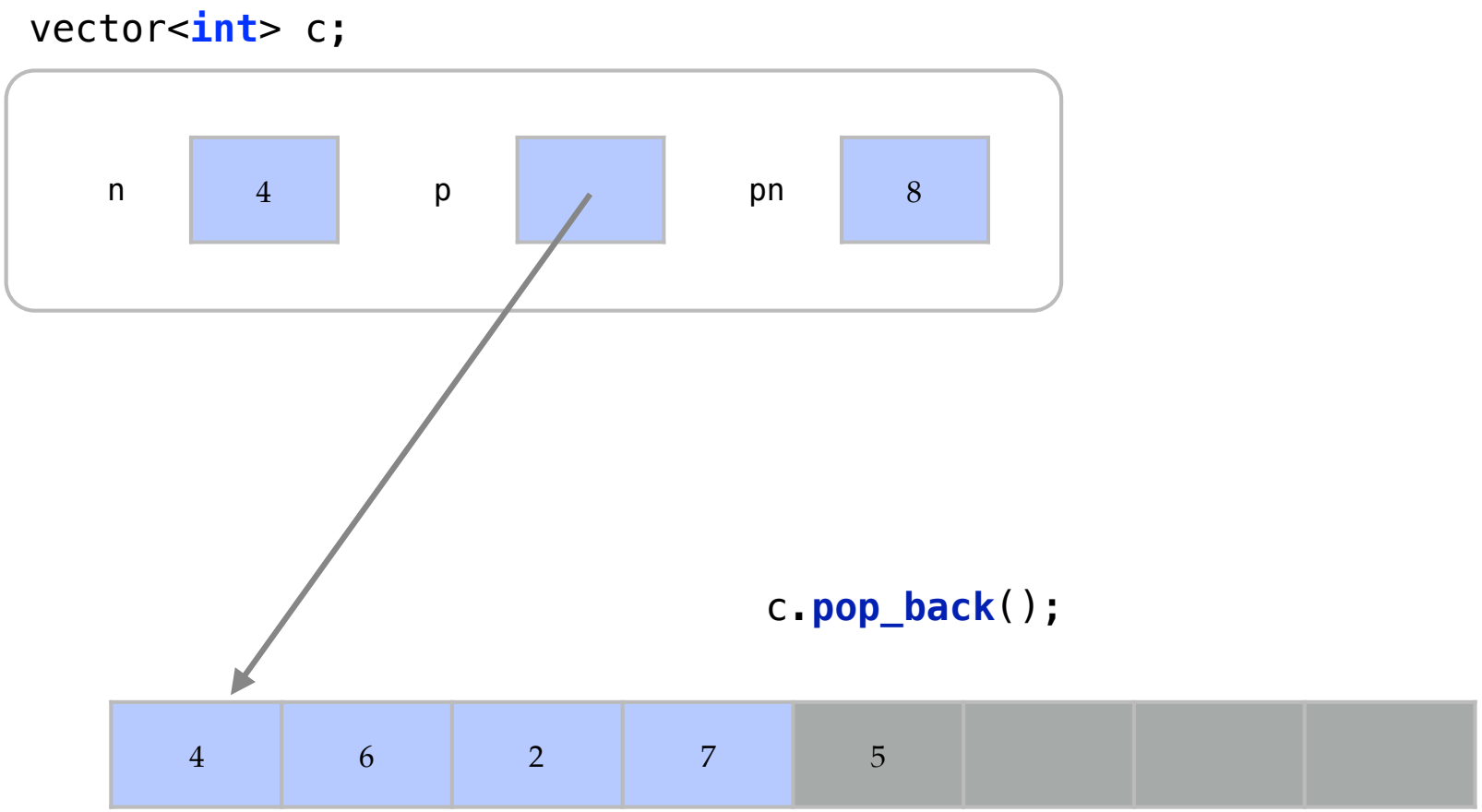
pn é dobrado, n incrementado, o valor 5 é inserido normalmente com custo *O(1)* e `temp` deixa de existir por ter apenas escopo de função.



Contudo, toda a operação de inserção teve custo $O(n)$. Este custo, porém, se amortizado entre várias inserções tem um custo médio de $O(1)$.



Existe inclusive a função `capacity` que retorna quantos elementos ainda podem ser colocados em um `vector` sem se alocar mais memória.

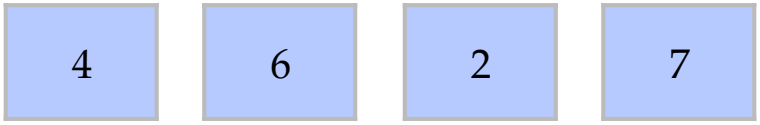


Com custo *O(1)*, a função para remover o último elemento do container é realizada apenas pela atualização do valor de *n*, o fazendo indicar que apenas os *4* primeiros elementos devem ser considerados.

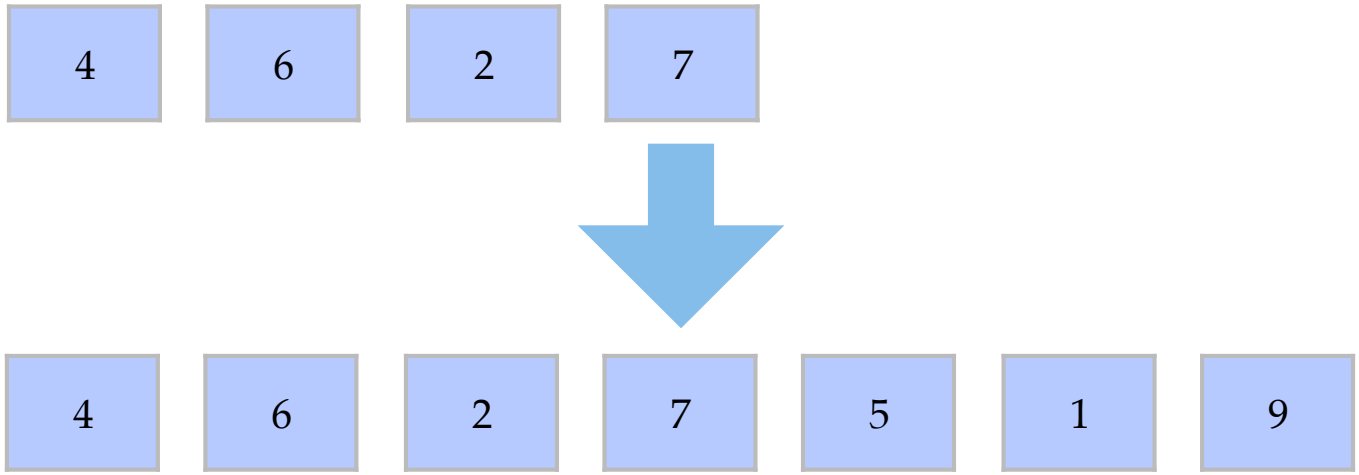
Deque

- Double ended queue (“fila com duas pontas”)
- Indicado para sequências que crescem nas duas direções
- Inserção de elementos no início e no final da sequência é rápida
- Exige a inclusão do cabeçalho `<deque>`

Imagine a sequência de elementos abaixo. Imagine que o deque que contém esta sequência se chama c.

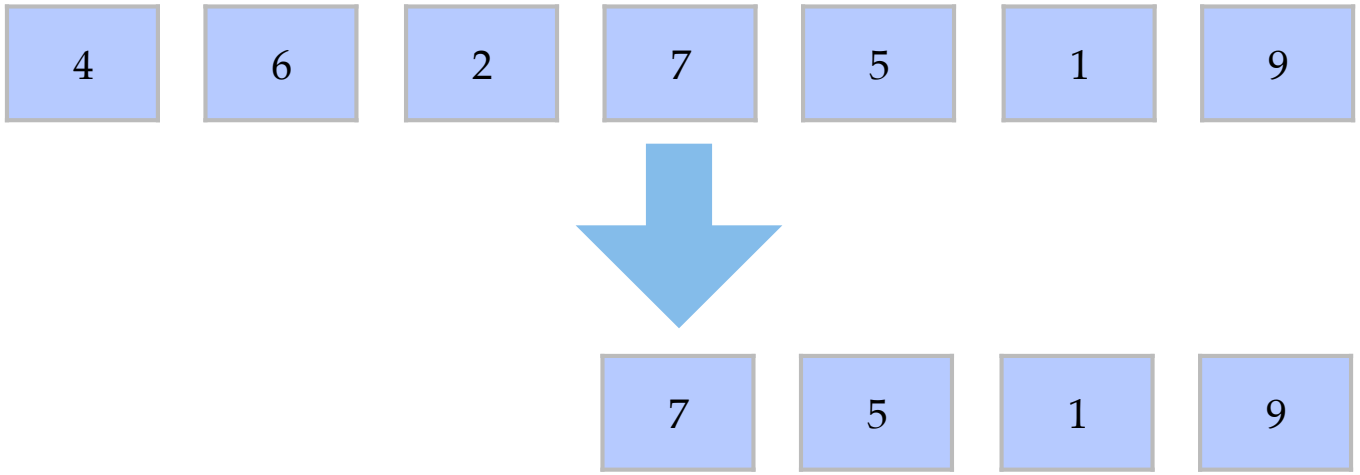


```
c.push_back(5);
c.push_back(1);
c.push_back(9);
```



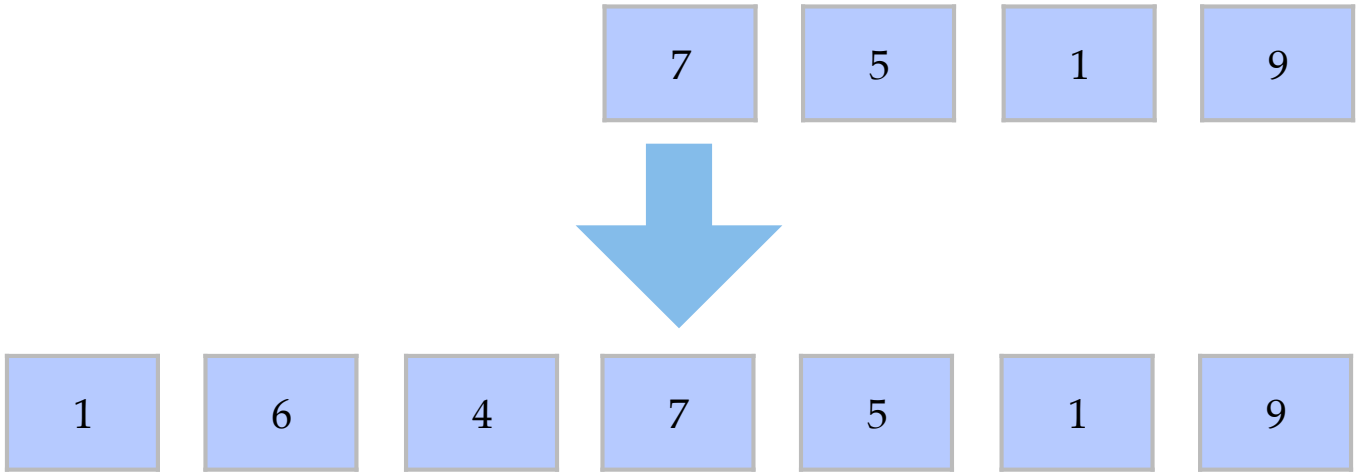
Novos elementos podem ser inseridos ao final de c com a função push_back.

```
c.pop_front();
c.pop_front();
c.pop_front();
```



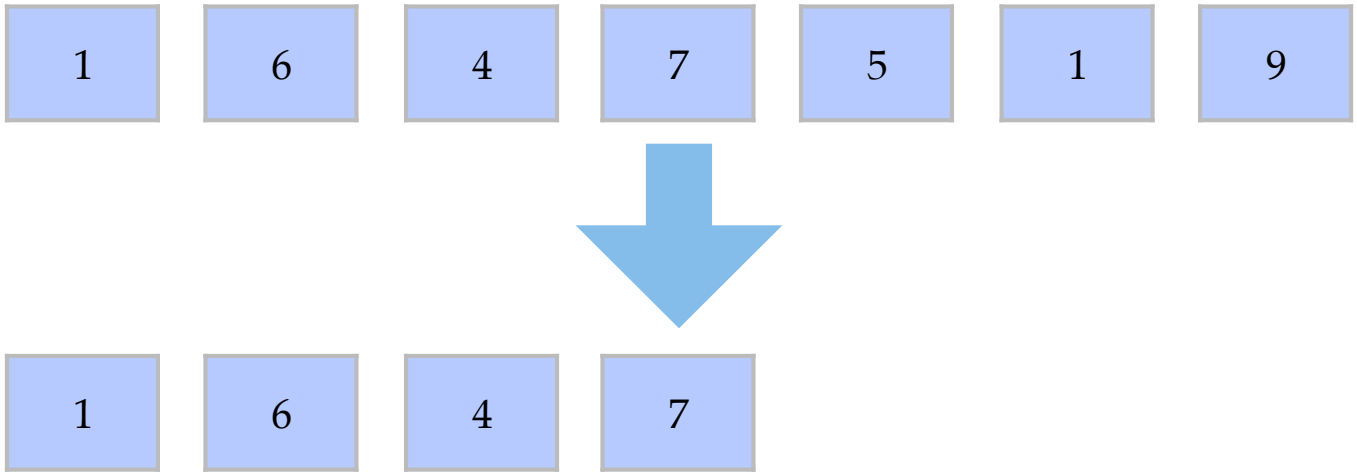
Elementos podem ser removidos do início de `c` com a função `pop_front`.

```
c.push_front(4);
c.push_front(6);
c.push_front(1);
```



Novos elementos podem ser inseridos no início de `c` com a função `push_front`.

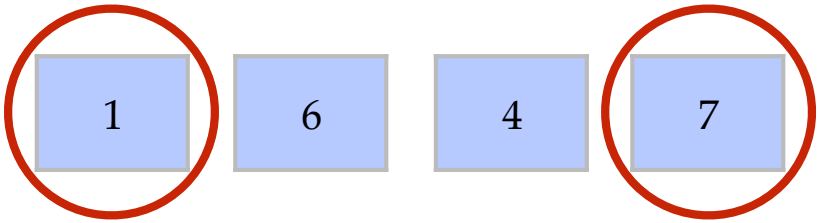
```
c.pop_back();
c.pop_back();
c.pop_back();
```



Elementos podem ser removidos no fim de `c` com a função `pop_back`.

```
cout << c.front() << " " << c.back() << endl;
```

```
1 7
```



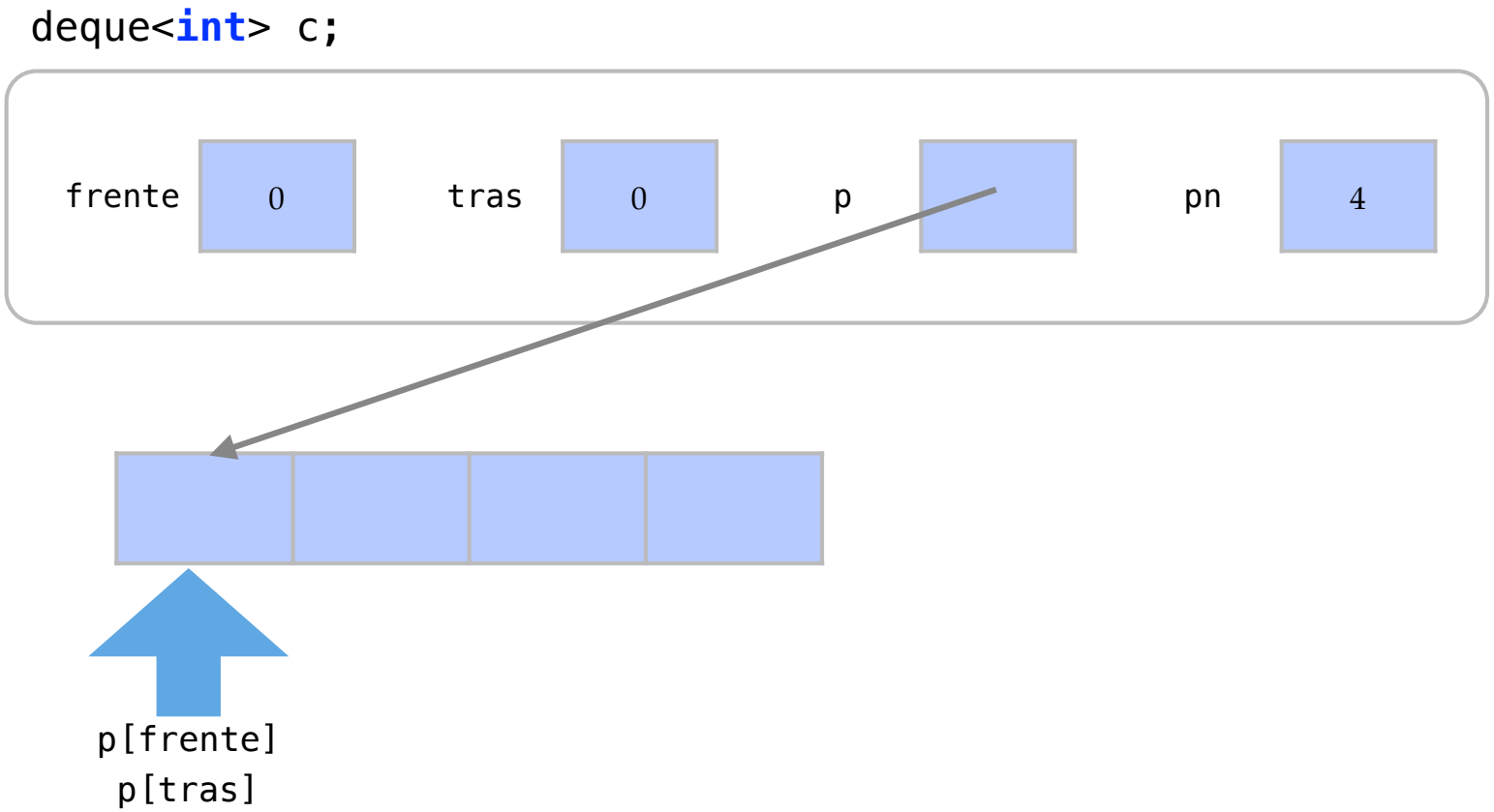
O primeiro e último elementos de `c` podem ser acessados com as funções `front` e `back`.

Deque - Como funciona

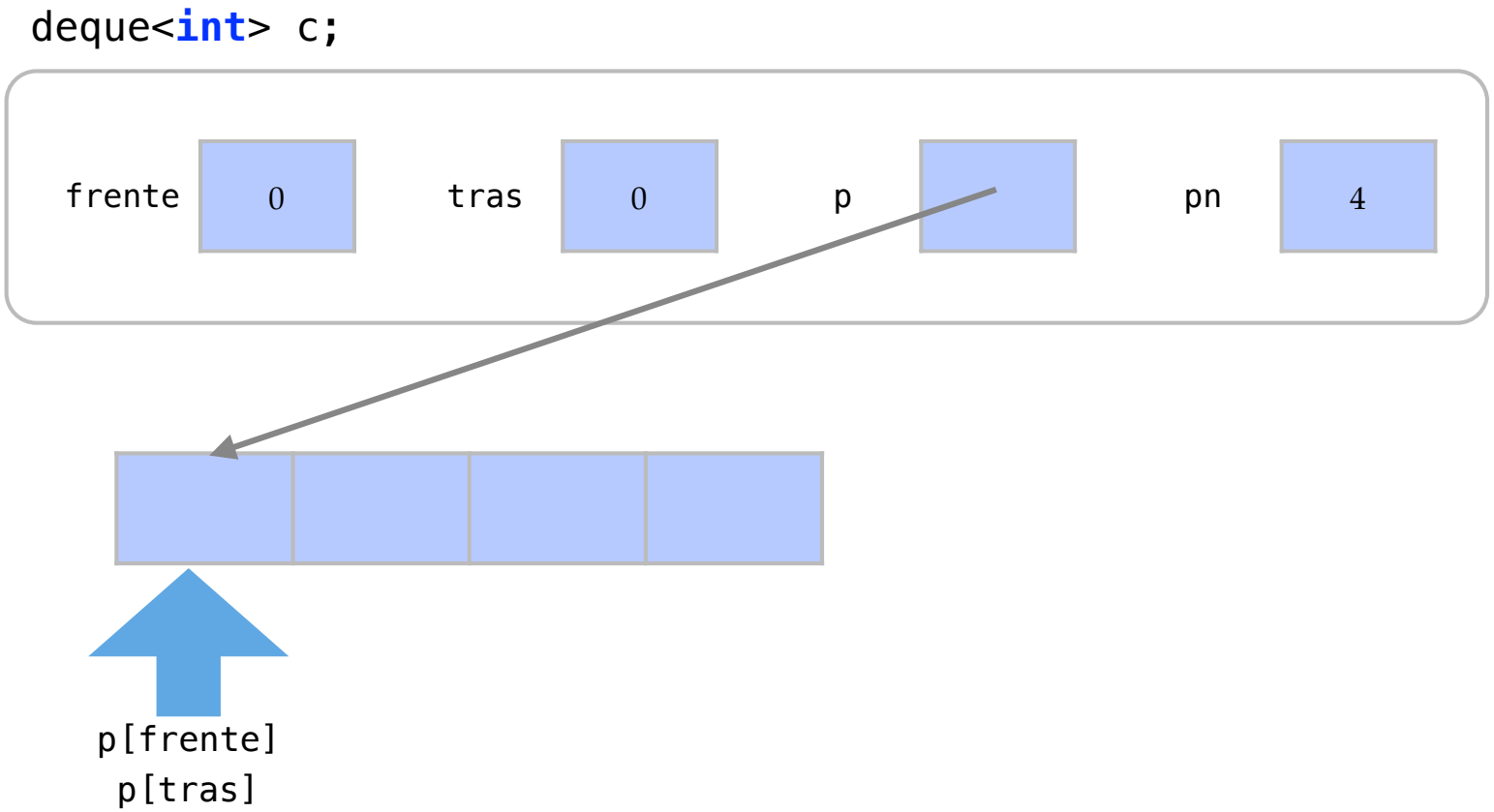
- Os deques utilizam uma estrutura de dados um pouco mais complicada que os vectors
- Para que elementos do início sejam removidos eficientemente, deques utilizam uma estrutura de arranjo diferente para não precisar deslocar os elementos

```
deque<int> c;
```

Com este comando, podemos criar um deque vazio. Internamente, o deque tem um ponteiro e três números inteiros.

