Методические указания к лабораторной работе № 7 по курсу Программирование на основе классов и шаблонов

"Контейнерные классы. Тип массивы "

Москва, МГТУ

СОДЕРЖАНИЕ

СОДЕРЖАНИЕ 2
1. Цель работы 4
2. Задачи, решаемые в лабораторной работе 4
3. Основные понятия и примеры 5
3.1. Контейнеры 5
3.2. Контейнерные и элементные классы 5
3.3. Разновидности контейнеров 6
3.4. Контейнеры - массивы 7
3.5. Операции с контейнерами 7
3.6. Итераторы 8
3.7. Операции с контейнерами типа массив 8
3.8. Контейнерные классы в VS 8
3.9. Контейнерные классы массивов в STL (vector) 9
3.10. Контейнерные классы массивов в MFC 12
3.11. Контейнерные классы массивов в ATL 14
3.12. Наследование для массивов контейнерных классов ДЗ/КЛР 15
3.13. Контейнеры в BC (дополнительные требования) 16
3.14. Сравнение списков и массивов 16
3.15. Отладка программ с контейнерами типа массив 17
4. Порядок работы и методические указания (основные требования) 17
4.1. Создать и русифицировать консольный проект в VS 2005-12 (LAB7) 17
4.2. Варианты для выполнения заданий ЛР 18
4.3. Изучить раздел данного документа “Основные понятия” 18
4.4. Выполнить описание вектора vector типа int/long 18
4.5. Выполнить заполнение массива - вектор 18
4.6. Создать свою функцию распечатки векторов типа int/long 18
4.7. Использование итератора для векторов 18
4.8. Создать функцию распечатки векторов типа int/long с итератором 18
4.9. Выполнить добавление элементов в массив по номеру 18
4.10. Выполнить удаление элементов из массива по номеру 19
4.11. Выполнить очистку массива и проверку пустого массива 19
4.12. Выполнить описание вектора для своего класса 19
4.13. Выполнить методы класса vector для собственного класса 19
4.14. Создать функцию распечатки массивов своего класса с итератором 19
4.15. Выполнить описание класса MFC по варианту для своего класса 19
4.16. Выполнить заполнение массива для своего класса 19
4.17. Выполнить манипуляции в массиве 19
4.18. Создание массива для элементного класса ДЗ/КЛР 20
4.19. Создание массива/списка для контейнерного класса ДЗ/КЛР 20
4.20. Оформить отчет по ЛР 20
4.21. Дополнительные требования для сильных студентов 20
5. Варианты по группам и студентам 20
6. Диаграммы классов (дополнительные требования) 21
7. Ошибки и их запоминание (требования) 21
8. Контрольные вопросы 21
9. Оформление отчета (требования) 22
10. Сроки и порядок защиты ЛР 22
11. Литература 23
12. Справочные материалы 23
12.1. Методы vector 23

12.2. Методы CArray 25
12.3. Методы CObArray 26
12.4. Методы CAtlArray 27

1. Цель работы

Целью лабораторной работы №7 изучение контейнерных классов типа массив. Изучаются понятия контейнерных классов. Изучаются методы и особенности классов типа массив. Выполняются практические работы по применению классов стандартных библиотек VS: STL, MFC и ATL.

2. Задачи, решаемые в лабораторной работе

В процессе выполнения ЛР студенты индивидуально должны выполнить следующие задачи. Перечислим основные требования к лабораторной работе:

- Прочитать и усвоить раздел данных МУ – " **3. Основные понятия**" (Это желательно сделать до начала ЛР).
- Выполнить все задания из раздела "**4. Порядок выполнения работы**":
  - Создать в VS 2005 консольный проект (**LAB7**).
  - Обеспечить русификацию ввода и вывода с консоли.
  - Изучение теоретической части ЛР (Раздел - "Основные положения").
  - Работа с классом массив из STL – vector, для наполнения типа int.
  - Работа с классом массив из STL – vector, для наполнения типа своего простого класса (см. таблицу вариантов группы).
  - Работа с классом массив из MFC/ATL по варианту, для наполнения типа своего простого класса (см. таблицу вариантов группы – в **данном** документе).
  - Работа с классом массив из MFC/ATL по варианту ДЗ/ (см. список вариантов ДЗ/КЛР на сайте).
  - Оформление отчета по ЛР.
  - Все действия по программированию выполняются в интерактивном режиме с использованием отладчика.
  - После выполнения перечисленных действий студенты: демонстрируют работу программы, оформляют на основе шаблона отчет по лабораторной работе и защищают ее, отвечая на контрольные вопросы представленные в данных методических указаниях.

