Chapter 6
Lists and Strings

Contents Points

- List Definition
- Implementation of Lists
  - Class Templates
  - Contiguous Implementation
  - Simply Linked Implementation
  - Doubly Linked Lists
  - Comparison of Implementations
- Strings

List Definition

- A list of elements of type T is a finite sequence of elements of T together with the following operations:
  - Construct the list, leaving it empty
  - Determine whether the list is empty or not
  - Determine whether the list is full or not
  - Find the size of the list
  - Clear the list to make it empty
  - Insert an entry at a specified position of the list
  - Remove an entry from a specified position in the list.
  - Retrieve an entry from a specified position in the list.
  - Replace the entry at a specified position in the list.
  - Traverse the list, performing a given operation on each entry.

List Definition

- Some Important Concepts
  - ADT (Abstract Data Type)
  - STL (Standard Template Library)
  - STL classes list and vector
- Method Specifications

List::List();

Post: The List has been created and is initialized to be empty.

List Definition

- Method Specifications

```
void List :: clear( );
```

Post: All List entries have been removed; the List is empty.

```
bool List :: empty( ) const;
```

Post: The function returns **true** or **false** according to whether the List is empty or not.

List Definition

- Method Specifications

bool List :: full( ) **const;**

Post: The function returns **true** or **false** according to whether the List is full or not.

int List :: size( ) **const;**

Post: The function returns the number of entries in the List.

1、关于position（位序）的含义

将长度为n(n>0)的非空表记为如下形式：$L=(a_0,a_1,a_2,\ldots a_i,\ldots a_{n-1})$，其中$a_0$的position是0，$a_i$的position是i。

2、根据position进行元素的插入、删除、读写等操作。

```
Error_code List :: insert(int position, const List_entry &x);
```

postcondition: If the List is not full and $0 \leq$ position $\leq n$, where n is the number of entries in the List, the function succeeds: Any entry formerly at position and all later entries have their position numbers increased by 1, and x is inserted at position in the List.
Else: The function fails with a diagnostic error code.

```
Error_code List :: remove(int position, List_entry &x);
```

postcondition: If $0 \leq$ position $< n$, where n is the number of entries in the List, the function succeeds: The entry at position is removed from the List, and all later entries have their position numbers decreased by 1. The parameter x records a copy of the entry formerly at position.
Else: The function fails with a diagnostic error code.

```
Error_code List :: retrieve(int position, List_entry &x) const;
```

postcondition: If $0 \leq$ position $< n$, where n is the number of entries in the List, the function succeeds: The entry at position is copied to x; all List entries remain unchanged.
Else: The function fails with a diagnostic error code.

```
Error_code List :: replace(int position, const List_entry &x);
```

postcondition: If $0 \leq$ position $< n$, where n is the number of entries in the List, the function succeeds: The entry at position is replaced by x; all other entries remain unchanged.
Else: The function fails with a diagnostic error code.

```
void List :: traverse(void (*visit)(List_entry &));
```

postcondition: The action specified by function *visit has been performed on every entry of the List, beginning at position 0 and doing each in turn.

Class Template

- Define format

```
template <class Entry_type_param>
class List{
    //Add in member information for the class
};
```

