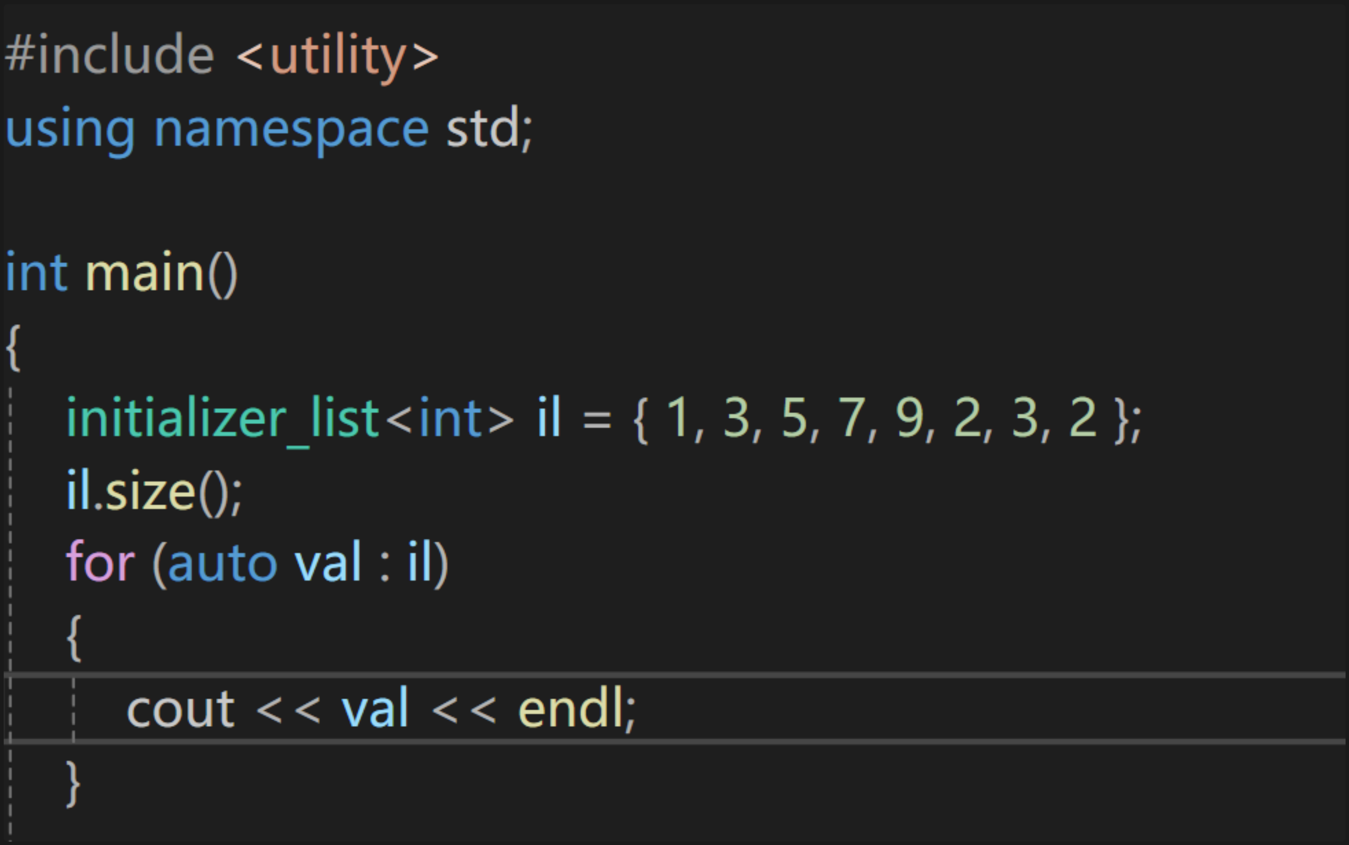
1. 动态数组补充



```cpp
#include <utility>
using namespace std;

int main()
{
    initializer_list<int> il = { 1, 3, 5, 7, 9, 2, 3, 2 };
    il.size();
    for (auto val : il)
    {
        cout << val << endl;
    }
```