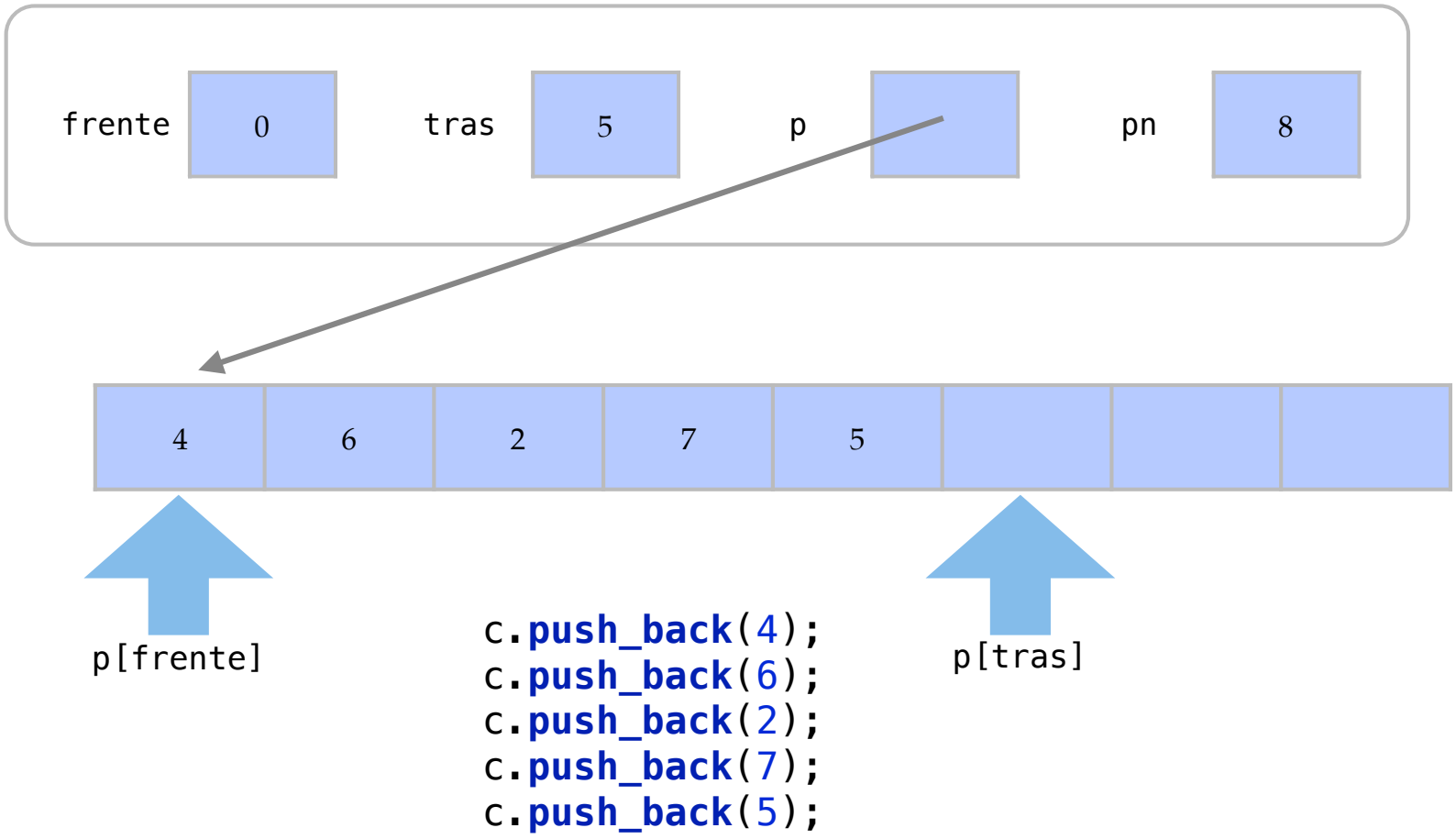


Diferentemente de vector, não temos mais o número n de elementos no container e sim dois elementos `frente` e `tras`.



O elemento `frente` diz onde o primeiro elemento do container se localiza no arranjo. O elemento `tras` diz onde acaba o arranjo.

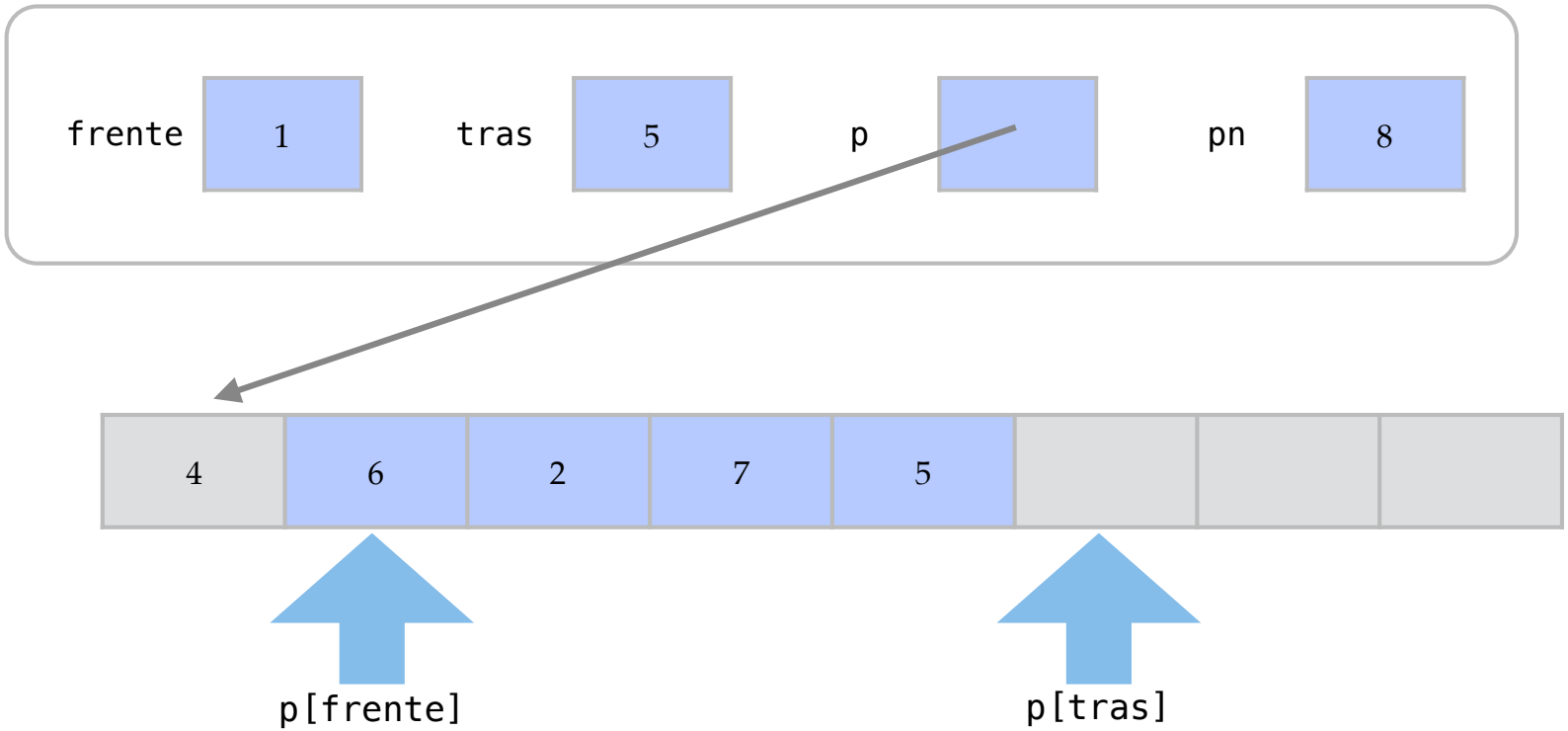
```
deque<int> c;
```




```
c.push_back(4);
c.push_back(6);
c.push_back(2);
c.push_back(7);
c.push_back(5);
```

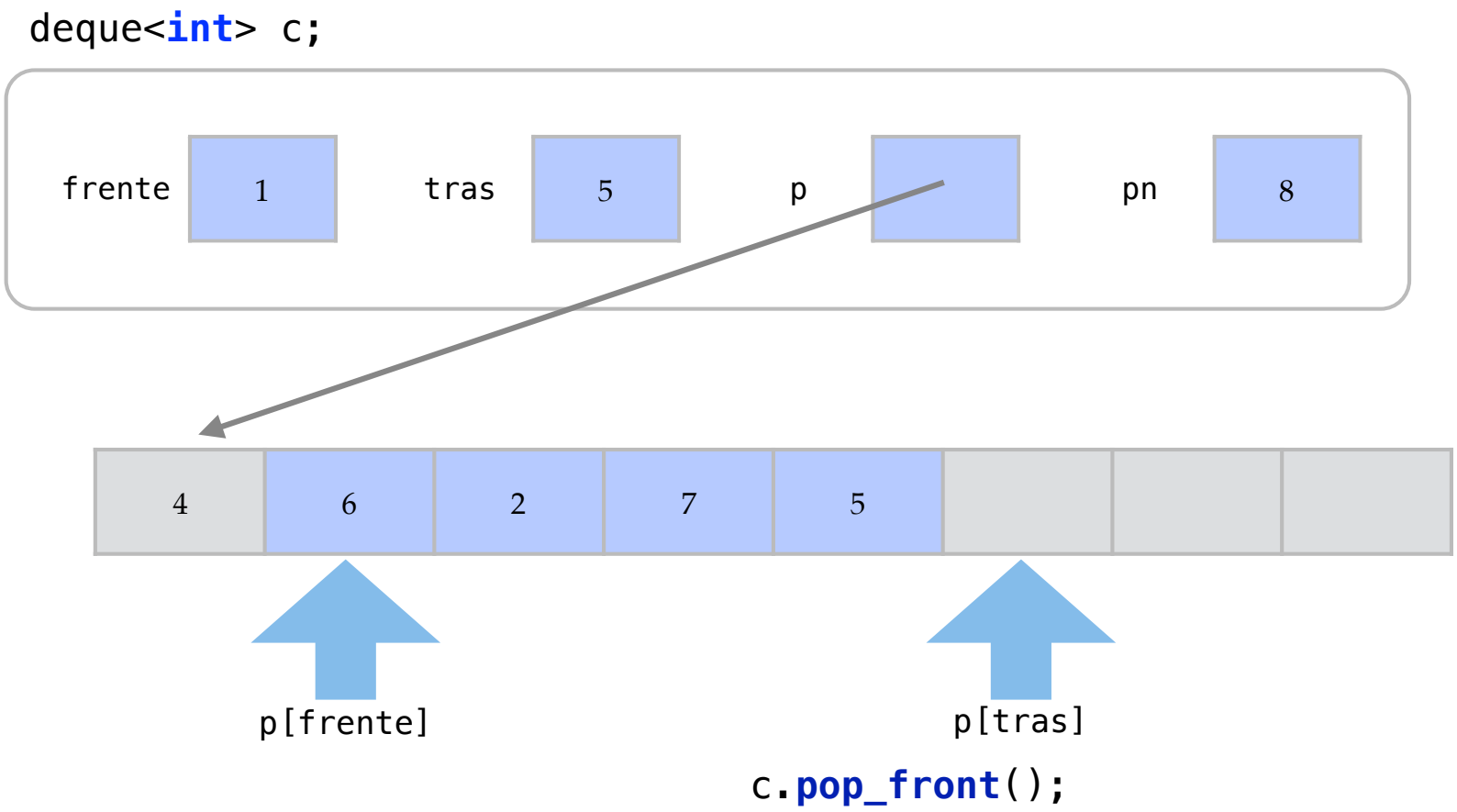
Assim, como no `vector`, a inserção de vários elementos causa o aumento de memória alocada no arranjo.

```
deque<int> c;
```



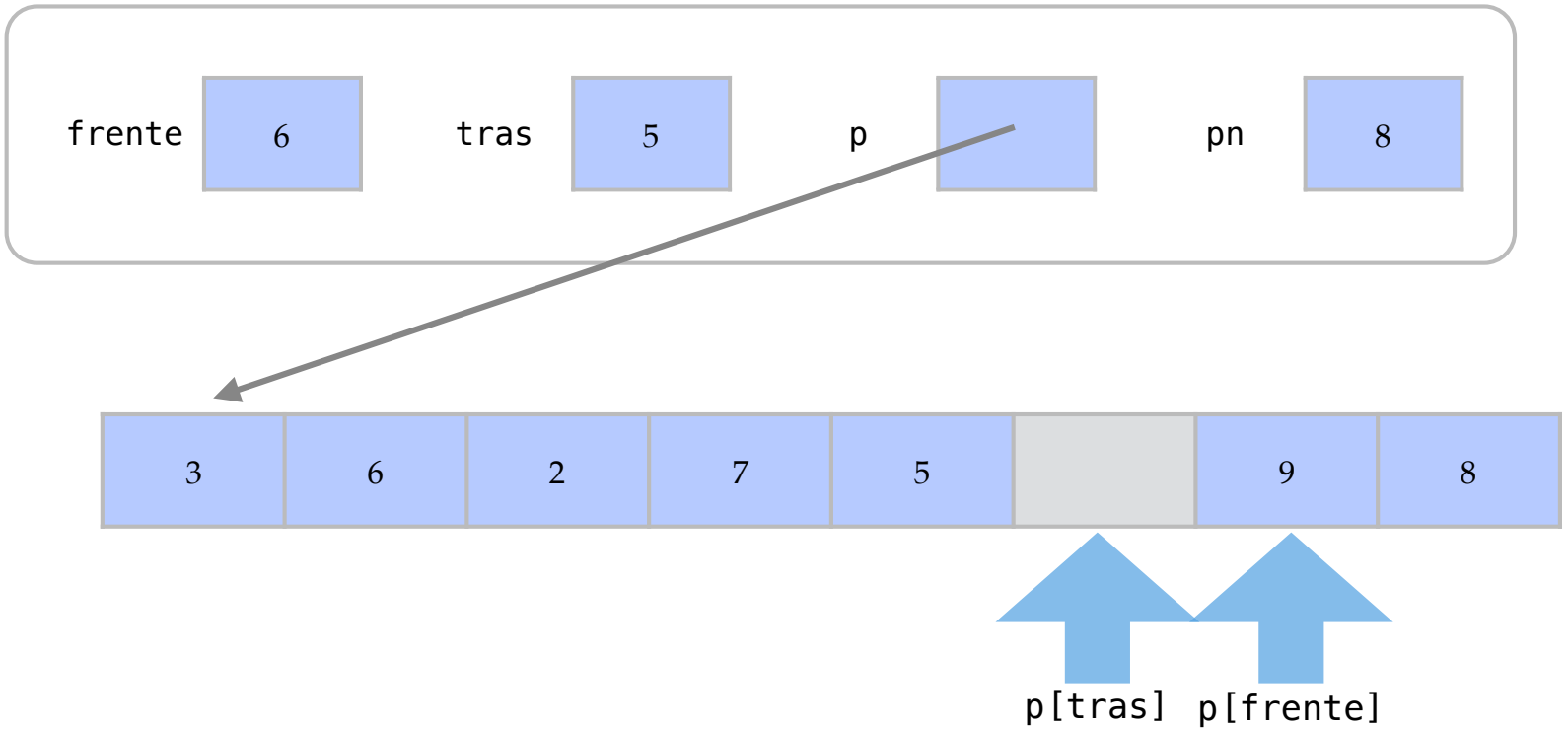

```
c.pop_front();
```

Quando um elemento do início é removido, isto é feito incrementando o valor de `frente`.



Os valores de `frente` e `tras` indicam que devemos considerar apenas valores entre `p[frente]` e `p[tras]` como pertencentes ao container.

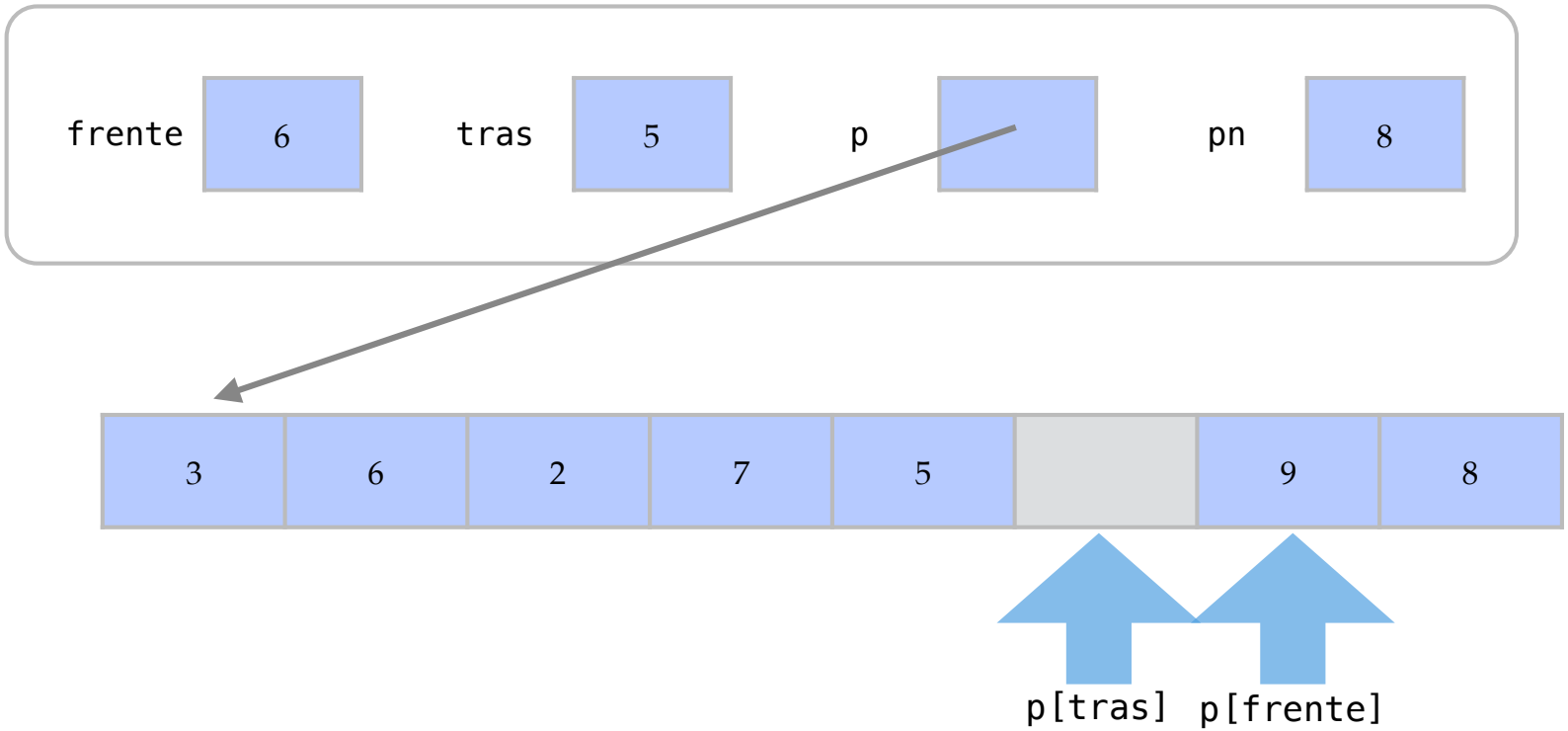
```
deque<int> c;
```




```
c.push_front(3);
c.push_front(8);
c.push_front(9);
```

Quando muitos elementos são inseridos uma estrutura toroidal é utilizada para representar os elementos do container.

```
deque<int> c;
```




```
c.push_front(3);
c.push_front(8);
c.push_front(9);
```

Pode parecer estranho que `frente > tras`, mas isto indica que os elementos do container vão de `p[6]` até `p[7]` e depois continuam entre `p[0]` e `p[4]`, formando uma estrutura circular.