- Usage way

```
List <real_type> list_obj;
```

example:

```
List <int> first_list;
 List<char> second_list;
```

Contiguous Implementation

- template definition

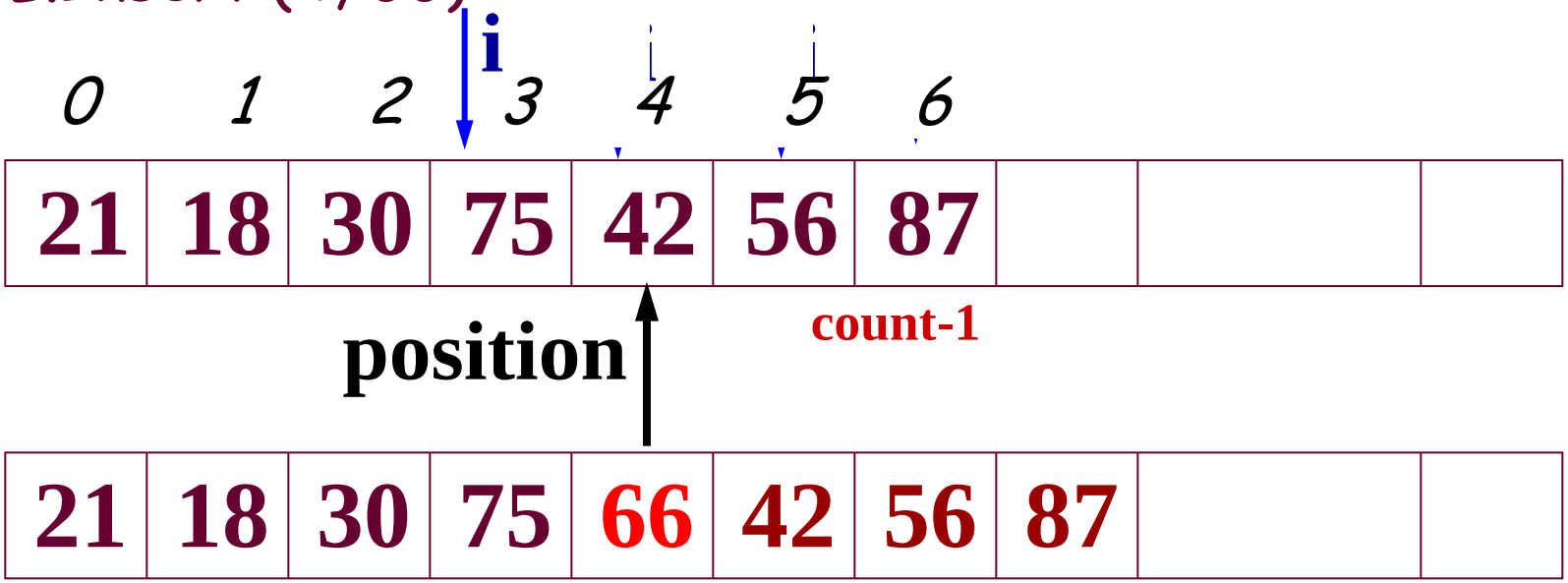
```
template <class List_entry>
class List{
        List();  //构造函数
        int size() const;
        bool full() const;
        bool empty() const;
        void clear();
        void traverse(void (*visit)(List_entry& ));
        Error_code retrive(int position,List_entry& x) const;
        Error_code replace(int position, const List_entry& x);
        Error_code remove(int position, List_entry& x);
        Error_code insert(int position, const List_entry& x);
private: //私有成员数据
        int count;
        List_entry entry[max_list];  //所谓contiguous是指采用数组方式存储
};
```

Contiguous Implementation

- Realization of some member functions

```
template <class List_entry>
int List <List_entry>::size() const{
    return count;
}
```

L.Insert (4, 66)




```
for (int i=count-1;i>=position; i--)
        entry[i+1]=entry[i];  //移动元素（从后向前）
        entry[position]=x;  //放入元素
```

```
template <class List_entry>
Error_code List<List_entry>::insert(int position,
const List_entry& x)
{
   if (full())          return overflow;        //溢出

   if (position<0||position>count)
                        return range_over; //插入点错误

   for (int i=count-1;i>=position; i--)
                        entry[i+1]=entry[i];  //移动元素（从后向前）

   entry[position]=x;     //插入元素
   count++; //增加统计个数
   return success;
}
```