Figure 1-1

收初始化列表,但是只支持相同类型

1.1 CVector.h

```cpp
#pragma once

#include <assert.h>
#include <memory>
#include <utility>

// 动态数组类
template<typename T>
class CVector
{
public:
    CVector();
    CVector(std::initializer_list<T> il);
    CVector(size_t nSize);
    CVector(const CVector& obj);
    CVector(CVector&& obj);
    CVector& operator=(const CVector& obj);
    virtual ~CVector();

    void PushHeard(const T& val);// 头插
    void PushTail(const T& val);// 尾插
    void Insert(size_t nIdx, const T& val);// 指定位置插入

    void PopHead();// 头删
    void PopTail();// 尾删
    void Delete(size_t nIdx);// 指定位置删除

    T& operator[](size_t nIdx);// 修改

    int Find(size_t val) const;// 查询，返回下标

    void Sort();// 排序,默认从小到大

    bool IsEmpty() const;// 是否为空

    size_t GetCount() const;// 获取元素个数

    void Clear();// 清空
private:
    void Reset();

private:
    T* m_pBuff;// 存储元素的缓冲区
    size_t m_nBuffLen;// 缓冲区大小
    size_t m_nCount;// 类型T的元素个数
};

template<typename T>
void CVector<T>::Reset()
{
    m_pBuff = nullptr;
    m_nBuffLen = 0 * sizeof(T)*m_nCount;
    m_nCount = 0;
}

template<typename T>
CVector<T>::CVector()
{
    Reset();
}
template<typename T>
CVector<T>::CVector(std::initializer_list<T> il)
{
    Reset();
    for (auto val : il)// 调拷贝构造
    {
        PushTail(val);
    }
}

template<typename T>
CVector<T>::CVector(size_t nSize)// 多少个T。也就是Count
{
        m_pBuff = new T[nSize];
        if (m_pBuff = nullptr)
        {
            return;
        }
        m_nBuffLen = nSize * sizeof(T) * m_nCount;
        m_nCount = nSize;
}

template<typename T>
CVector<T>::CVector(const CVector& obj)
```

```cpp
{
    *this = obj;// 调用=运算符重载
}

template<typename T>
CVector<T>::CVector(CVector<T>&& obj)
{
    m_pBuff = obj.m_pBuff;
    m_nBuffLen = obj.m_nBuffLen;
    m_nCount = m_nCount;

    obj.Reset();
}

template<typename T>
CVector<T>& CVector<T>::operator=(const CVector<T>& obj)
{
    if (*this != obj)
    {
        if (obj.IsEmpty())
        {
            Clear();
            return *this;
        }
        else
        {
            if (m_nBuffLen < obj.m_nBuffLen)
            {
                T* pNewBuff = nullptr;
                pNewBuff = new T[obj.m_nCount];
                if (pNewBuff != nullptr)
                {
                    return *this;
                }
                // memcpy(pNewBuff, obj.m_pBuff, obj.m_nBuffLen * sizeof(T));
                for (size_t i = 0; i < obj.m_nCount; i++)
                {
                    m_pBuff[i] = obj.m_pBuff[i];
                }
                m_pBuff = pNewBuff;
                m_nBuffLen = obj.m_nBuffLen;
                m_nCount = obj.m_nCount;
            }
            else
            {
                // memcpy(m_pBuff, obj.m_pBuff, obj.m_nBuffLen * sizeof(T));
                for (size_t i = 0; i < obj.m_nCount; i++)
                {
                    m_pBuff[i] = obj.m_pBuff[i];
                }
                m_nBuffLen = obj.m_nBuffLen;
                m_nCount = obj.m_nCount;
            }
        }
    }
    return *this;
}