List

- O container `list` representa listas duplamente encadeadas
- Assim como o `deque`, ele consegue eficientemente inserir e remover elementos das primeiras posições com uma sintaxe similar (push_front, push_back, pop_front, pop_back)
- Sua forma de representação, porém, é muito diferente
- Exige a inclusão do cabeçalho `<list>`

List - Como funciona

- As listas utilizam estruturas para representar células que contém ponteiros e valores
- Cada célula aponta para uma célula similar para formar a lista
- Assim, quando um novo elemento é inserido, é alocado mais espaço na memória apenas para este elemento

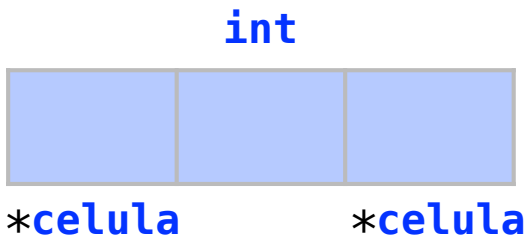
```
list<int> c;
```

Com este comando, podemos criar um `list` vazio. Internamente, o `list` tem dois ponteiros e um número inteiro.

```
list<int> c;
```

Os elementos `frente` e `tras` são ponteiros. Enquanto isso, `n` conta o número de elementos no container.

```
list<int> c;
```




```
c.push_back(4);
```

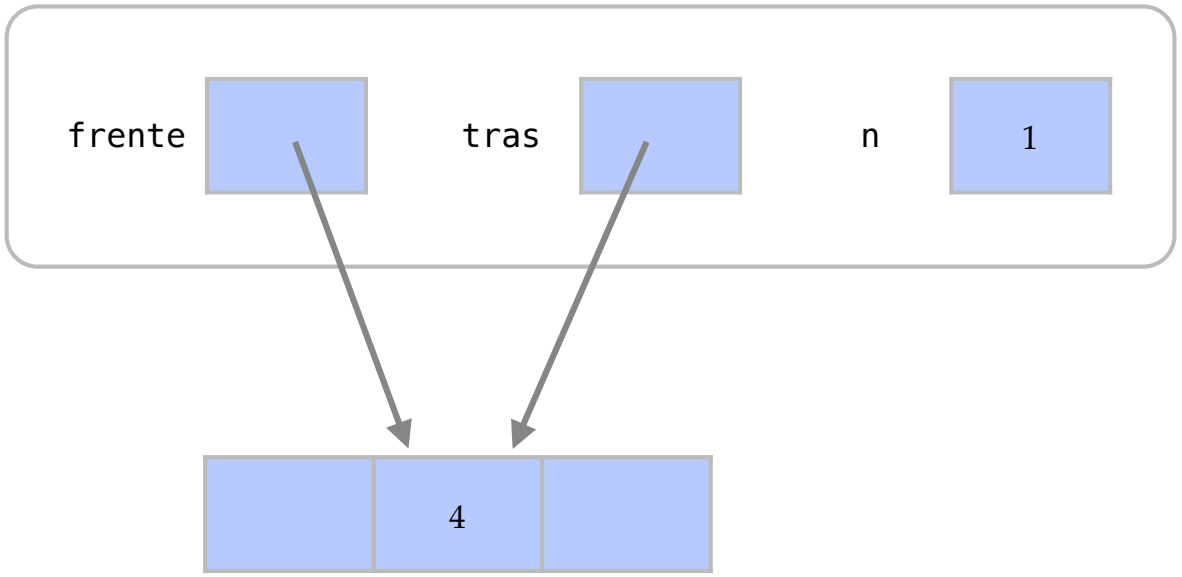
Quando inserimos um elemento no container, uma estrutura chamada célula é criada para este elemento. Cada célula contém 2 ponteiros e um elemento.

```
list<int> c;
```


```
c.push_back(4);
```

O elemento é inserido na célula e os ponteiros não são utilizados por enquanto.

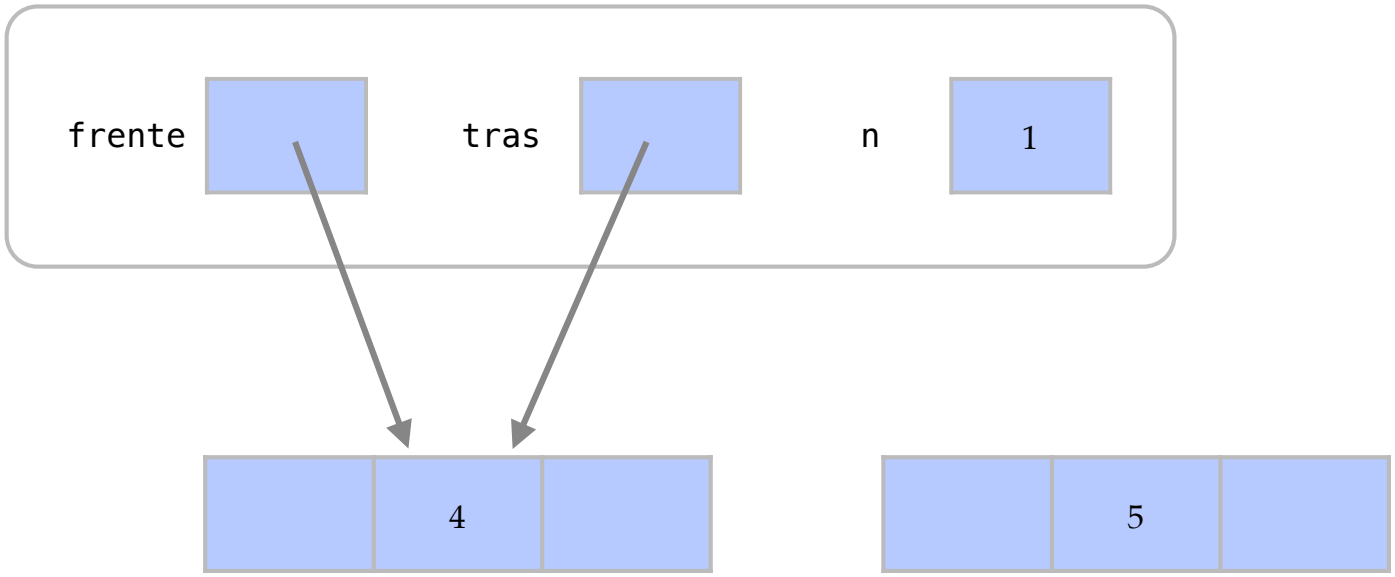
```
list<int> c;
```




```
c.push_back(4);
```

Como esta é a única célula da lista, os ponteiros `frente` e `tras` apontam para ela.

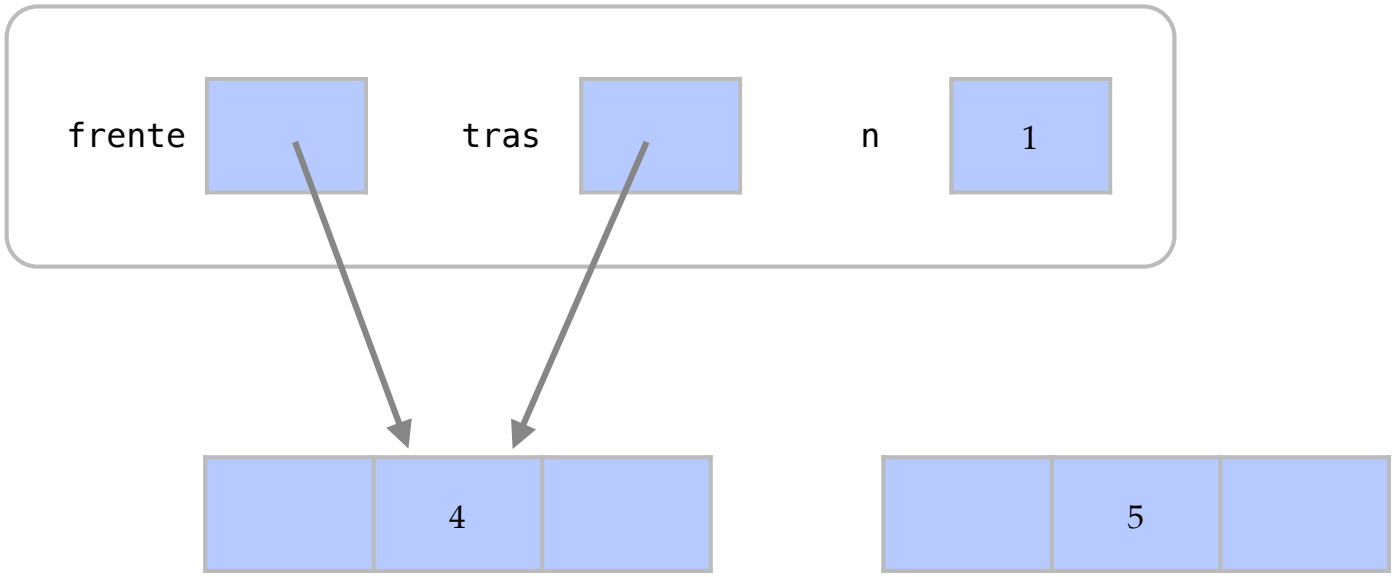
```
list<int> c;
```




```
c.push_back(5);
```

Ao se inserir mais um elemento, é criada para ele mais uma célula.

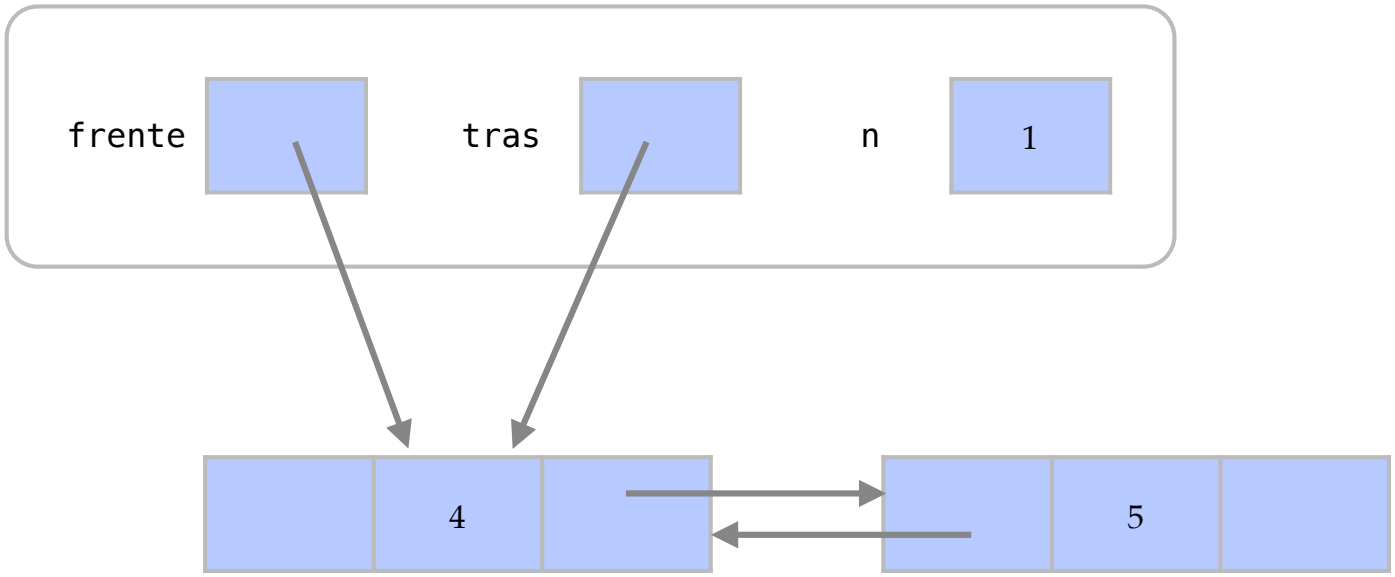
```
list<int> c;
```




```
c.push_back(5);
```

A memória alocada para esta célula está em um local qualquer da memória.

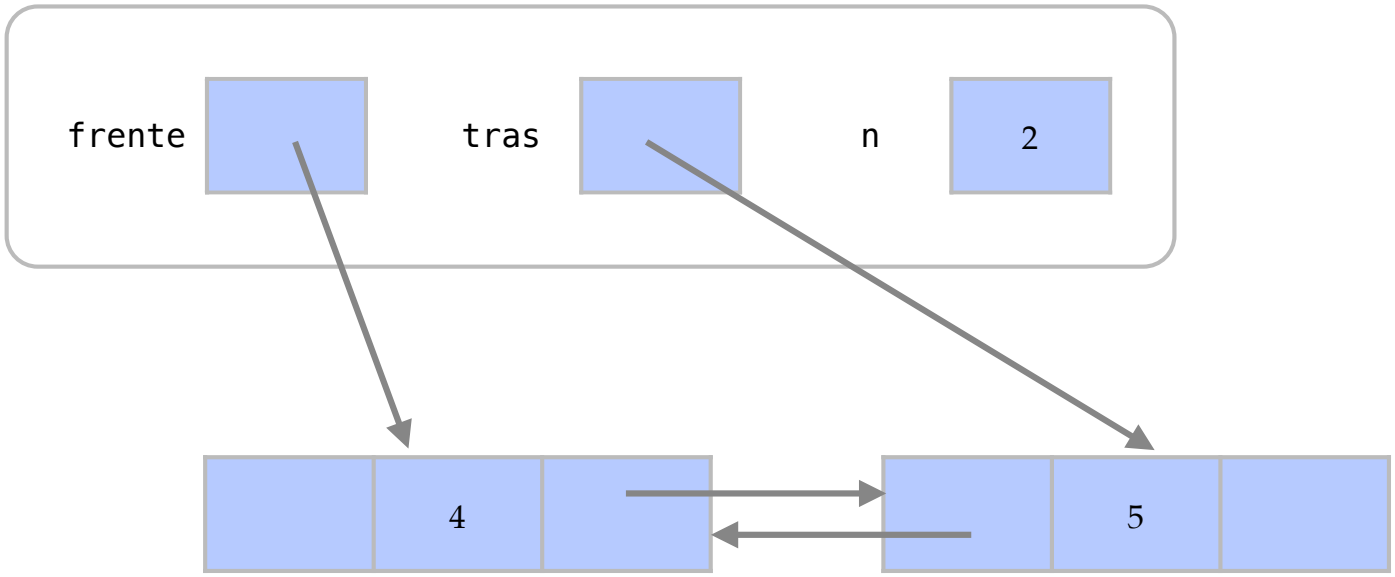
```
list<int> c;
```




```
c.push_back(5);
```

Um ponteiro da célula aponta para o último elemento da lista e o último elemento da lista aponta para a nova célula.

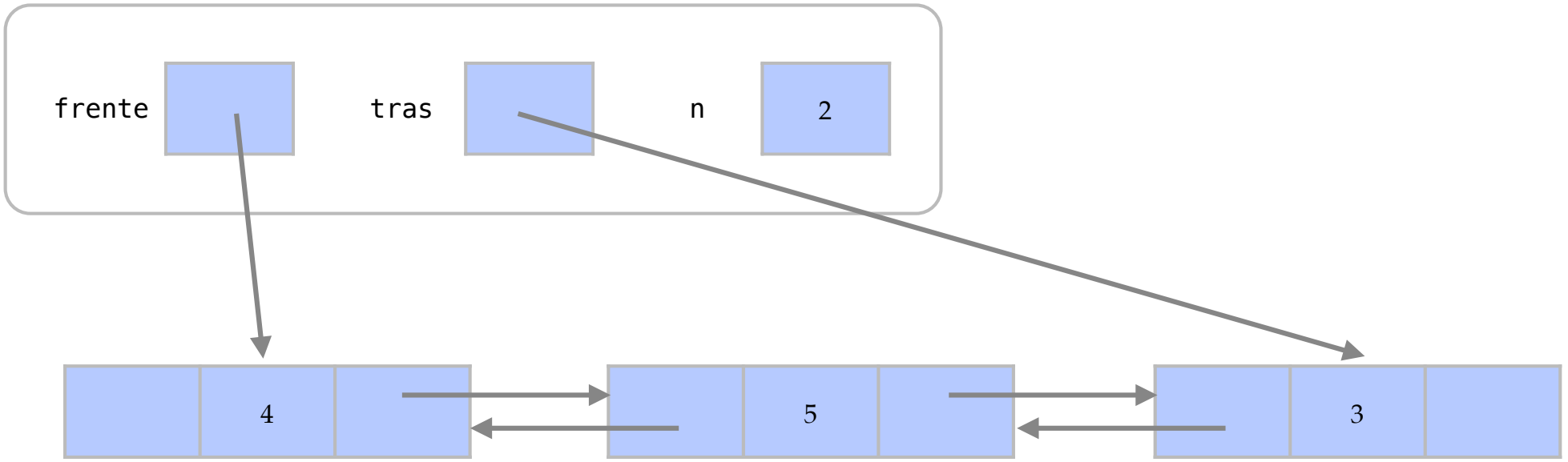
```
list<int> c;
```




```
c.push_back(5);
```

O ponteiro `tras` passa a apontar para a nova célula, `n` é incrementado e o novo elemento está inserido.

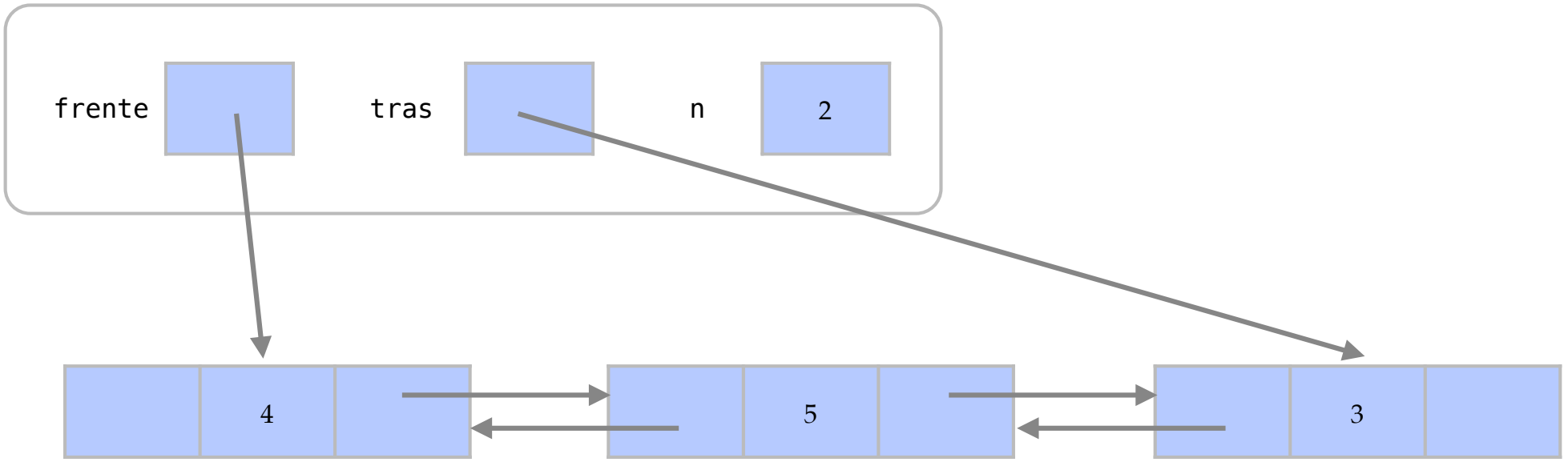
```
list<int> c;
```




```
c.push_back(3);
```

Podemos assim inserir vários elementos com tempo *O(1)*.

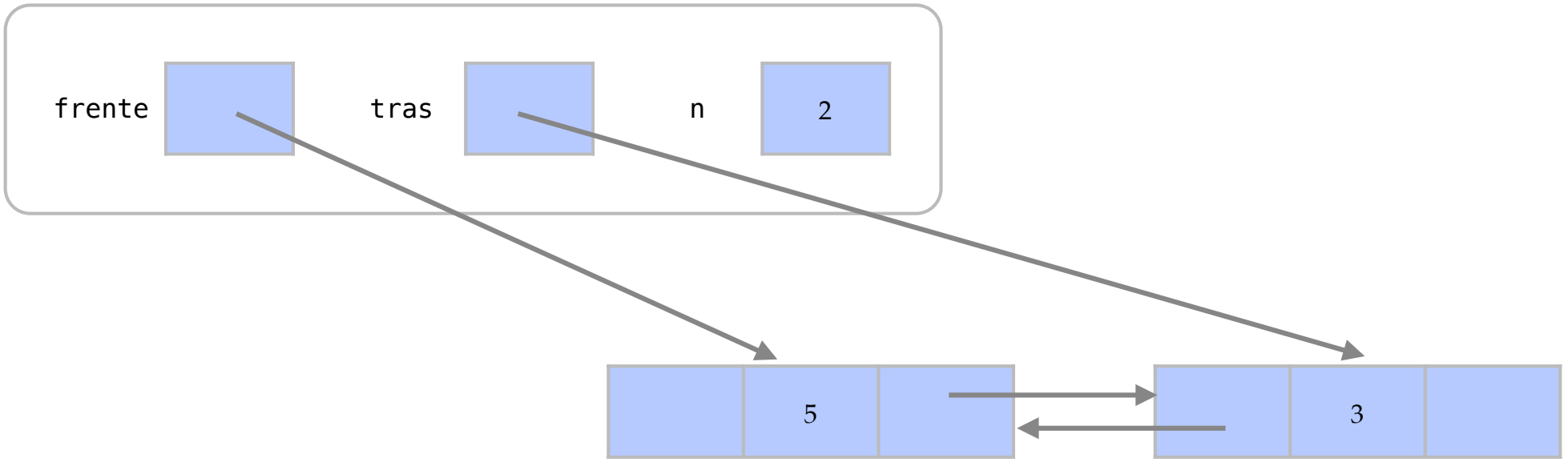
```
list<int> c;
```




```
c.pop_front();
```

Para remover um elemento do início ou do final, um processo muito similar é feito.

```
list<int> c;
```




```
c.pop_front();
```

`frente` aponta para o próximo elemento e o anterior é removido, com suas conexões.

Utilizando subscritos

- Assim como utilizamos os subscritos `[ ]` para acessar posições de arranjos, podemos utilizar subscritos para acessar elementos dos containers de sequência
- Apenas os containers de sequência chamados de containers **de acesso aleatório** tem este recurso
- O subscrito `[i]` é utilizado para acessar o (i+1)-ésimo elemento do container.