编写算法的步骤：

1、确定详细功能。对于类的方法，此步骤为类设计时所作事情。

包括考虑函数的入口参数（为完成此功能须从外界获取得到的信息。）

出口参数（函数执行结束后向外界返回的信息。）

2、分析，通常可借助图示。

3、写出一般情形下函数的主体执行部分。

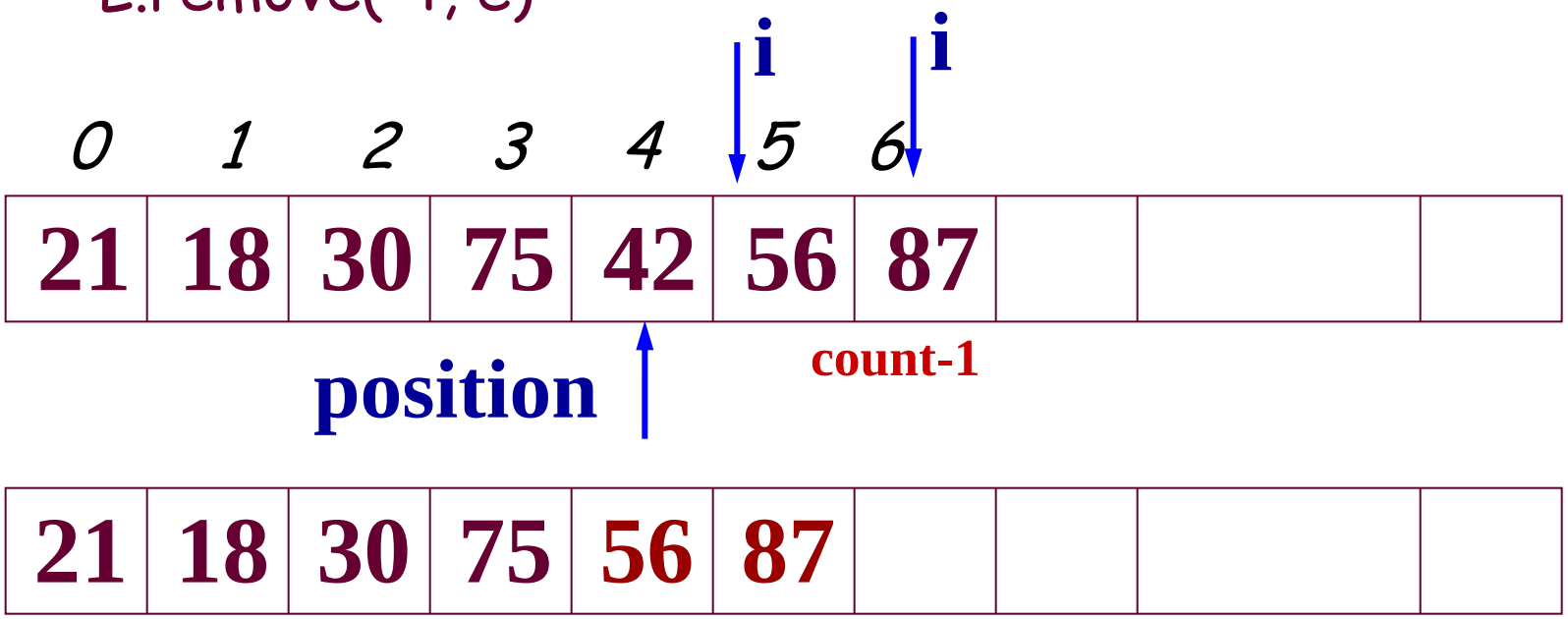
4、检查特殊情况。

5、检查参数合法性。

Contiguous Implementation

```
template <class List_entry>
Error_code List<List_entry> :: remove(int position, List_entry &x)
{
    if (count == 0) return underflow;
    if (position < 0 || position >= count) return range_error;
    x = entry[position];
    for (int i=postion+1；  i<count;i++) {
            entry[i-1] = entry[i];
    }
    count--;
    return success;
}
```

L.remove(4, e)




```
for (int i=postion+1;  i<count;i++) {
        entry[i-1] = entry[i];
}
```

```
template <class List_entry>

Error_code List<List_entry> :: retrieve(int position, List_entry &x)
const

//如果position值非法，返回出错信息，否则根据position的值，取得表中的第position个元素，并由x返回。

{

if (position < 0 || position >= count)

        return range_error;

x = entry[position];

return success;

}
```

在顺序结构中，元素的获取是随机得到的，只要根据数组下标就可以计算出该元素的存储位置，顺序存储结构的特征是随机存取。

Contiguous Implementation

- Realization of some member functions

```
template <class List_entry>
void  List<List_entry>::traverse(void
      (*visit)(List_entry&)){
      for (int i=0;i<count;i++)
            (*visit)(entry[i]);
//此处，visit函数表示指定的遍历时对每个表元素
      执行的具体动作
}
```

- void Print_out(List_entry& x)
- {cout<<x;
- }

In processing a contiguous list with n entries:

- insert *and* remove *require time approximately proportional to n.*
- List, clear, empty, full, size, replace, *and* retrieve *operate in constant time.*
- the time requirement of traverse is approximately proportional to *n*.(对每个元素访问一次，基本操作执行次数为n次。)

- Advantage
  - Simple
  - Random access ---适合于读取操作
- Disadvantage
  - Need move entries when to be inserted or deleted
    - Entries are individually small
    - the size of the list is small
  - Overflow
    - max_list