```

```cpp
template<typename T>
CVector<T>::~CVector()
{
    Clear();
}

template<typename T>
void CVector<T>::PushHeard(const T& val)
{
    Insert(0, val);
}

template<typename T>
void CVector<T>::PushTail(const T& val)
{
    Insert(m_nCount, val);
}

template<typename T>
void CVector<T>::Insert(size_t nIdx, const T& val)
{
    // 检查
    assert(nIdx <= m_nCount);
    // 判断内存是否为空
    if (m_pBuff == nullptr)
    {
        m_pBuff = new T(val);
        m_nBuffLen = 1 * sizeof(T) * m_nCount;
        m_nCount = 1;
        return;
    }

    size_t nNewLen = m_nCount * 2;// 只是两倍Count的空间
    T* pNewBuff = new T[nNewLen];
    if (pNewBuff == nullptr)
    {
        return;
    }
    // 拷贝原来的数据
    // memcpy(pNewBuff, m_pBuff, m_nBuffLen * sizeof(T));// 浅拷贝
    for (size_t i = 0; i < m_nCount; i++)
    {
        pNewBuff[i] = m_pBuff[i];// 调用=运算符重载
    }
    // 删除原来的
    m_nCount == 1 ? delete m_pBuff : delete[] m_pBuff;
    m_pBuff = pNewBuff;
    m_nBuffLen = nNewLen;

    // 移动数据
    // memcpy(m_pBuff + nIdx + 1, m_pBuff + nIdx, (m_nCount - nIdx) * sizeof(T));
    for (size_t i = m_nCount; i > nIdx; --i)
    {
        m_pBuff[i] = m_pBuff[i - 1];
    }
    // 数据赋值
    m_pBuff[nIdx] = val;
    // 更新元素个数
    m_nCount++;
}

template<typename T>
void CVector<T>::PopHead()
{
    Delete(0);
}

template<typename T>
void CVector<T>::PopTail()
{
    Delete(m_nCount - 1);
}

template<typename T>
void CVector<T>::Delete(size_t nIdx)
{
    assert(nIdx < m_nCount);
    if (m_pBuff == nullptr)
    {
        return;
    }
    else
    {
        //memcpy(m_pBuff + nIdx, m_pBuff + nIdx + 1, (m_nBuffLen - nIdx) * sizeof(T));
        // m_nBuffLen -= sizeof(T);容量不应该减少
        for (size_t i = nIdx; i < m_nCount - 1; i++)
        {
            m_pBuff[i] = m_pBuff[i + 1];
        }
        m_nCount--;
    }
}

template<typename T>
T& CVector<T>::operator[](size_t nIdx)
{
    assert(nIdx < m_nCount);
    return m_pBuff[nIdx];
}

template<typename T>
int CVector<T>::Find(size_t val) const
{
    for (size_t i = 0; i < m_nCount; i++)
    {
        if (m_pBuff[i] == val)
        {
            return i;
        }
    }
    return -1;
}

template<typename T>
void CVector<T>::Sort()
```

```cpp
{
}

template<typename T>
bool CVector<T>::IsEmpty() const
{
    return m_nCount == 0;
}

template<typename T>
size_t CVector<T>::GetCount() const
{
    return m_nCount;
}

template<typename T>
void CVector<T>::Clear()
{
    if (m_pBuff != nullptr)
    {
        m_nCount == 1 ? delete m_pBuff : delete[] m_pBuff;
    }
    Reset();
}
```

Fence 1-1

1.2 动态数组.cpp

```cpp
// 动态数组.cpp

#include <iostream>
#include "CVector.h"
using namespace std;

class CA
{
public:
    CA() :m_p(nullptr) {}
    CA(int n) :m_p(new int(n)) {}
    CA(const CA& obj) :m_p(new int(*obj.m_p)) {}
    CA& operator=(const CA& obj)
    {
        if (m_p != nullptr)
        {
            delete m_p;
        }
        m_p = new int(*obj.m_p);
        return *this;
    }
    ~CA()
    {
        if (m_p != nullptr)
        {
            delete m_p;
        }
    }
private:
    int* m_p;
};

int main()
{
    // 动态数组,元素是CA类型的对象
    // 先创建3个对象的一个C++自带的动态数组,
    // 再迭代器这个动态数组，每个对象尾插到自己的动态数组vet

    CVector<CA> vet({ CA(4), CA(6), CA{8} });

    vet.PopHead();


    CVector<int> vec;
    vec.Insert(0, 4);
    vec.Insert(1, 5);
    // vec.Insert(5, 7);
    vec.Insert(2, 9);
    vec.Insert(1, 2);
    vec.Insert(3, 5);

    vec.PopHead();
    vec.Delete(2);

    vec[1] = 7;
    int nRes = vec.Find(7);

    return 0;
}
```

Fence 1-2

2. STL中的Vector、迭代器

```cpp
// STL中的Vector.cpp

#include <iostream>
#include <vector>
using namespace std;

int main()
{
    vector<int> vct({6, 9, 8, 2, 41, 45, 65});
    vct.push_back(4);
    vct.push_back(2);

    vct[2] = 8;

    vct.front();
    vct.back();

    vct.pop_back();
    vct.pop_back();

    for (auto val : vct)
    {
        cout << val << endl;
    }

    // 迭代器 == 位置
    vector<int>::iterator itr = vct.begin();
    cout << *itr << endl;
    (*itr) = 89;
    itr++;
    itr++;
    *itr = 88;

    auto itr0 = vct.end();// 指向的最后一个元素的后面一个地方
    --itr0;
    --itr0;
    *itr0 = 33;
    for (auto itr1 = vct.begin(); itr1 != vct.end(); ++itr1)
    {
        cout << *itr1 << endl;
    }
    //                             最后一个                第一个
    for (auto itr1 = vct.rbegin(); itr1 != vct.rend(); ++itr1)
    {
        cout << *itr1 << endl;
    }

    initializer_list<int> il = { 3, 4, 5, 77, 33, 23, 54 };
    for (initializer_list<int>::iterator itr = il.begin(); itr != il.end(); itr++)
    {
        cout << *itr << endl;
    }

    return 0;
}
```

Fence 2-1