Vector - Subscritos

```cpp
#include <iostream>
#include <vector>

using namespace std;

int main(){
    vector<int> c;
    for (int i=1; i<6; i++) {
        c.push_back(i);
    }
    for (int i=0; i<c.size(); i++) {
        cout << c[i] << " ";
    }
    cout << endl;
}
```

```cpp
#include <iostream>
#include <vector>

using namespace std;

int main(){
    vector<int> c;
    for (int i=1; i<6; i++) {
        c.push_back(i);
    }
    for (int i=0; i<c.size(); i++) {
        cout << c[i] << " ";
    }
    cout << endl;
}
```

Este código utiliza subscritos para imprimir os valores do container `c`.

```cpp
#include <iostream>
#include <vector>

using namespace std;

int main(){
    vector<int> c;
    for (int i=1; i<6; i++) {
        c.push_back(i);
    }
    for (int i=0; i<c.size(); i++) {
        cout << c[i] << " ";
    }
    cout << endl;
}
```

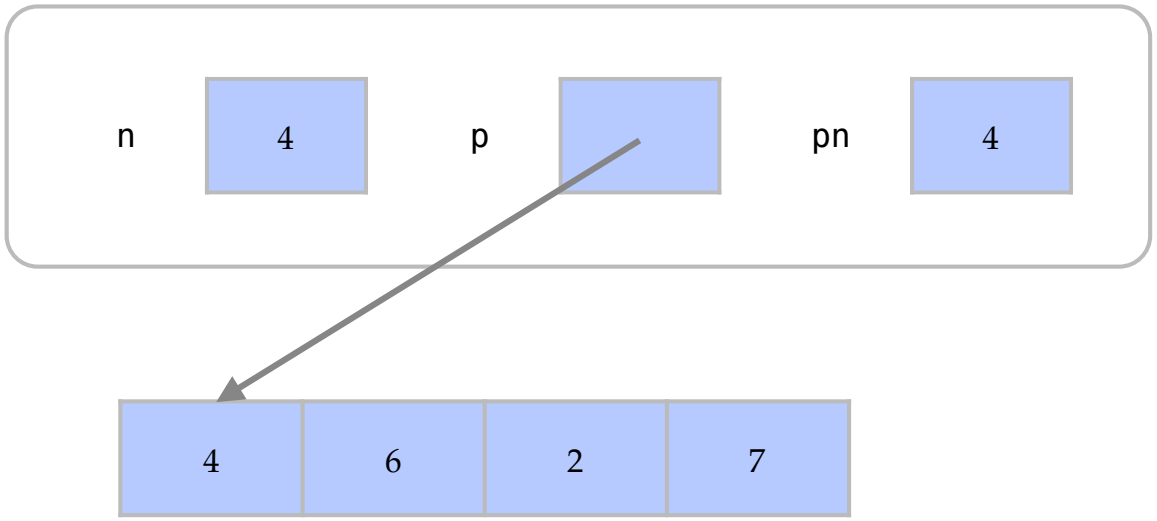
Este trecho de código cria um `vector` chamado `c` e insere em seu final os elementos i = 1, 2, 3, 4 e 5

```cpp
#include <iostream>
#include <vector>

using namespace std;

int main(){
    vector<int> c;
    for (int i=1; i<6; i++) {
        c.push_back(i);
    }
    for (int i=0; i<c.size(); i++) {
        cout << c[i] << " ";
    }
    cout << endl;
}
```

Neste trecho de código, o subscrito `c[i]` é utilizado para imprimir os elementos de `c`.

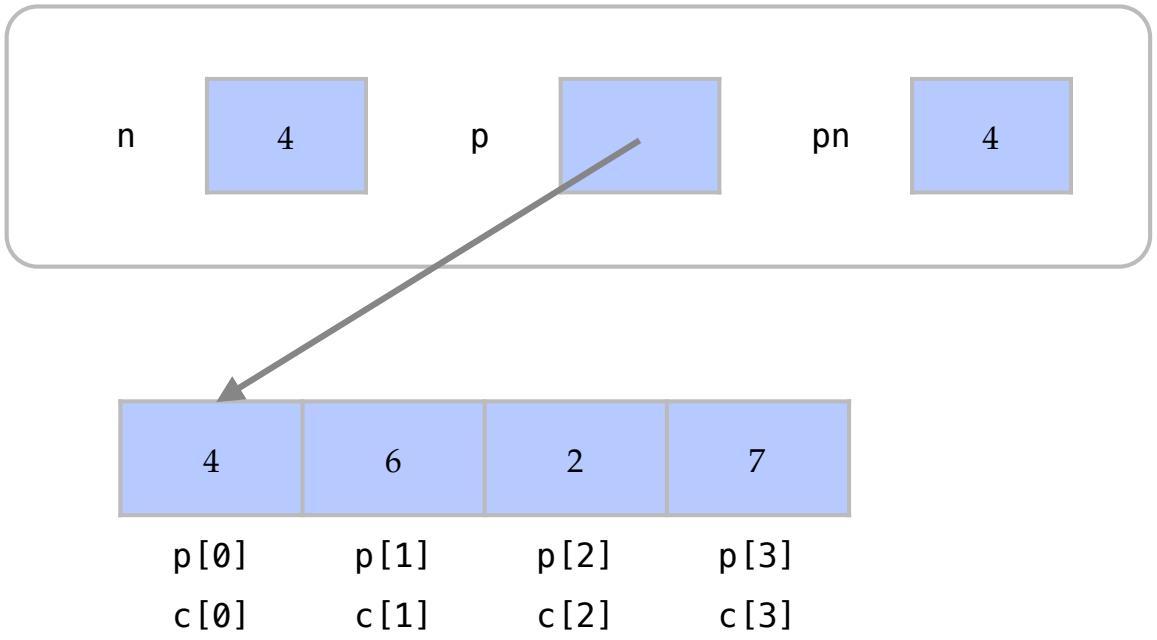
```cpp
#include <iostream>
#include <vector>

using namespace std;

int main(){
    vector<int> c;
    for (int i=1; i<6; i++) {
        c.push_back(i);
    }
    for (int i=0; i<c.size(); i++) {
        cout << c[i] << " ";
    }
    cout << endl;
}
```

```
vector<int> c;
```



Internamente, o i-ésimo elemento do vector pode ser acessado ao se acessar o i-ésimo elemento do arranjo p que guarda seus elementos.

```
vector<int> c;
```



Assim, as posições de subscrito do arranjo apontado por p são as mesmas que devem ser retornadas pelos subscritos do container c.

Deque - Subscritos

```cpp
#include <iostream>
#include <deque>

using namespace std;

int main()
{
    deque<float> col;
    for (int i=1; i<6; i++) {
        col.push_front(i*1.1);
    }
    for (int i=0; i<col.size(); i++) {
        cout << col[i] << " ";
    }
    cout << endl;
}
```

```cpp
#include <iostream>
#include <deque>

using namespace std;

int main()
{
    deque<float> c;
    for (int i=1; i<6; i++) {
        c.push_front(i*1.1);
    }
    for (int i=0; i<c.size(); i++) {
        cout << c[i] << " ";
    }
    cout << endl;
}
```

Este código utiliza subscritos também para imprimir os valores no deque `c`.

```cpp
#include <iostream>
#include <deque>

using namespace std;

int main()
{
    deque<float> c;
    for (int i=1; i<6; i++) {
        c.push_front(i*1.1);
    }
    for (int i=0; i<c.size(); i++) {
        cout << c[i] << " ";
    }
    cout << endl;
}
```

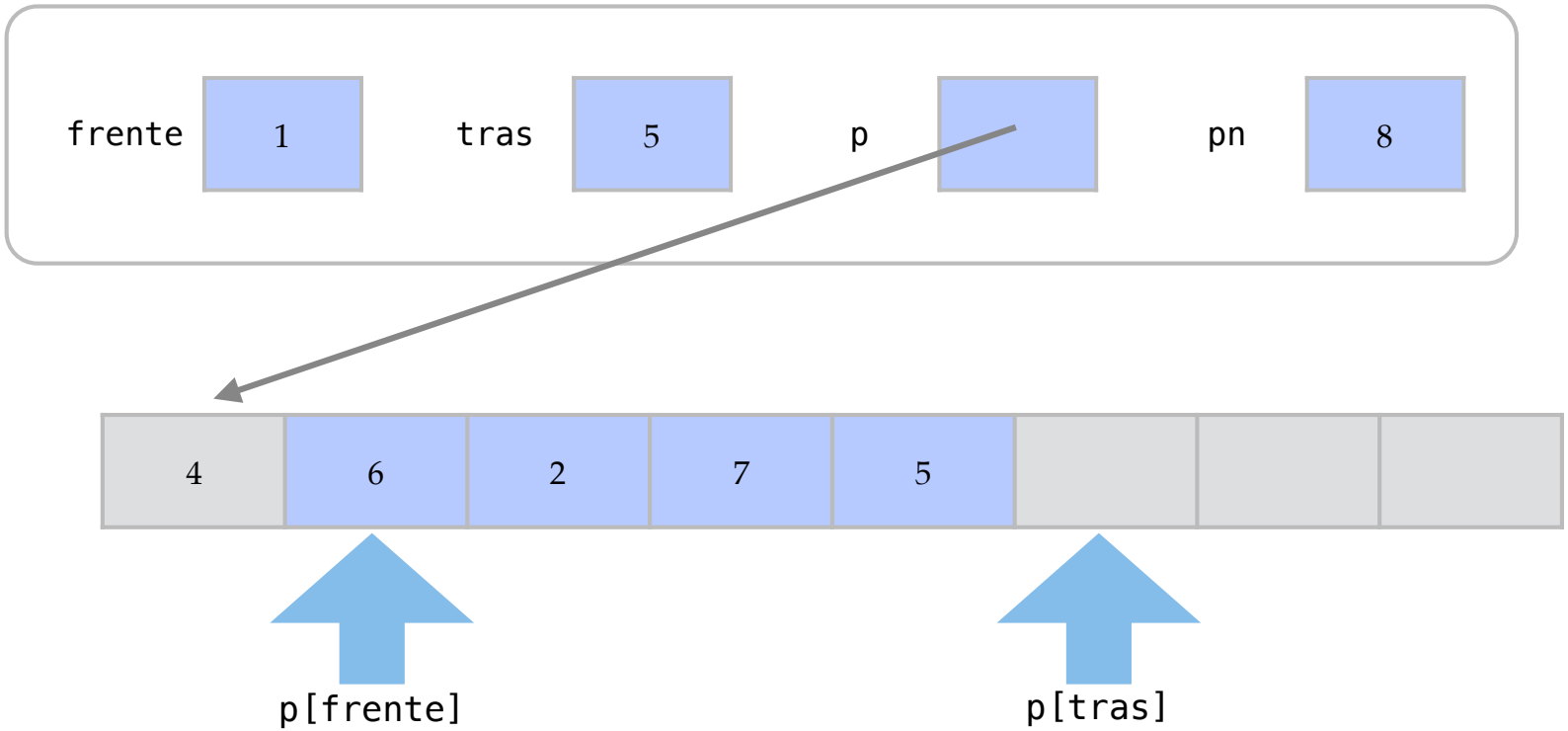
Este bloco de código cria um deque chamado c e insere em seu início os elementos i = 1.1, 2.2, 3.3, 4.4 e 5.5

```cpp
#include <iostream>
#include <deque>

using namespace std;

int main()
{
    deque<float> c;
    for (int i=1; i<6; i++) {
        c.push_front(i*1.1);
    }
    for (int i=0; i<c.size(); i++) {
        cout << c[i] << " ";
    }
    cout << endl;
}
```

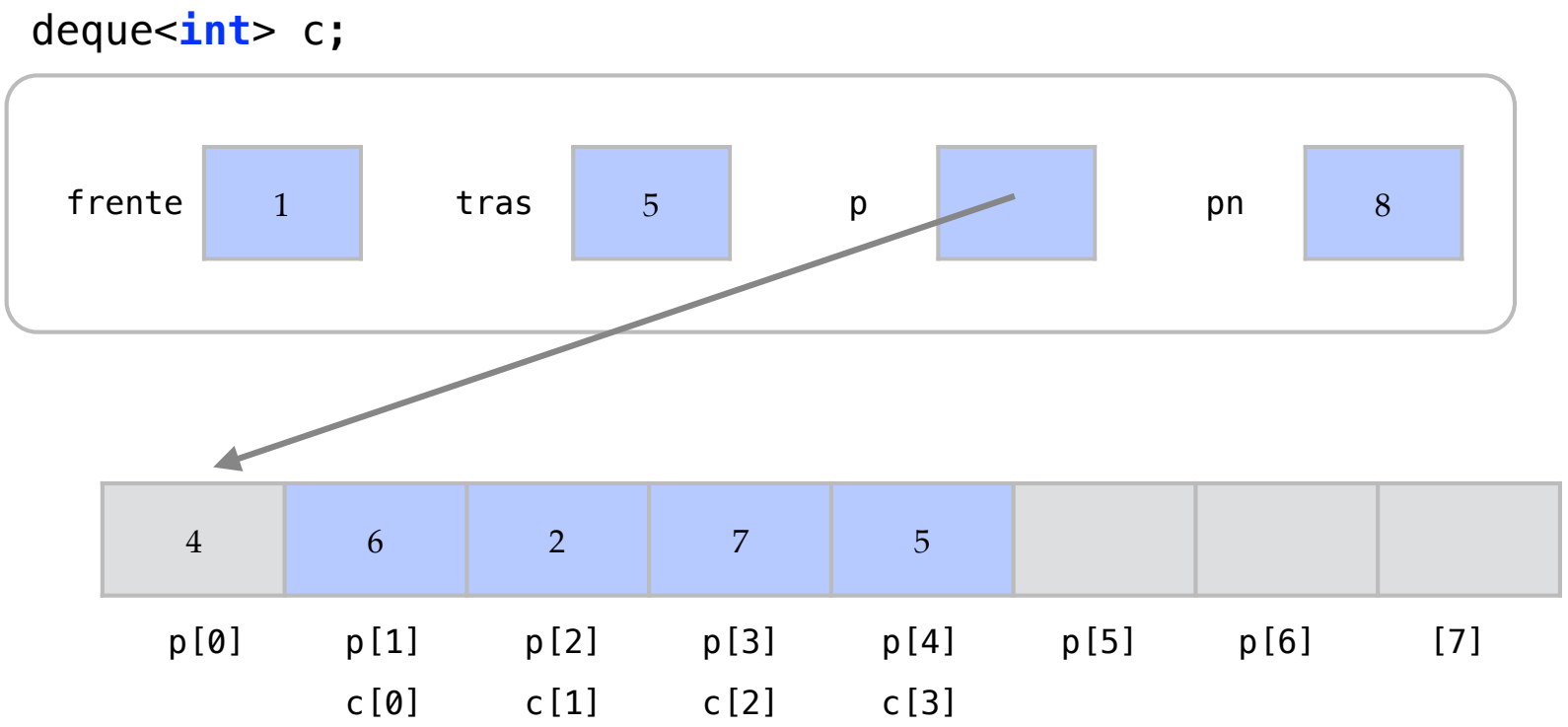
Neste trecho de código, o subscrito `c[i]` é utilizado para imprimir os elementos de `c`.

```cpp
#include <iostream>
#include <deque>

using namespace std;

int main()
{
    deque<float> c;
    for (int i=1; i<6; i++) {
        c.push_front(i*1.1);
    }
    for (int i=0; i<c.size(); i++) {
        cout << c[i] << " ";
    }
    cout << endl;
}
```

```
deque<int> c;
```

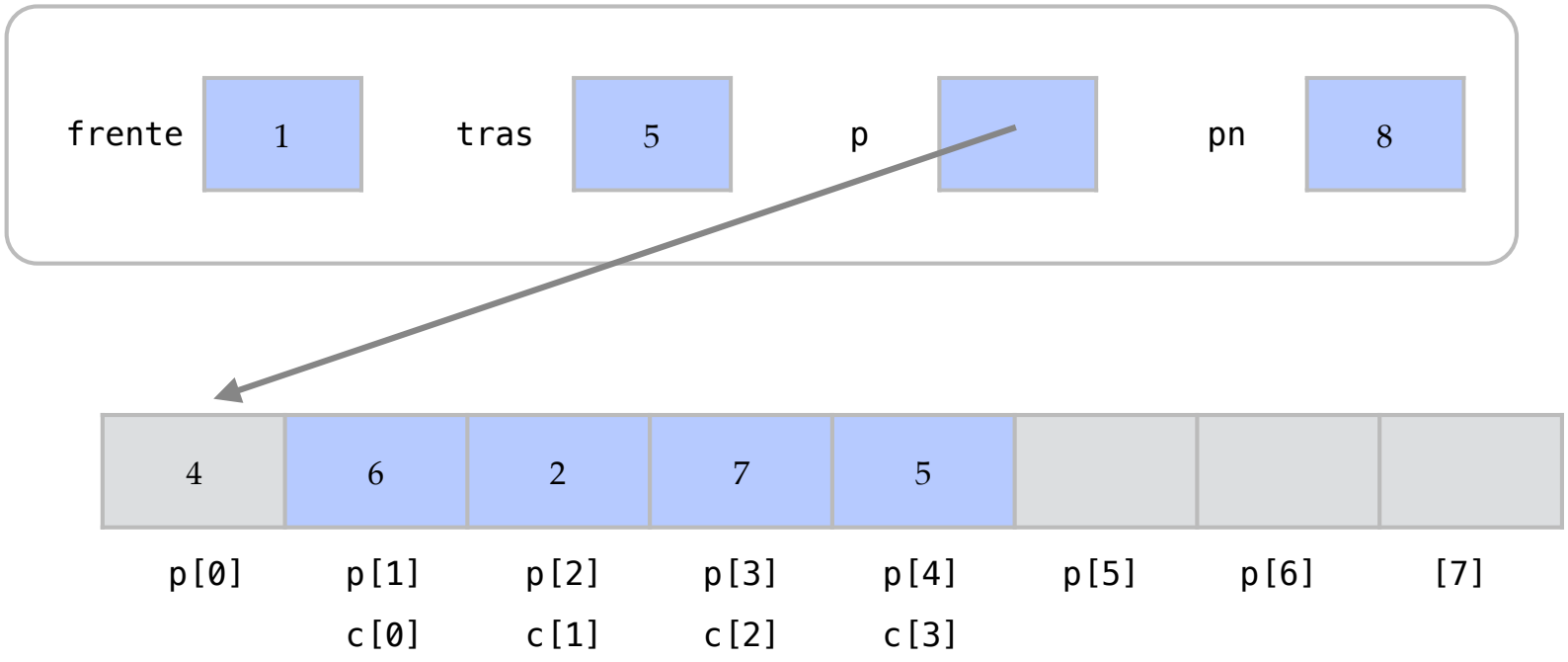


Internamente o i-ésimo elemento do `deque` pode ser acessado ao se acessar o i-ésimo elemento do arranjo `p` após o elemento `p[frente]`.

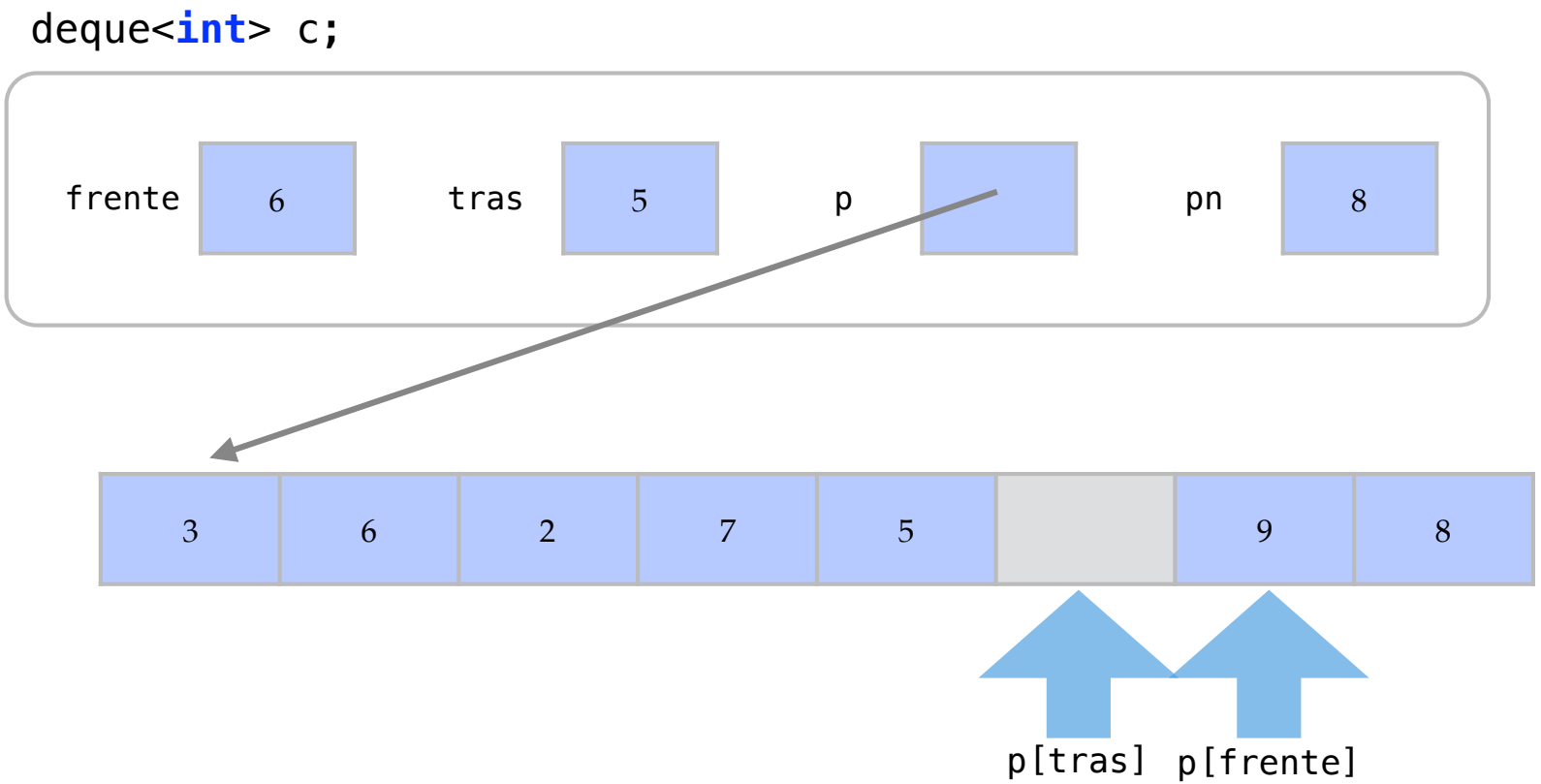


Assim, a localização das posições `frente` e `tras` devem ser consideradas ao se procurar o i-ésimo elemento do deque `c`.

```
deque<int> c;
```

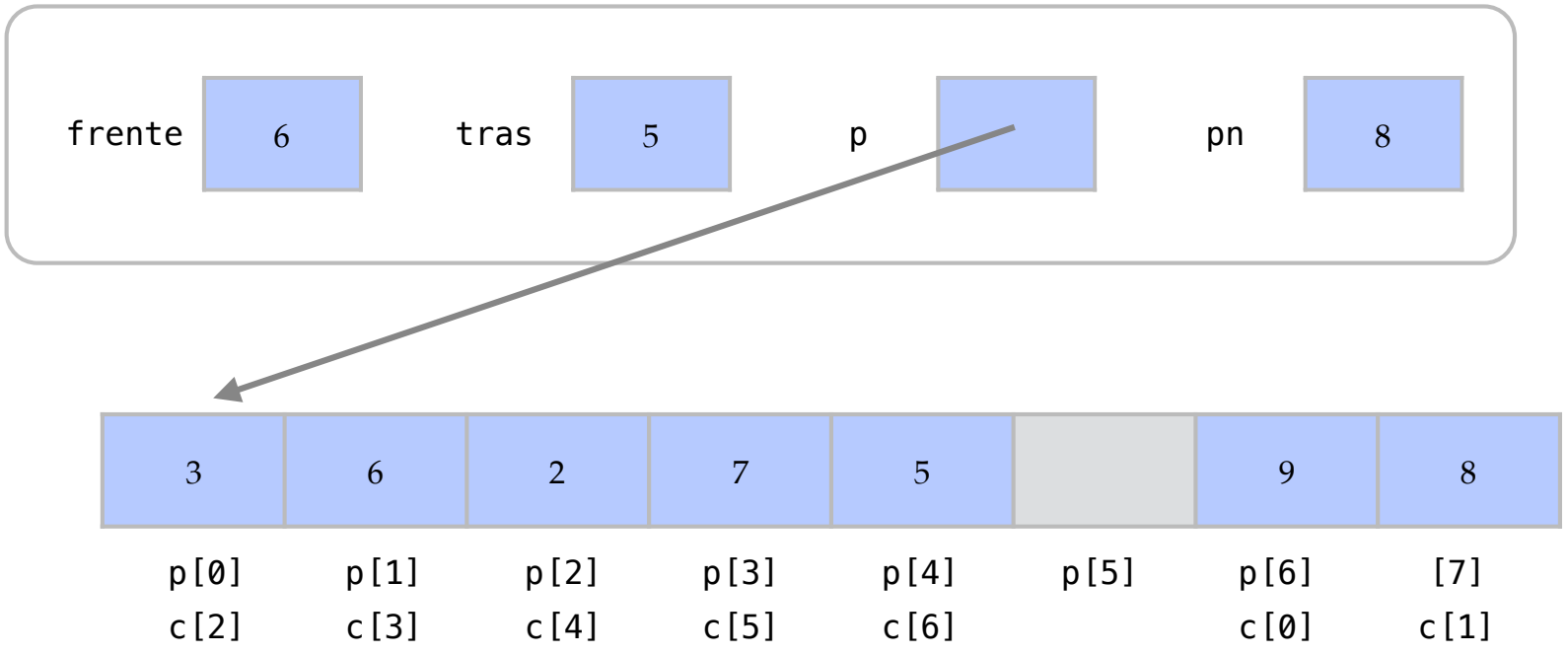


Neste caso, a posição do i-ésimo elemento do container pode ser encontrado na posição `p[frente+i]` do arranjo.



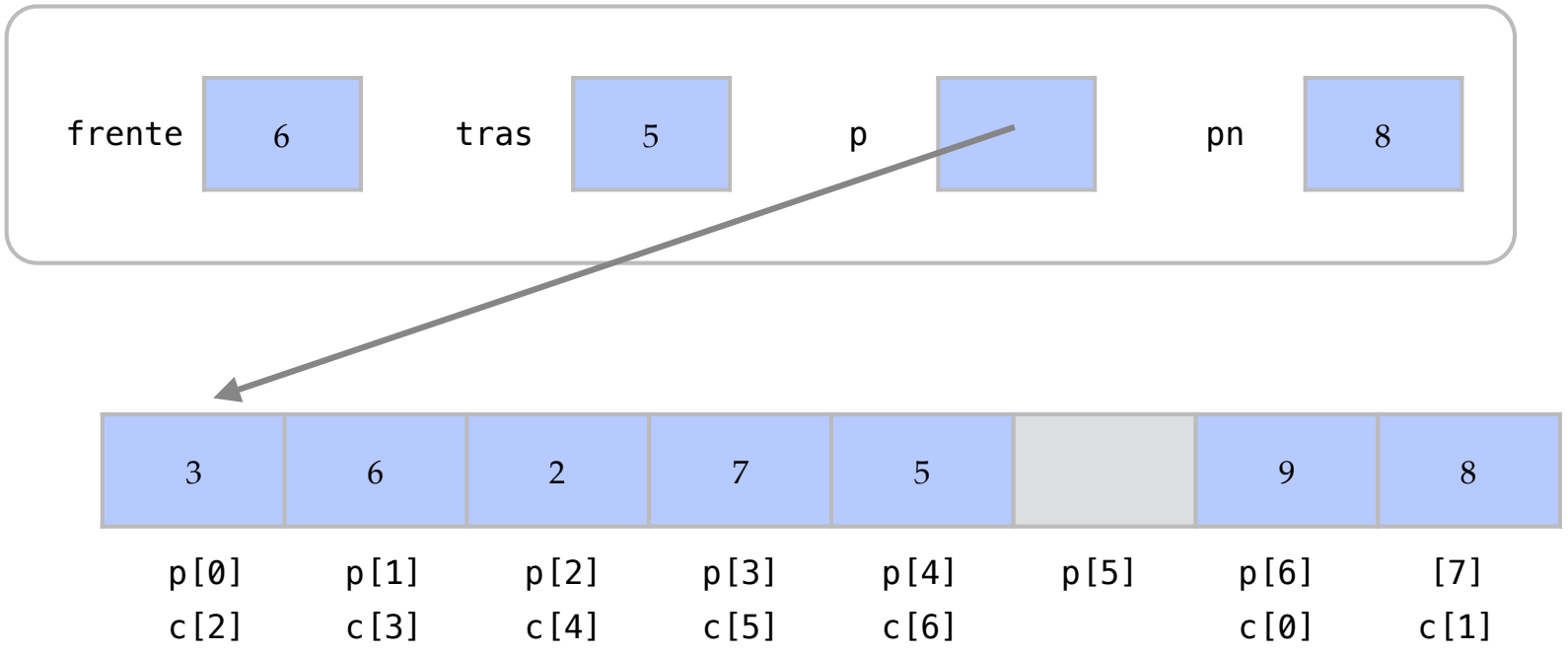
Quando ciclos ocorrem, uma operação de módulo no valor encontrado é necessária para achar a posição correta.

```
deque<int> c;
```



Por exemplo, veja os subscritos do arranjo e os subscritos do container neste caso.

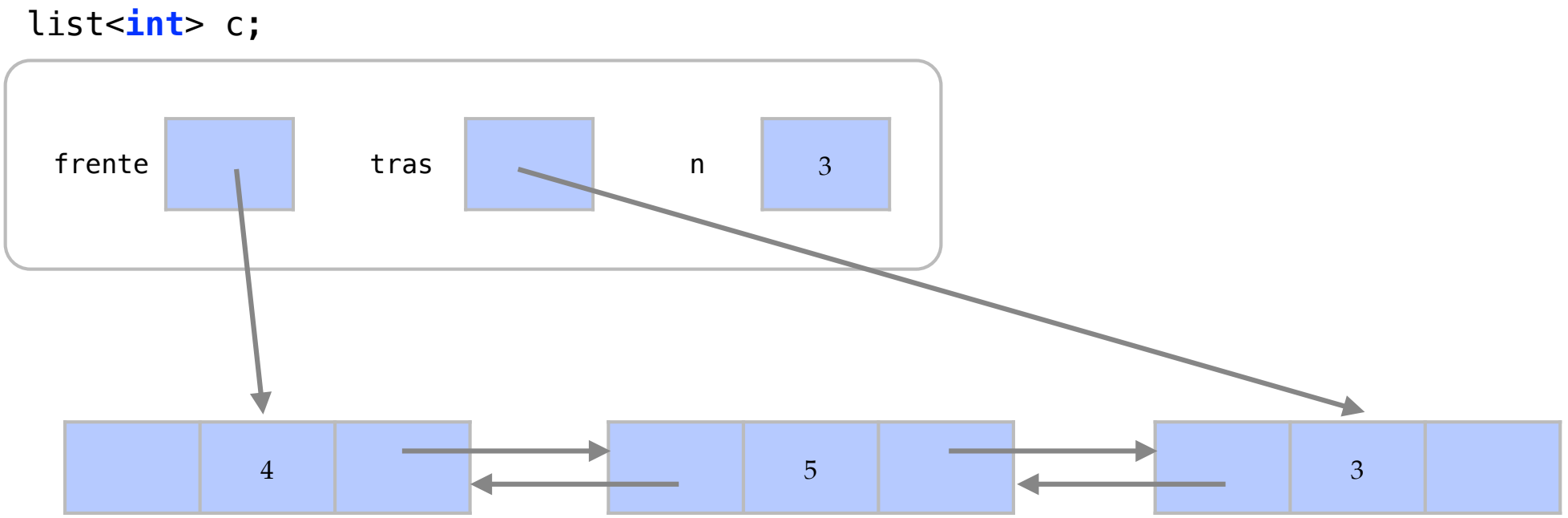
```
deque<int> c;
```



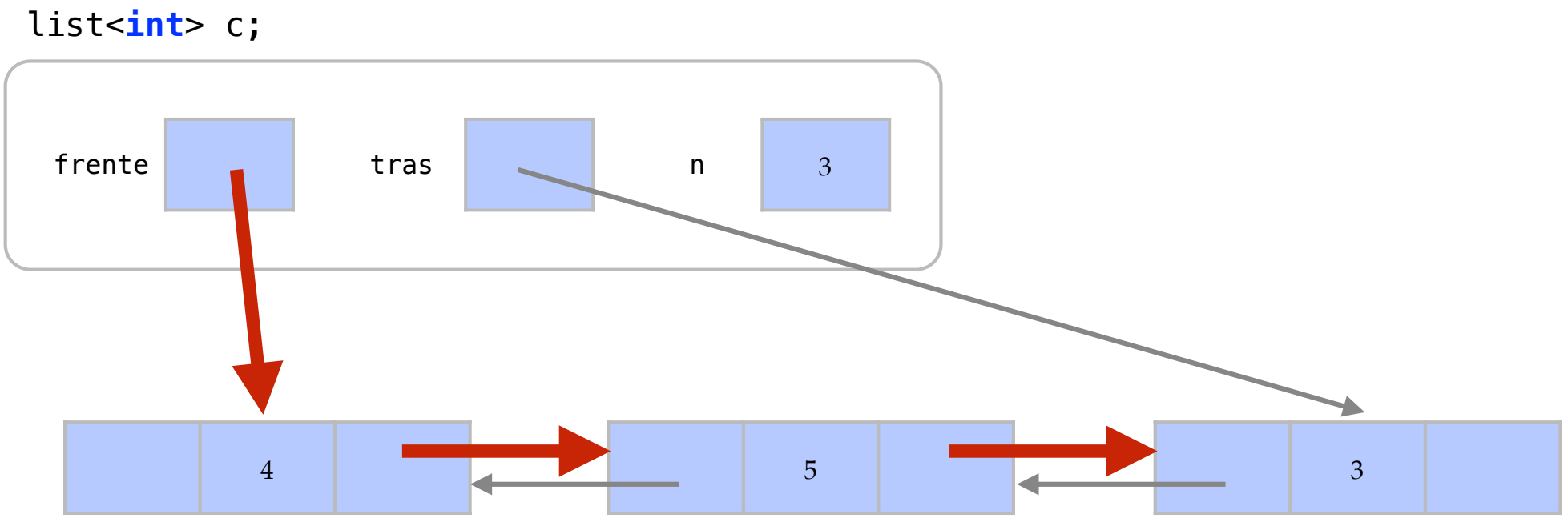
A posição do i-ésimo elemento do container pode ser encontrado na posição `p[(frente+i)%pn]` do arranjo.

List - Subscritos

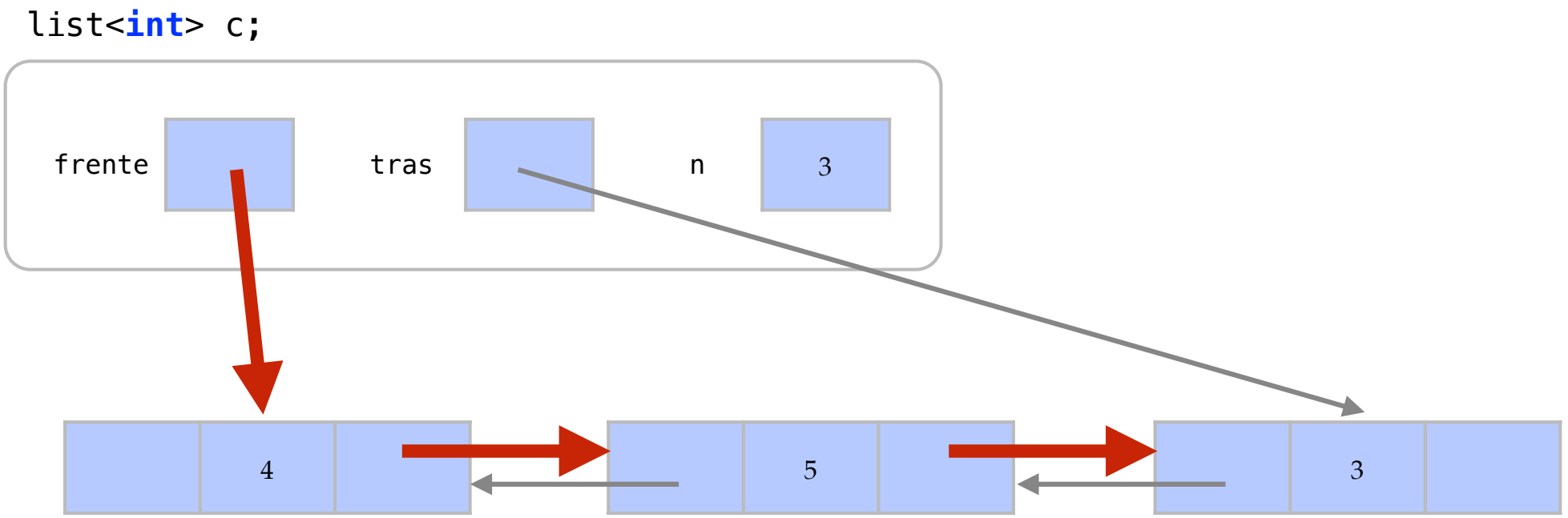
- Já para o container `list`, não é possível encontrar diretamente o i-ésimo elemento da sequência já que os elementos não são organizados em arranjos nesta estrutura
- Assim, apenas as estruturas `vector` e `deque` são de acesso aleatório por serem baseadas em arranjos.



Como os elementos do `list c` estão espalhados na memória, não existe uma operação que calcule diretamente a posição do i-ésimo elemento do container na memória.



A única maneira de se encontrar o i-ésimo elemento de c é analisando no 1° elemento onde está o 2°, no 2° onde está o 3°, no 3° onde está o 4°, ..., no (i-1)-ésimo onde está o i-ésimo.



Ou seja, precisamos seguir os ponteiros i vezes. Assim, achar o i-ésimo elemento a partir do primeiro tem um custo $O(i)$, ou seja, $O(n)$ no pior caso e caso médio.

Subscritos

- Como vimos, os subscritos são utilizados para acessar diretamente o i-ésimo elemento do container
- Este é um recurso que encontra o i-ésimo elemento do container no arranjo alocado para guardar os elementos
- Isto é possível apenas para os containers `vector` e `deque` que usam arranjos, que alocam os elementos em sequência na memória

Containers X Arranjos

- A utilização de subscritos é muito útil pois permite até que utilizemos um container de sequência para substituir arranjos
- Com isto ganhamos várias vantagens em relação a arranjos:
  - Containers já controlam seus próprios tamanhos
  - Containers contém em si mesmos algoritmos eficientes para várias tarefas

Iteradores

- Como vimos, a opção de se acessar elementos de um container através do operador de subscrito é restrita apenas a alguns tipos de container
- Para conseguirmos acessar elementos de todos os tipos de container, precisamos de iteradores.

Iteradores

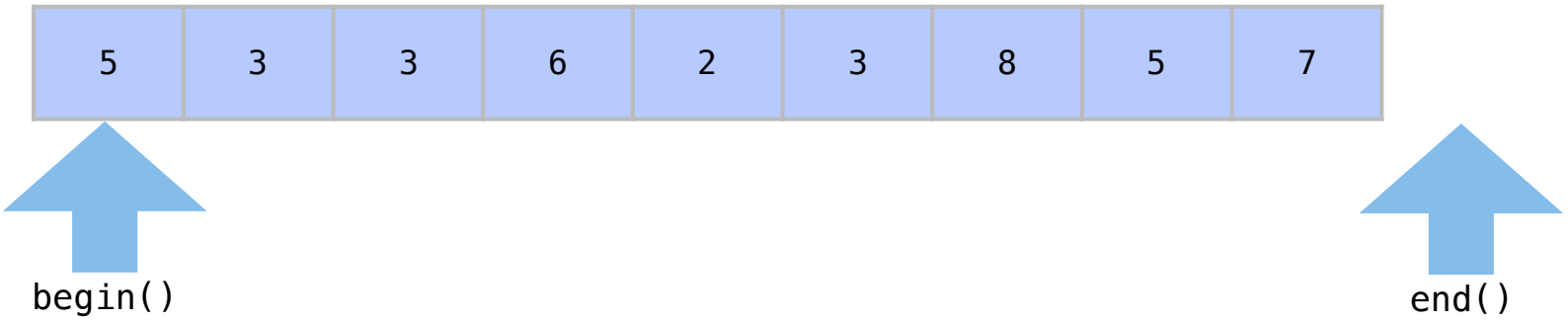
- Os iteradores são objetos que caminham (iteram) sobre elementos
  - Funcionam como um ponteiro especial para elementos de containers
  - Enquanto um ponteiro representa uma posição na memória, um iterador representa uma posição em um container
- São muito importantes pois permitem criar funções genéricas para qualquer tipo de container

Iteradores

- Funções básicas de qualquer iterador `i` são:
  - `*i` retorna o elemento na posição do iterador
  - `++i` avança o iterador para o próximo elemento
  - `==` e `!=` confere se dois iteradores apontam para mesma posição

Gerando iteradores

- Suponha um container chamado `c`.
- `c.begin()` retorna um iterador para o primeiro elemento deste container `c`
- `c.end()` retorna um iterador para uma posição após o último elemento do container `c`



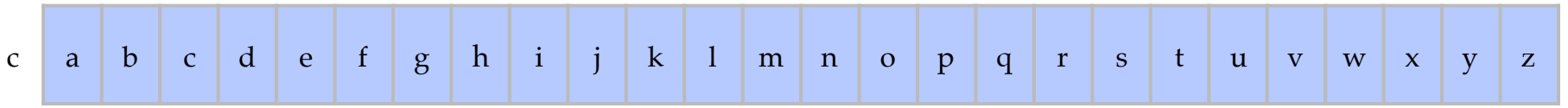
Iteradores - Exemplo

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

using namespace std;

int main(){
    list<char> c;
    for (char letra='a'; letra<='z'; letra++) {
        c.push_back(letra);
    }
    list<char>::iterator pos;
    for (pos = c.begin(); pos != c.end(); ++pos){
        cout << *pos << " ";
    }
    cout << endl;
}
```

Neste exemplo usamos um iterador para percorrer os elementos de um `list`.

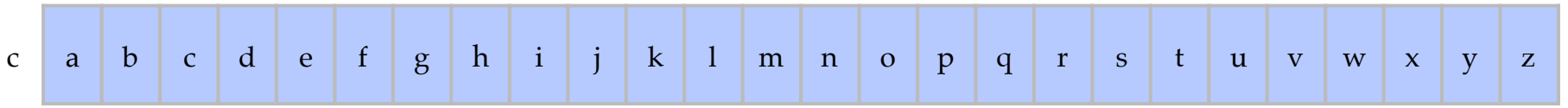
```cpp
#include <iostream>
#include <list>

using namespace std;

int main(){
    list<char> c;
    for (char letra='a'; letra<='z'; letra++) {
        c.push_back(letra);
    }
    list<char>::iterator pos;
    for (pos = c.begin(); pos != c.end(); ++pos){
        cout << *pos << " ";
    }
    cout << endl;
}
```

O list criado se chama `c` e guardará elementos do tipo `char`.




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

using namespace std;

int main(){
    list<char> c;
    for (char letra='a'; letra<='z'; letra++) {
        c.push_back(letra);
    }
    list<char>::iterator pos;
    for (pos = c.begin(); pos != c.end(); ++pos){
        cout << *pos << " ";
    }
    cout << endl;
}
```

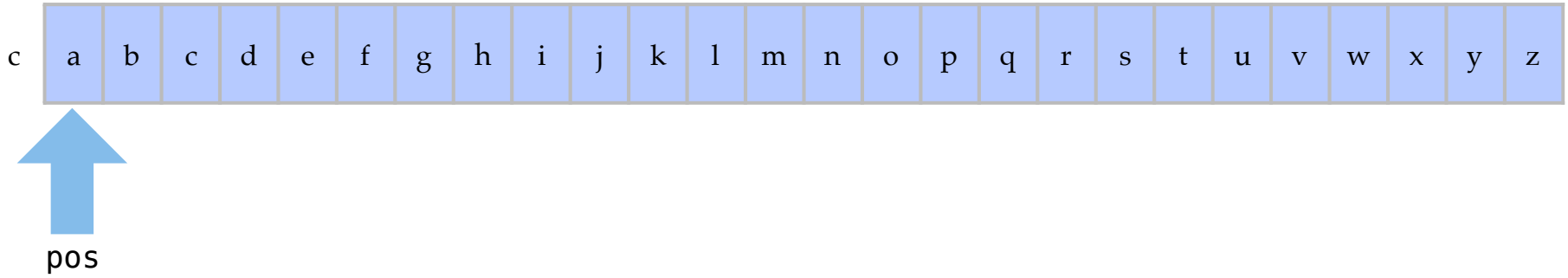
Neste trecho de código, colocamos no container c todos os chars entre a e z




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

using namespace std;

int main(){
    list<char> c;
    for (char letra='a'; letra<='z'; letra++) {
        c.push_back(letra);
    }
    list<char>::iterator pos;
    for (pos = c.begin(); pos != c.end(); ++pos){
        cout << *pos << " ";
    }
    cout << endl;
}
```

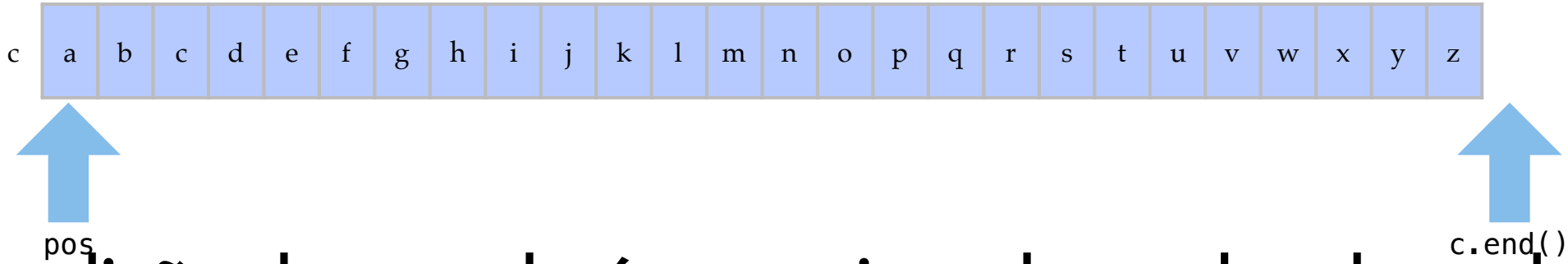
Criamos agora um iterador chamado `pos`. Repare que indicamos que `pos` é um iterador para um `list` de `char`.

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

using namespace std;

int main(){
    list<char> c;
    for (char letra='a'; letra<='z'; letra++) {
        c.push_back(letra);
    }
    list<char>::iterator pos;
    for (pos = c.begin(); pos != c.end(); ++pos){
        cout << *pos << " ";
    }
    cout << endl;
}
```



Inicializamos o `for` fazendo com que `pos` seja um iterador apontando para o primeiro elemento de `c`.

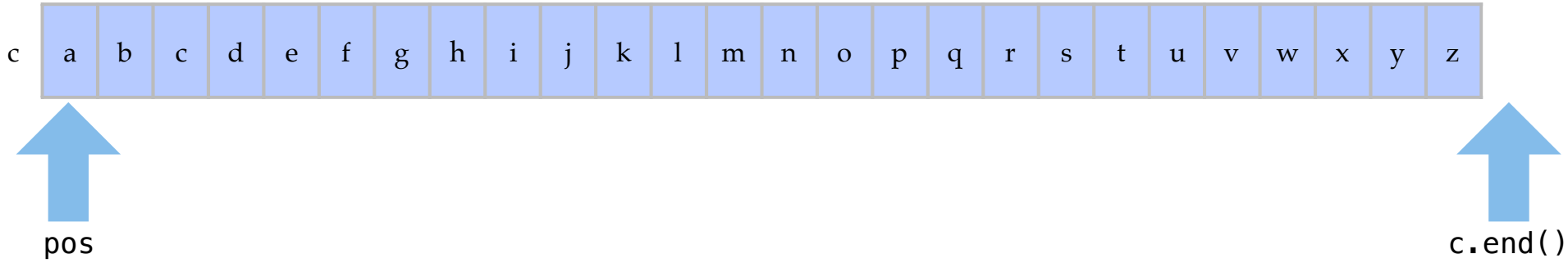
```cpp
#include <iostream>
#include <list>

using namespace std;

int main(){
    list<char> c;
    for (char letra='a'; letra<='z'; letra++) {
        c.push_back(letra);
    }
    list<char>::iterator pos;
    for (pos = c.begin(); pos != c.end(); ++pos){
        cout << *pos << " ";
    }
    cout << endl;
}
```



A condição de parada é que o iterador tenha chegado na posição `c.end()`, que é uma posição após o último elemento de `c`.

```
a
```

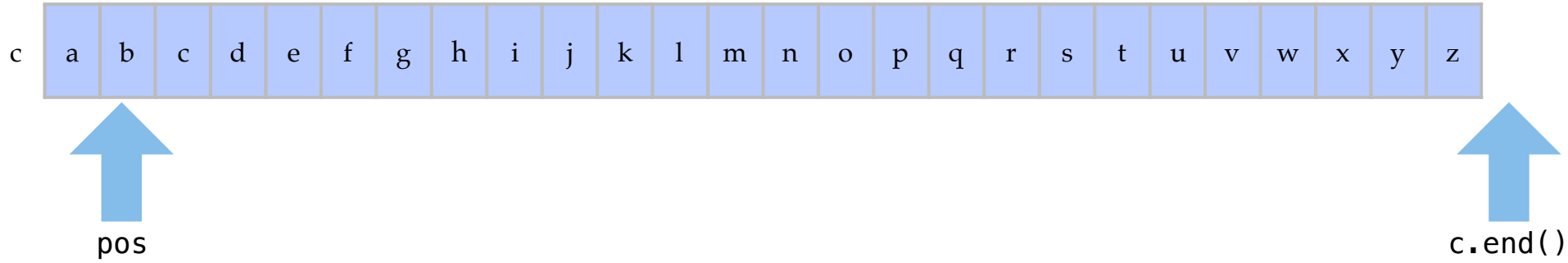
```cpp
#include <iostream>
#include <list>

using namespace std;

int main(){
    list<char> c;
    for (char letra='a'; letra<='z'; letra++) {
        c.push_back(letra);
    }
    list<char>::iterator pos;
    for (pos = c.begin(); pos != c.end(); ++pos){
        cout << *pos << " ";
    }
    cout << endl;
}
```



Retornamos então o elemento na posição do iterador com *pos.

```
a
```

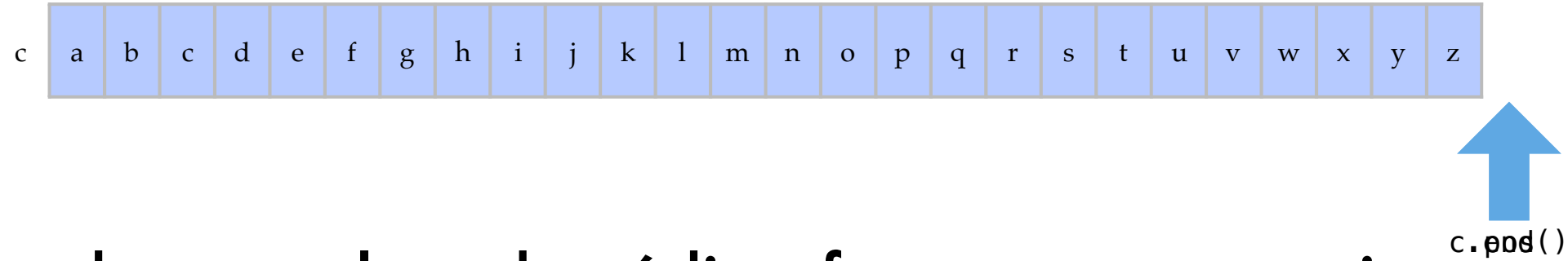
```cpp
#include <iostream>
#include <list>

using namespace std;

int main(){
    list<char> c;
    for (char letra='a'; letra<='z'; letra++) {
        c.push_back(letra);
    }
    list<char>::iterator pos;
    for (pos = c.begin(); pos != c.end(); ++pos){
        cout << *pos << " ";
    }
    cout << endl;
}
```



Independente do tipo de container, ++pos faz com que pos aponte para seu próximo elemento.

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

using namespace std;

int main(){
    list<char> c;
    for (char letra='a'; letra<='z'; letra++) {
        c.push_back(letra);
    }
    list<char>::iterator pos;
    for (pos = c.begin(); pos != c.end(); ++pos){
        cout << *pos << " ";
    }
    cout << endl;
}
```

```
a b c d e f g h i j k l m n o p q r
s t u v w x y z
```



O trecho completo de código faz com que consigamos imprimir todos os elementos de `c` e `pos` finalmente chega à posição `c.end()`.

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

using namespace std;

int main(){
    list<char> c;
    for (char letra='a'; letra<='z'; letra++) {
        c.push_back(letra);
    }
    list<char>::iterator pos;
    for (pos = c.begin(); pos != c.end(); ++pos){
        cout << *pos << " ";
    }
    cout << endl;
}
```

```
a b c d e f g h i j k l m n o p q r
s t u v w x y z
```

Repare que com iteradores, conseguimos acessar os elementos de um `list`, que não é um `container` de acesso aleatório.

Categorias de iteradores

Unidirecionais	Podem ir apenas para frente	++					
Bidirecionais	Podem ir para frente e para trás	++			−−		
De acesso aleatório	Além de ir para frente e para trás, pode-se fazer aritmética sobre eles	++	−−	+	−	<	>

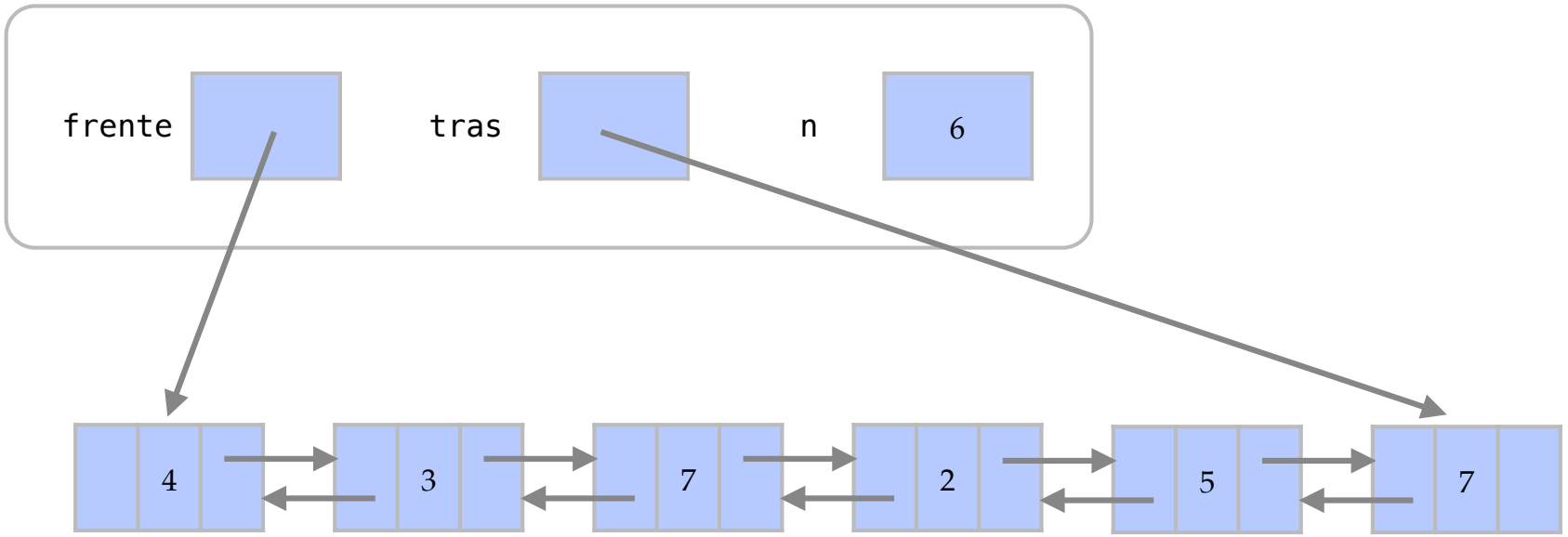
Categorias de iteradores

- O container list contém iteradores bidirecionais pois a partir de um elemento só temos acesso ao próximo elemento ou o elemento anterior
- Para os containers vector e deque, temos iteradores de acesso aleatório pois a partir de uma posição qualquer, basta uma operação aritmética para encontrar qualquer outro elemento

Insert

- Uma função dos containers de sequência que precisa de um iterador é a função `insert`.
- Esta função insere um elemento em uma posição qualquer da sequência.
- Para tal, a função precisa de um iterador para a posição onde queremos colocar o elemento e do elemento.

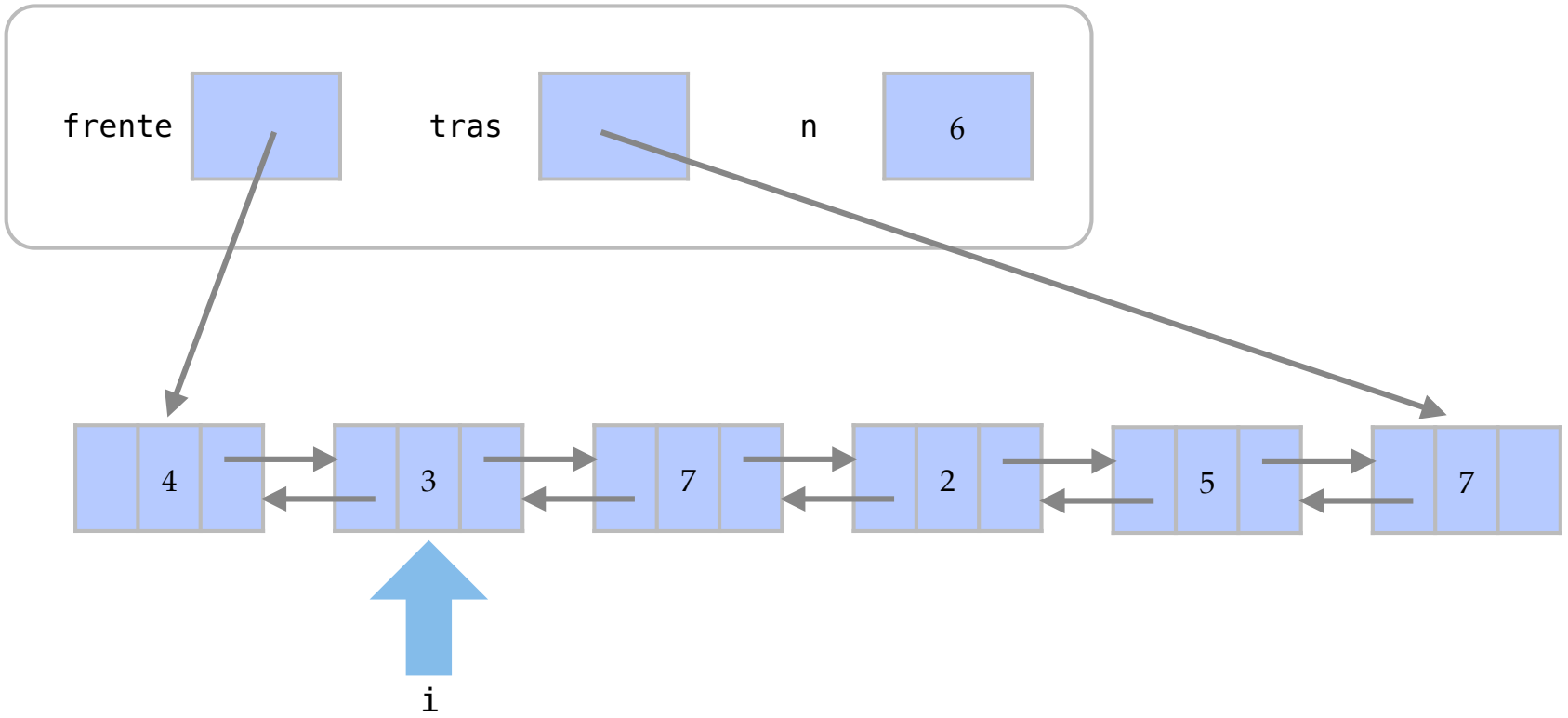
```
list<int> c;
```




```
list<int> c;
c.push_back(4);
c.push_back(3);
c.push_back(7);
c.push_back(2);
c.push_back(5);
c.push_back(7);
```

Imagine a estrutura interna de um container do tipo `list` chamado `c`.

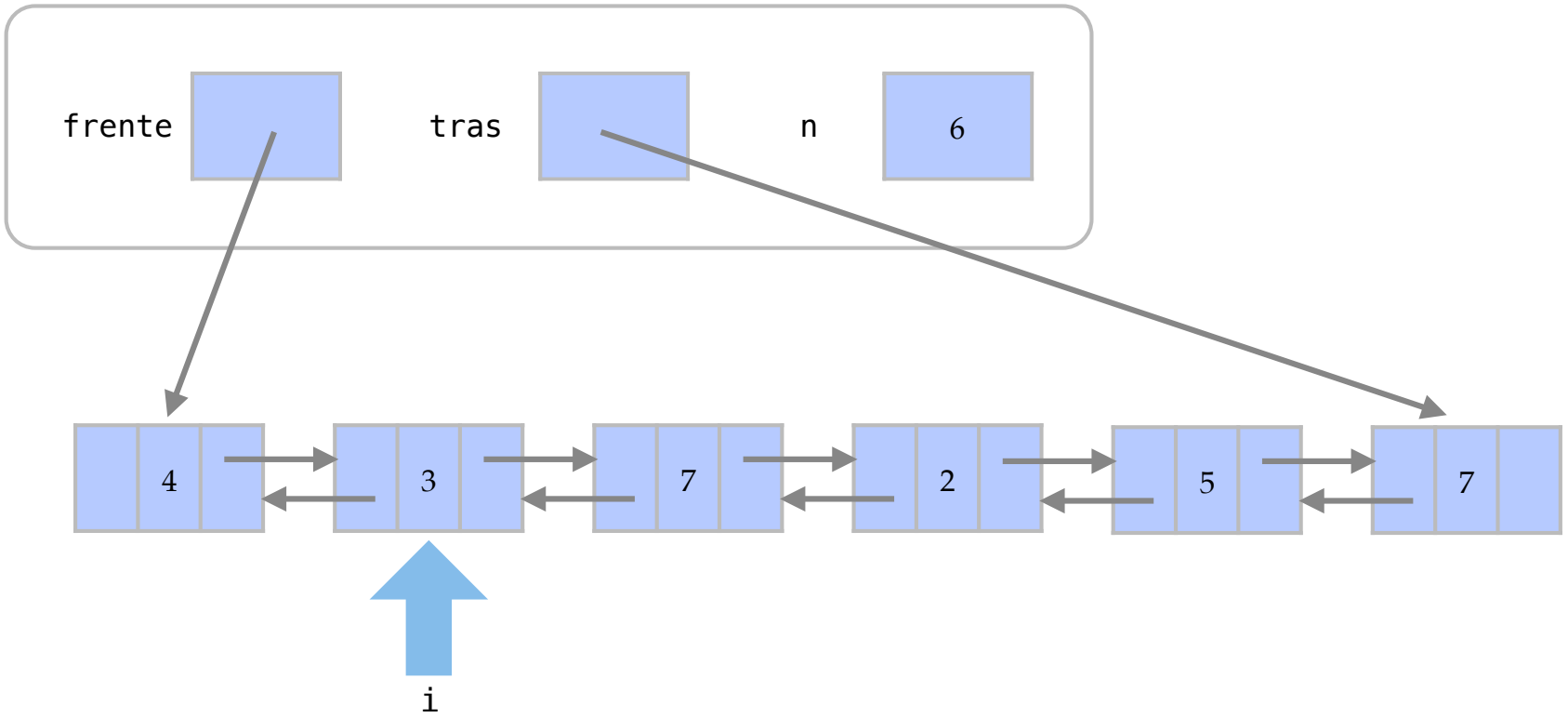
```
list<int> c;
```




```
list<int>::iterator i;
i = c.begin();
++i;
```

Imagine também que temos um iterador `i` apontando para a segunda posição de `c`.

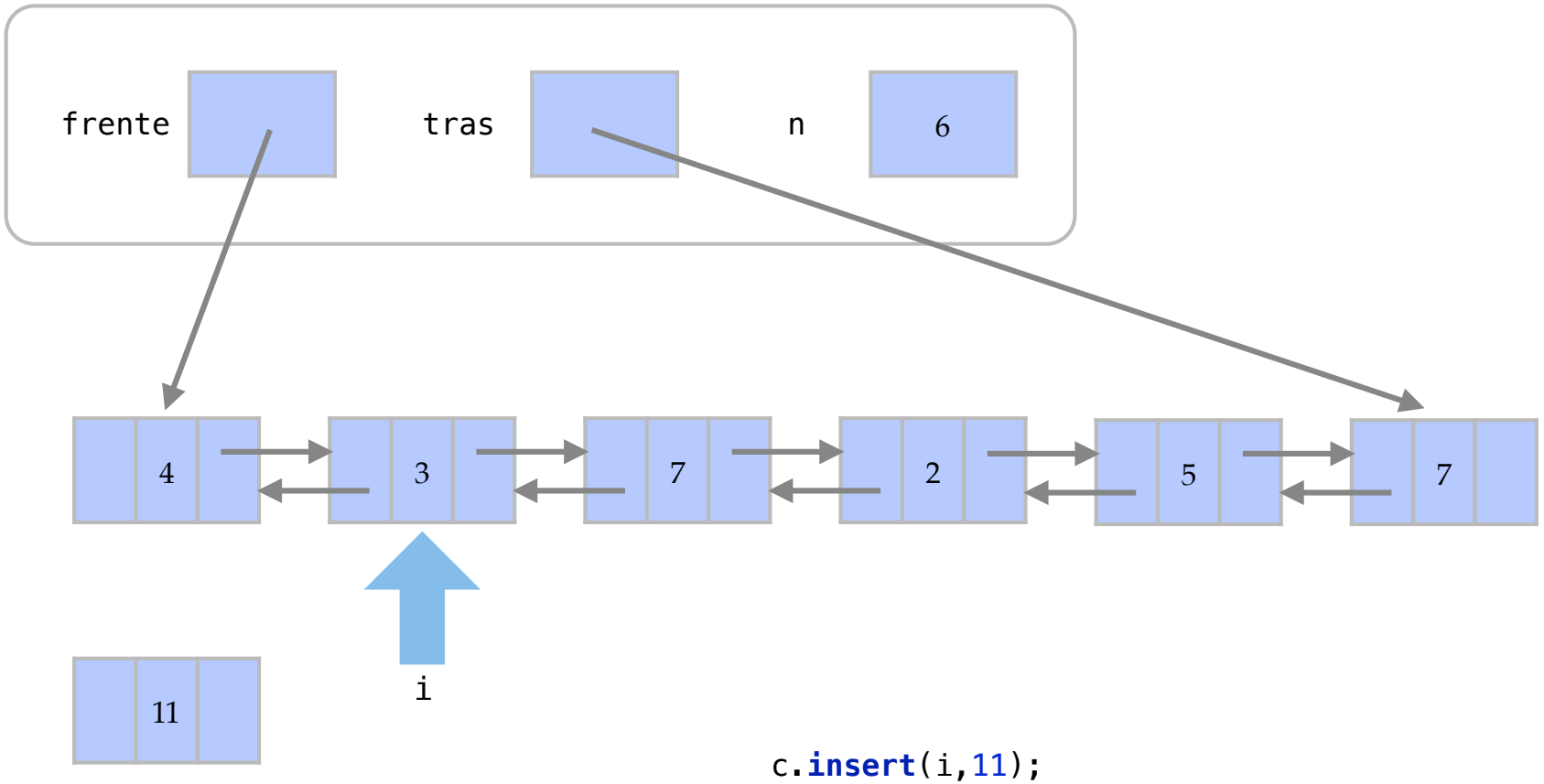
```
list<int> c;
```




```
c.insert(i,11);
```

Chamamos deste modo uma função para inserir um elemento 11 na posição `i` do container.

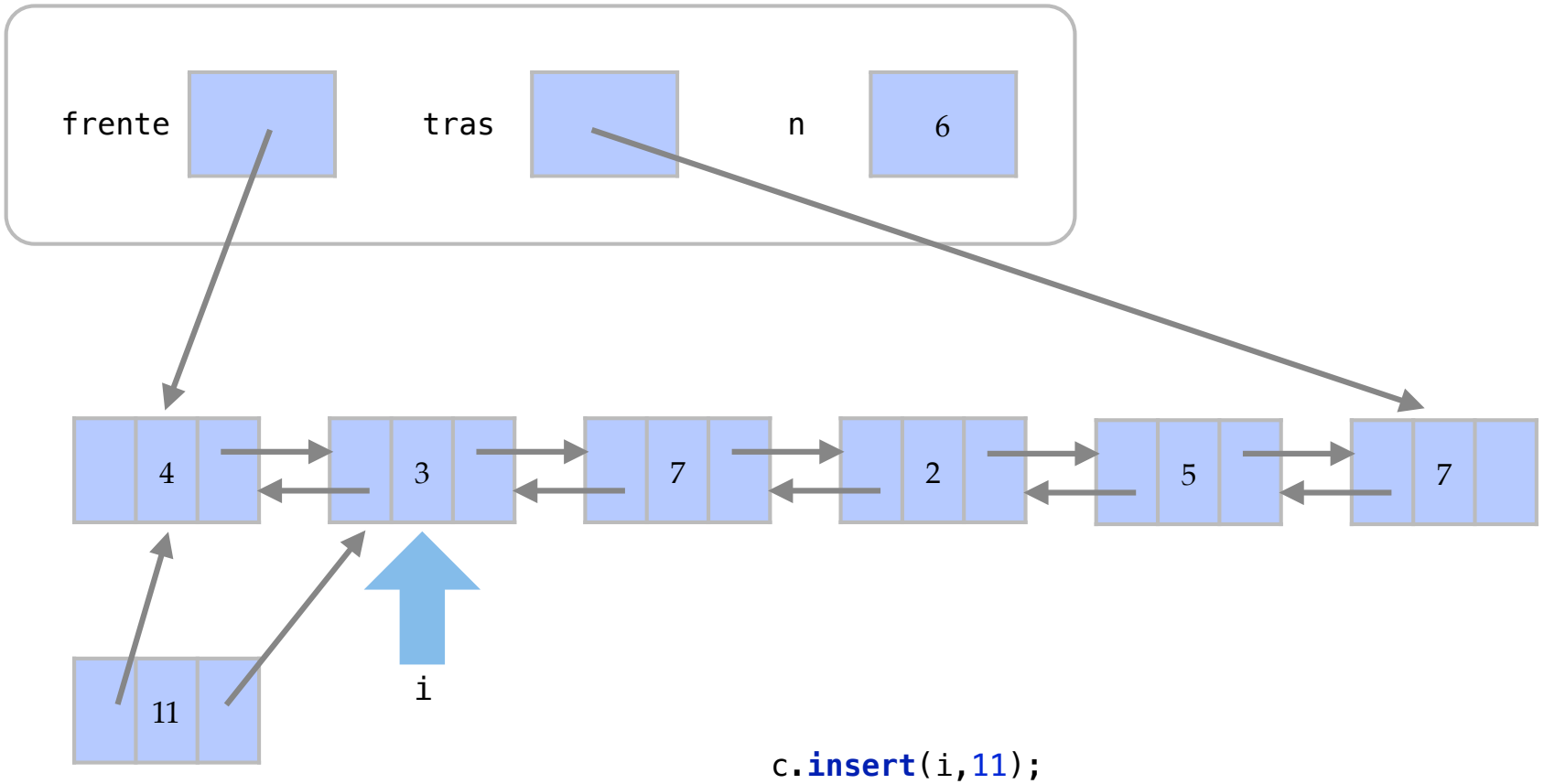
```
list<int> c;
```




```
c.insert(i,11);
```

Será alocada na memória uma nova célula para este elemento 11.

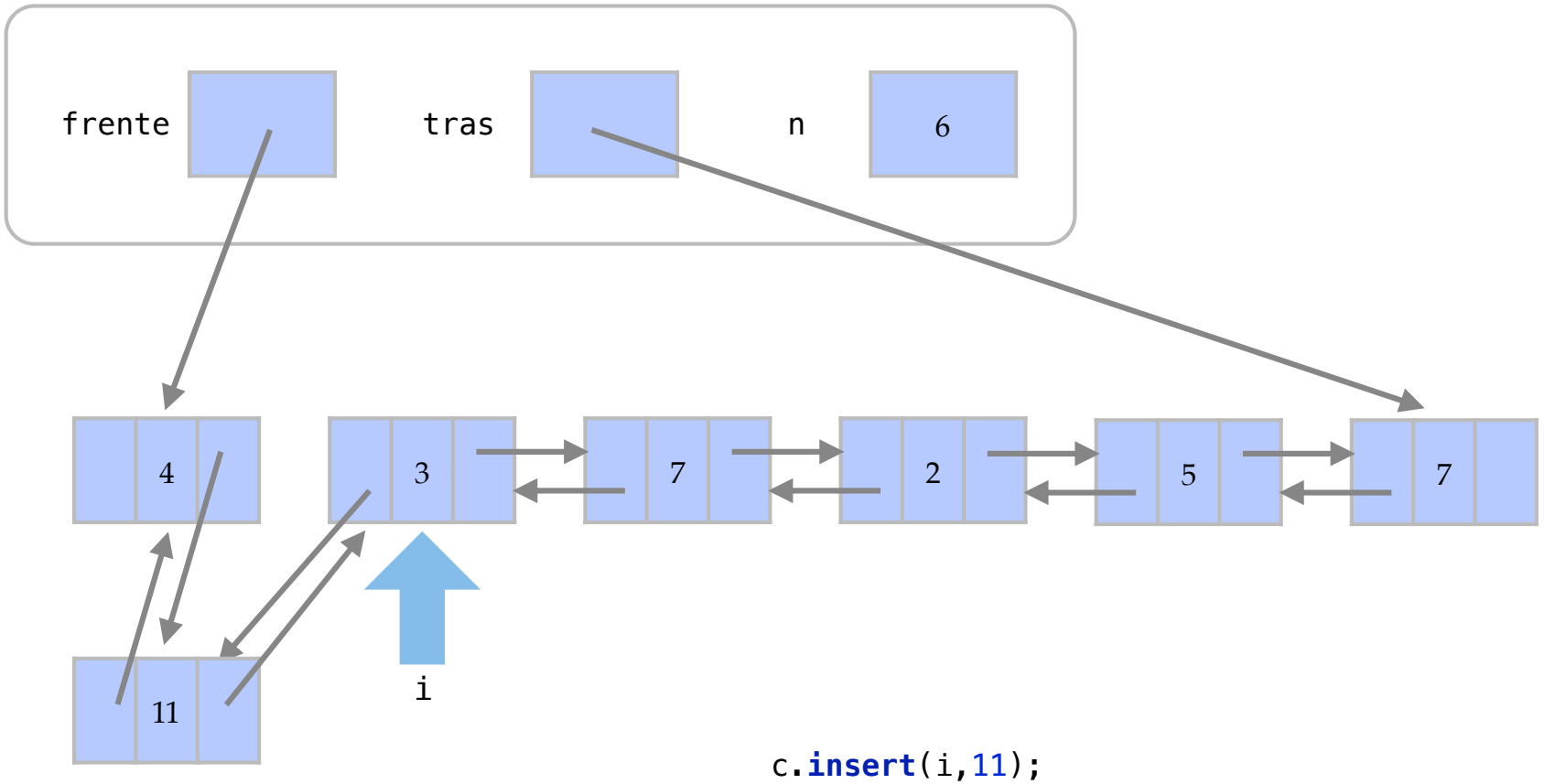
```
list<int> c;
```




```
c.insert(i,11);
```

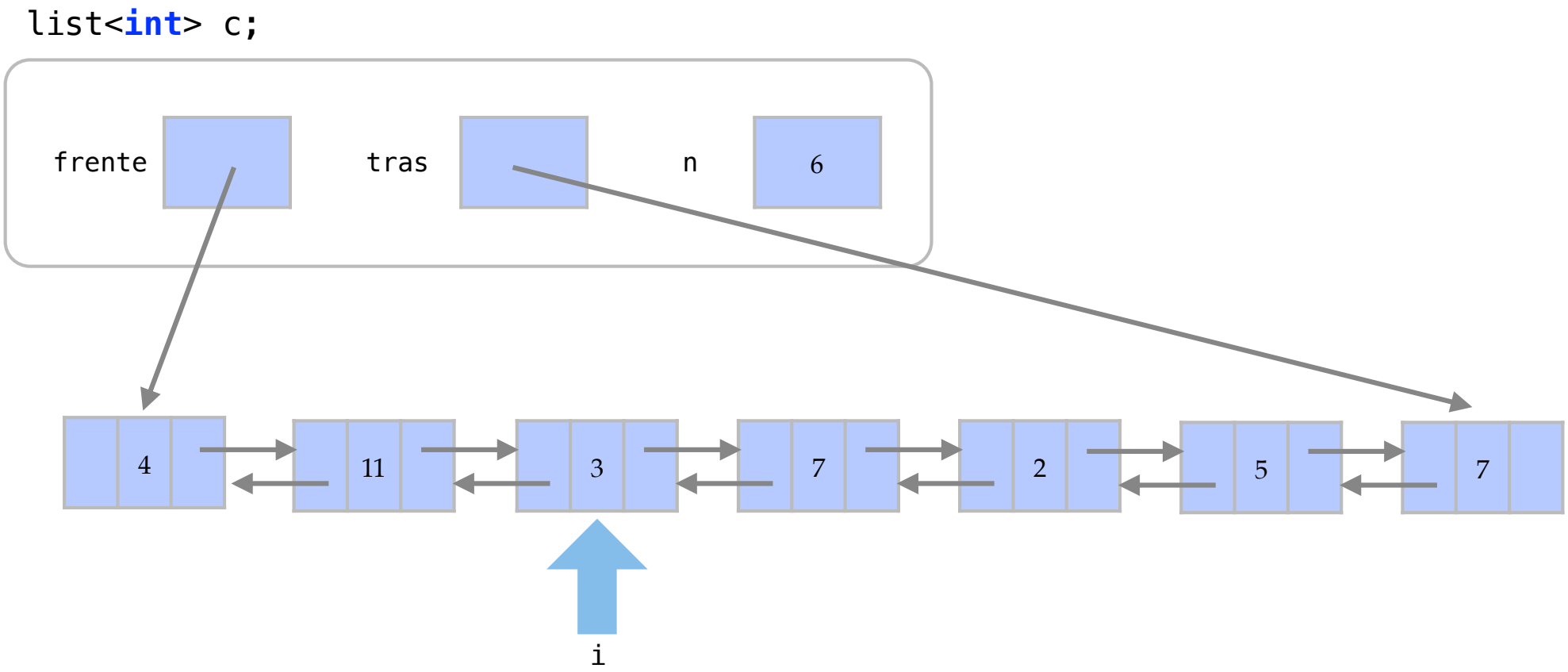
A nova célula aponta para a célula anterior à da posição `i` e para a própria célula da posição `i`.

```
list<int> c;
```




```
c.insert(i,11);
```

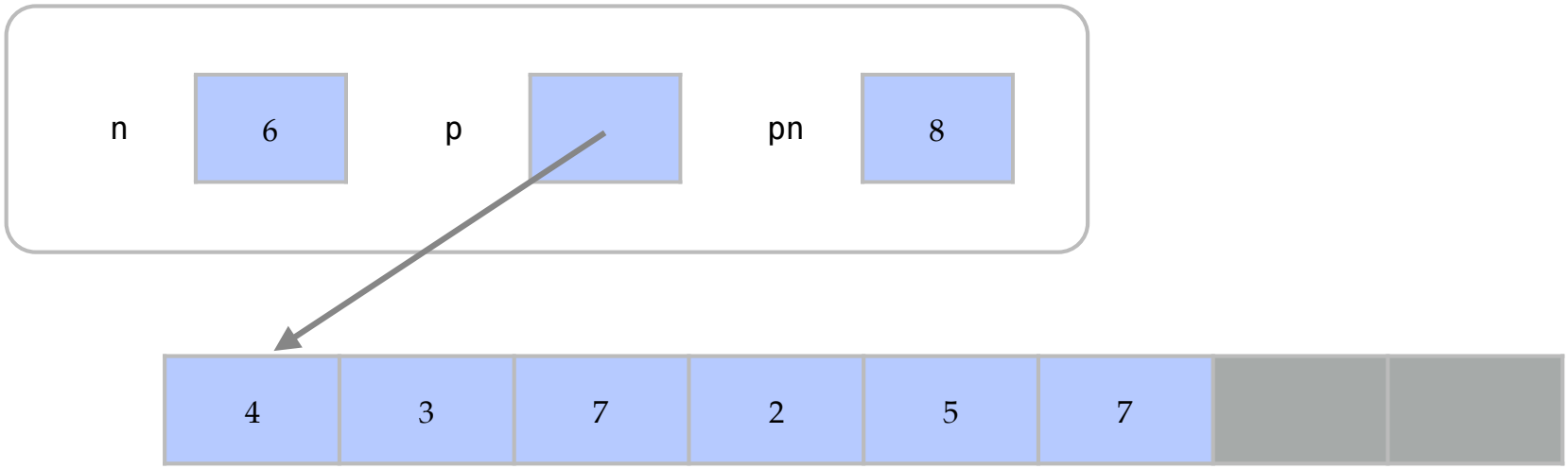
Alteramos então os ponteiros do elemento `i` e do elemento anterior a `i`.



Esta é exatamente a maior vantagem do container `list`. Fazer com que o iterador chegue à posição `i`, porém, pode ter custo *O(n)* pois list não tem acesso aleatório.

```
vector<int> c;
```

```
c.push_back(5);
```

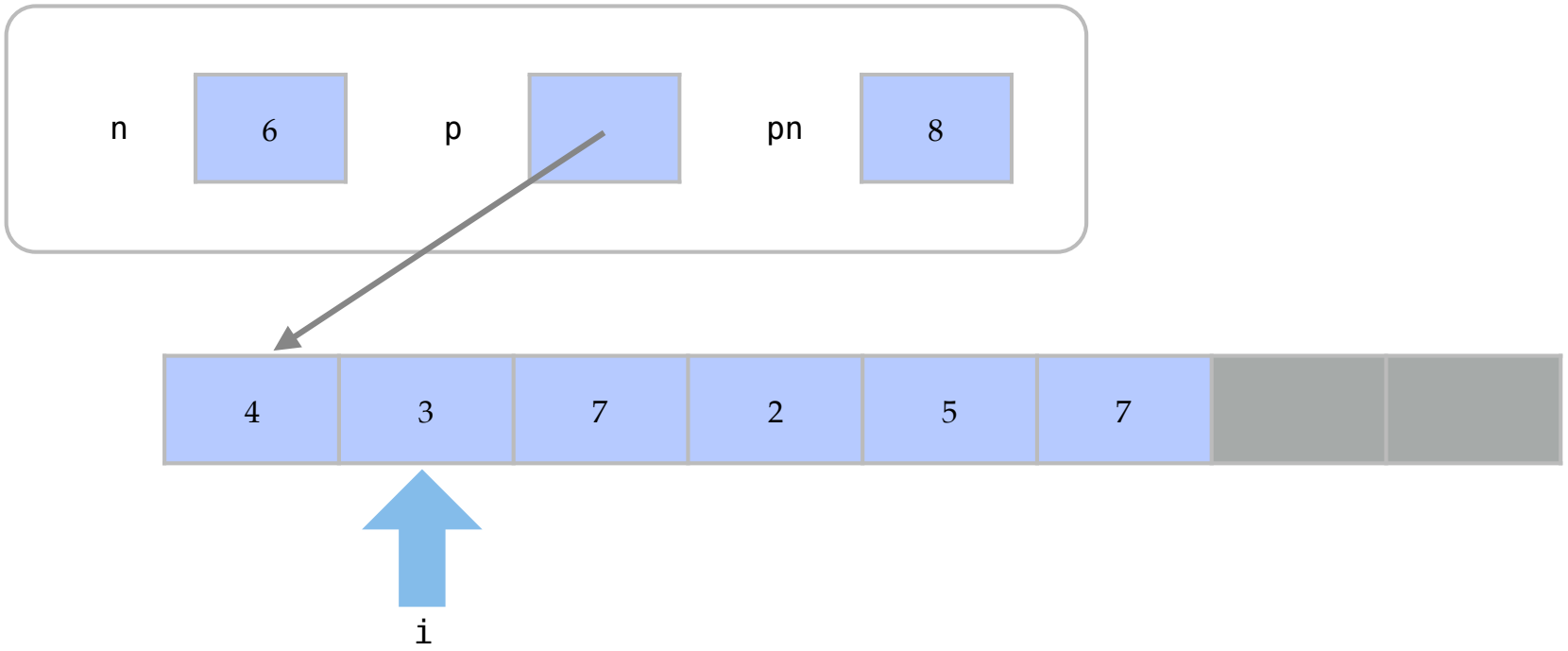



```
vector<int> c;
c.push_back(4);
c.push_back(3);
c.push_back(7);
c.push_back(2);
c.push_back(5);
c.push_back(7);
```

Imagine agora a estrutura interna da mesma sequência representada com um `vector`.

```
vector<int> c;
```

```
c.push_back(5);
```

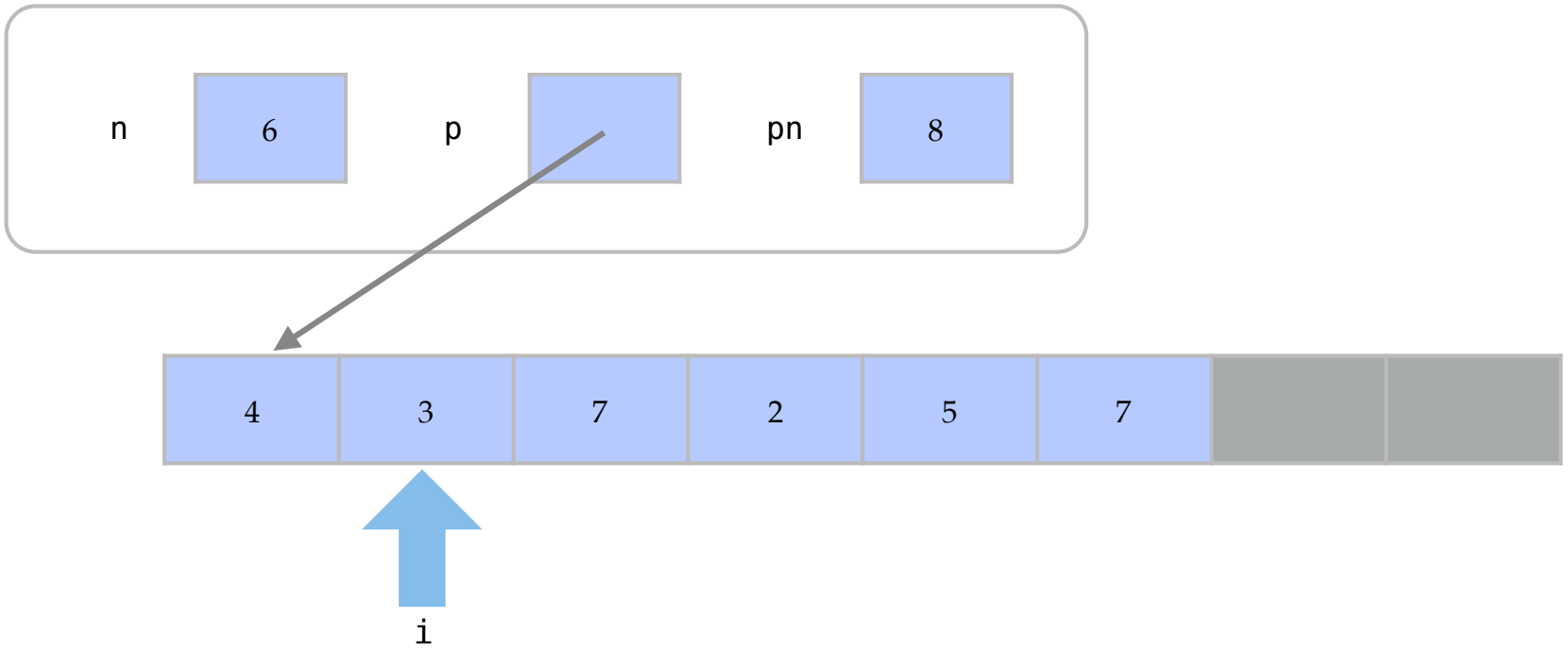



```
vector<int>::iterator i;
i = c.begin();
++i;
```

Imagine também um iterador `i` apontando para o segundo elemento da sequência.

```
vector<int> c;
```

```
c.push_back(5);
```

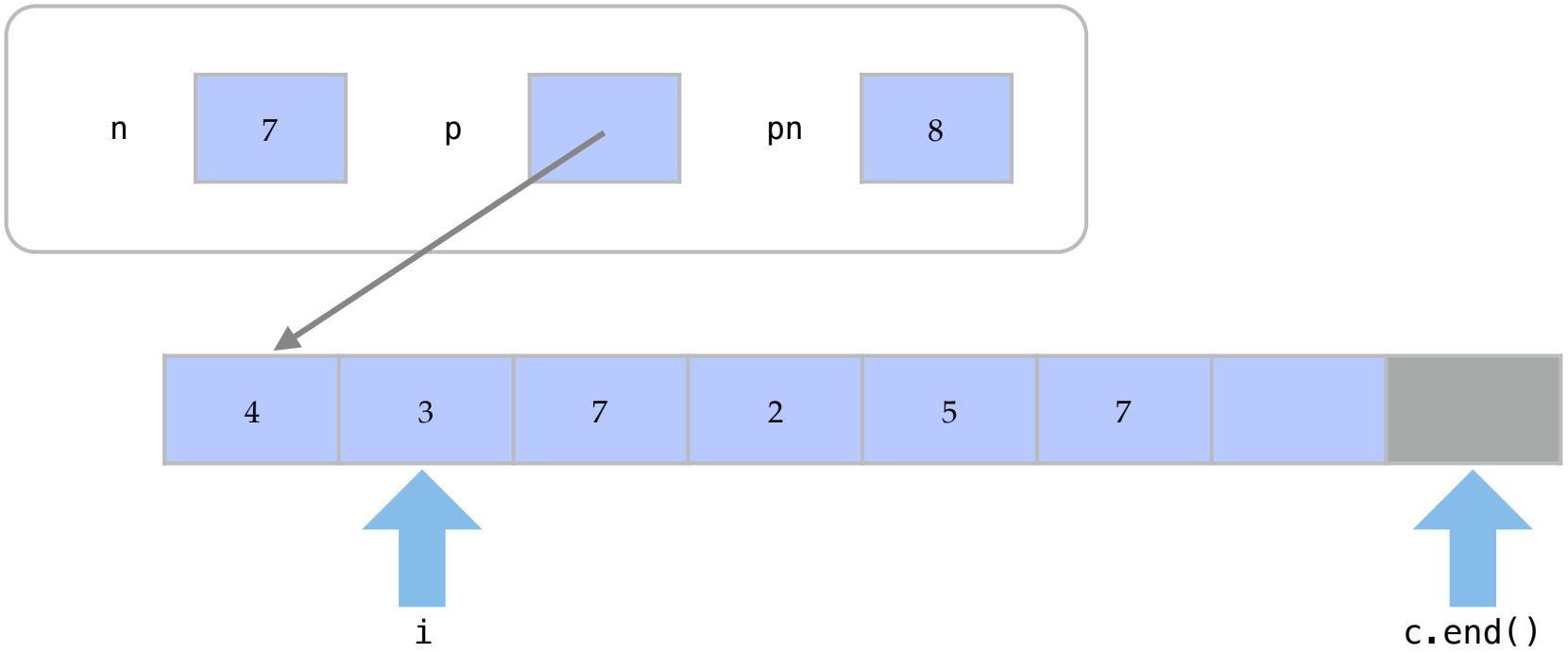



```
c.insert(i,11);
```

A função que insere o elemento 11 na posição `i` é mais complicada para um `vector`.

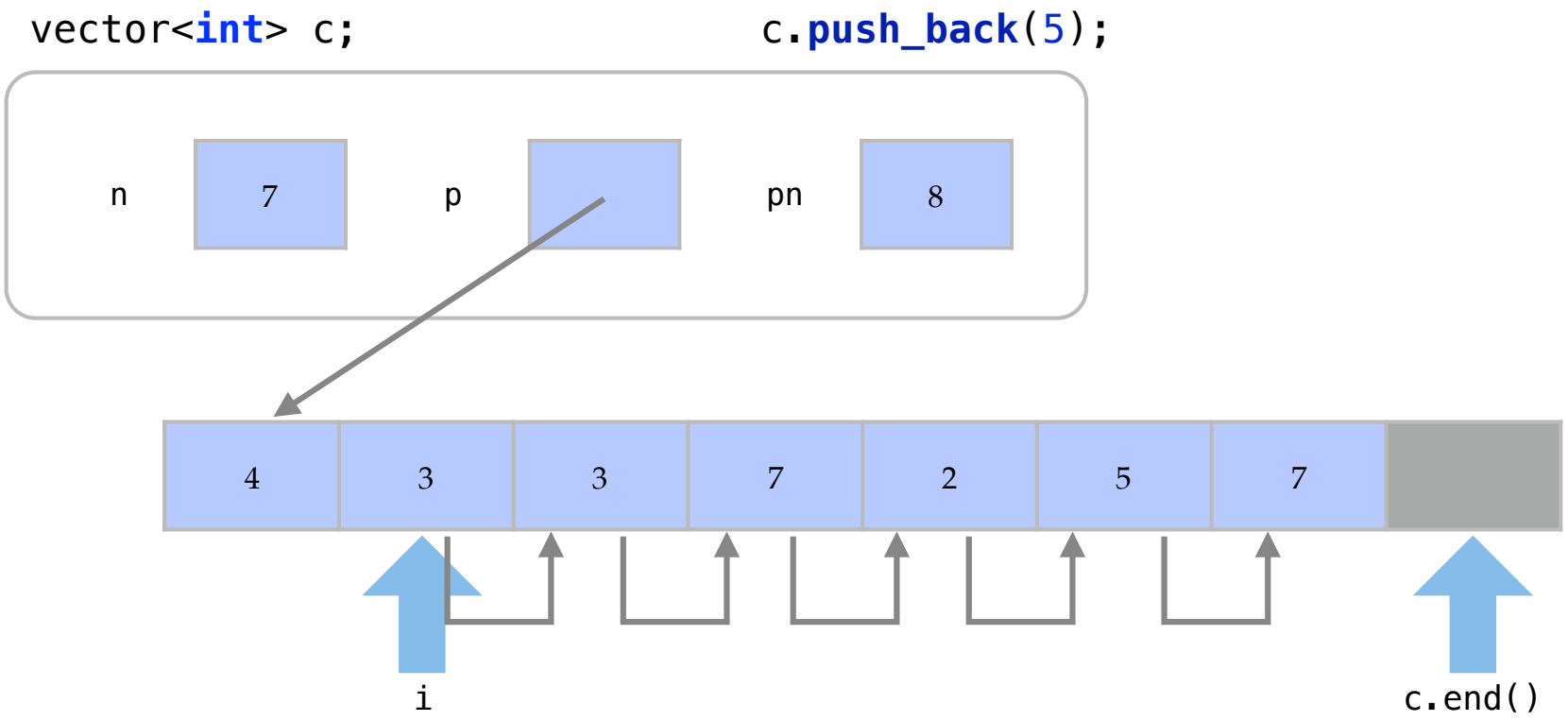
```
vector<int> c;
```

```
c.push_back(5);
```



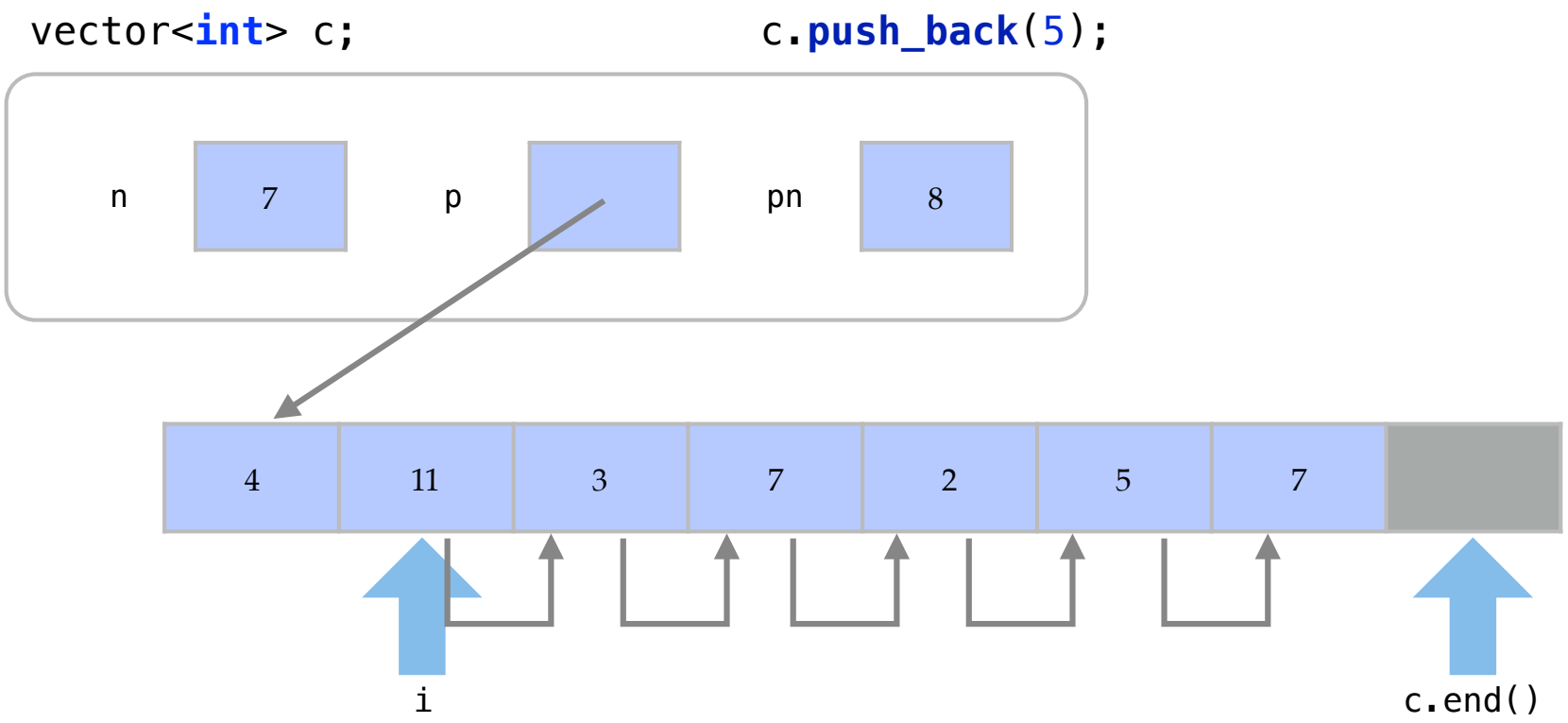

```
c.insert(i,11);
```

n é incrementado, abrindo espaço para mais um elemento. Em algumas situações, pode ser que um novo arranjo precise ser alocado.




```
c.insert(i,11);
```

A partir de `c.end() - 2`, todos os elementos entre `i` e `c.end() - 2` precisam ser movidos para a próxima posição do arranjo.

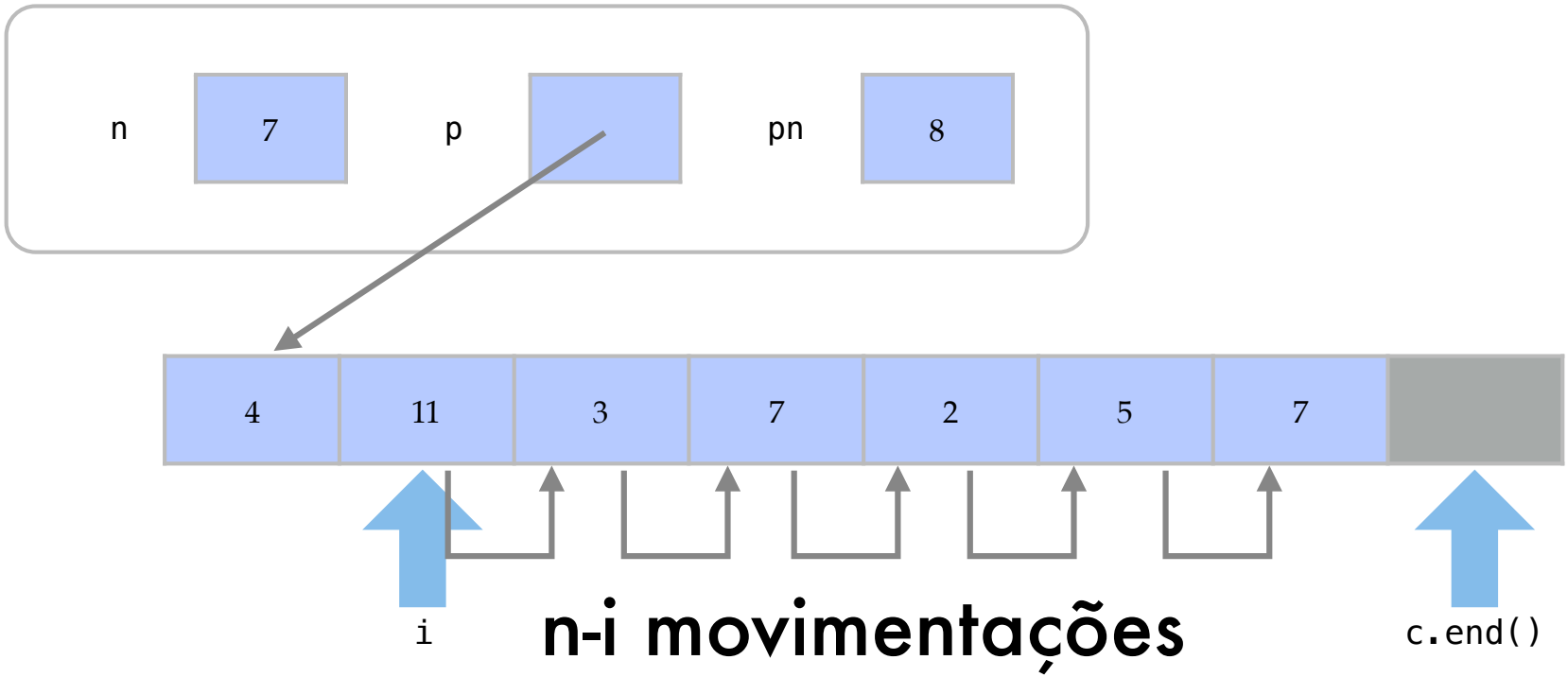



```
c.insert(i,11);
```

Agora sim o elemento 11 pode ser inserido na posição `i`. Devido a todas as movimentações esta operação de inserção no meio da sequência tem um custo $O(n)$.

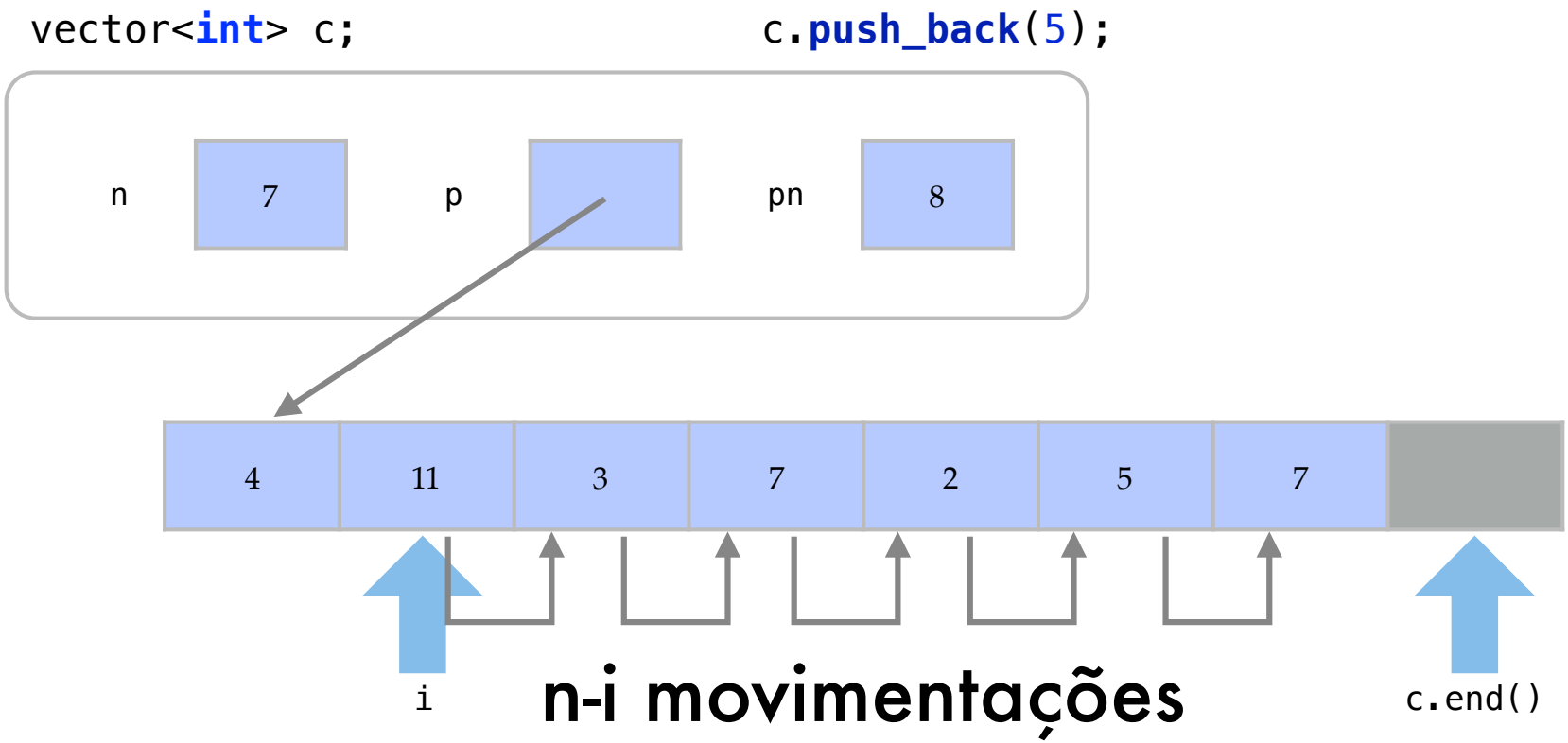
```
vector<int> c;
```

```
c.push_back(5);
```




```
c.insert(i,11);
```

A grande vantagem dos containers vector e deque vêm do fato de eles serem baseados em arranjos, o que faz com que os elementos fiquem em sequência na memória e possibilitem acesso aleatório.




```
c.insert(i,11);
```

Curiosamente, a grande desvantagem destas estruturas também se devem ao fato de utilizarem arranjos pois todos os elementos precisam ser movidos para que um elemento seja inserido no meio da sequência.

Insert - Erase

- De modo análogo à função `insert(i, elem)`, existe a função `erase(i)` que remove do container o elemento da posição `i`
- A função `insert` pode também receber 2 iteradores em vez de um elemento para inserir vários elementos de uma vez
- A função `erase` pode também receber 2 iteradores em vez de um para remover vários elementos de uma vez