Примечания. 1. Для удобства восприятия текста используется цвет и тип шрифта. Фрагменты исходного текста, включаемые в программы, я буду выделять цветом. Например, вставка заголовочного файла в программу будет выглядеть так:

```
#include <iostream>
```

2. Вывод результата в консольное окно, который формируется программой, будем помечать коричневым цветом и устанавливать непропорциональный шрифт **Courier New**. Например:

```
Введите iVal: 10
```

3. Формализованные описания языка и синтаксические правила будем записывать зеленым цветом:

<левая часть выражения присваивания> = <правая часть выражения присваивания>;

4. Если в тексте встречается переменная, которая подчеркнута, то это означает, что дается определение важного понятия и это понятие встречается в данном тексте первый раз. Например:

Программа – это упорядоченная совокупность операторов …

Примечание 2. По каждой ЛР вам даются методические указания (данный документ) и шаблон оформления отчета. Эти материалы вы можете получить на сайте – www.sergebolshakov.ru в разделе “Лабораторные работы”. Пароль для доступа сообщу на лекции или во время ЛР. Кроме того, на сайте вы найдете общие методические указания к ЛР и ДЗ по курсу (“**Общие методические указания по дисциплине ПКШ**”). Для выполнения ЛР нужно познакомиться в первую очередь с разделами 4 и 8 (“**Технология создания исполнимых программ**” и “**Разработка блок схем**”). С этим материалом желательно познакомиться до начала ЛР. Изучаемые материалы с сайта считаются актуальными, если в колонтитуле документа стоит год соответствующий текущему семестру (например, 2012-2013 уч. год).

Примечание 3. После выполнения ЛР необходимо четко отвечать на все контрольные вопросы, которые приведены в данных МУ. Эти вопросы задаются преподавателем при защите ЛР, включаются в перечень вопросов рейтингов и экзаменационных билетов.

3. Основные понятия и примеры

Практически в любой программе, программной системе выполняется обработка множества объектов (переменных). Для совместного хранения этих объектов используется специальный тип объектов – контейнеры. Контейнеры предназначены для хранения, динамического включения, доступа, распечатки, сортировки и удаления объектов. Простейшим видом контейнеров является массив, в котором основной вид доступа к объектам выполняется по номеру (индексу). Данная ЛР посвящена изучению возможностей контейнеров типа массив и методов работы с этими разновидностями объектов. Первоначально познакомимся с общими свойствами и разновидностями контейнеров.

3.1. Контейнеры

Контейнером называется специальная разновидность объектов, которые обеспечивают хранение других объектов, которые мы будем называть элементными объектами, прямой доступ к этим объектам, а также их совместную обработку (например, печать, изменение и т.д.). Контейнеры-объекты создаются на основе контейнерных классов, которые имеют специальную структуру, специальные свойства и специальные операции. Примерами контейнеров (хранилищ других объектов) в программах могут быть, например:

- окно интерфейса (Window), в которое включены различные управляющие элементы интерфейса (поля ввода, кнопки, заголовки полей и т.д.);
- таблицы базы данных (БД), в которых хранятся описания отдельных полей БД;
- сама таблица записей БД, содержащая множество отдельных записей;
- объекты улицы с домами, расположенными на них;
- стеллажи с книгами в библиотеке;
- И многие другие примеры.

Расширенный перечень примеров контейнерных классов вы можете найти в списке вариантов домашнего задания (ДЗ/КЛР) по дисциплине. Этот список размещен на сайте.

3.2. Контейнерные и элементные классы

При создании контейнерного класса предполагается, что в программе будут использоваться объекты, которые будут включаться в контейнерные объекты. Такие объекты мы будем называть элементными объектами, а классы, которые создаются для их описания –

элементными классами. В зависимости от разновидностей контейнеров, элементных классов может быть много или один. Никаких ограничений на структуру элементных классов не накладывается. В частности некоторые контейнерные классы могут выступать в одном контексте для накопления как контейнерные классы (улицы с домами), а в другом контексте как элементные объекты (улицы, включаемые в город). Для хранения элементных объектов контейнеры они должны иметь специальную организацию, свойства и специфичную модель поведения (специальные методы).

В каждом варианте домашнего задания по дисциплине, студенты создают собственные контейнерные классы и собственные элементные классы (как минимум один контейнерный класс и один элементный класс). Для создания контейнерных классов можно использовать механизмы и способы: (1) наследование от стандартных классов системы программирования; (2) полная разработка контейнерных классов, (3) применение шаблонов классов и др.

3.3. Разновидности контейнеров

Контейнеры могут иметь разные свойства, особенности и разную реализацию. Контейнерные объекты подразделяются: по размерности, по типу доступа, по условиям упорядоченности, по универсальности и по типу или механизму создания класса.

По размерности контейнеры подразделяются на следующие типы:
- С предопределенной размерностью (фиксированной) – например, стандартные массивы;
- С динамической размерностью – например, списки;
- С изменяемой размерностью – например, массивы с наращиваемой размерностью.

По типу доступа контейнеры подразделяются на следующие типы:
- С прямым доступом по индексу (номеру) – например, массивы;
- С прямым доступом по параметру (строка) – например, ассоциативные массивы;
- С последовательным доступом к элементам– например, списки;
- Со случайным доступом к элементам – например, множества.

По упорядоченности контейнеры подразделяются на следующие типы:
- Упорядоченные контейнеры – например, массивы и списки;
- Неупорядоченные контейнеры – например, множества и мультимножества;

По универсальности контейнеры подразделяются на следующие типы:
- Контейнеры, содержащие однородные объекты (объекты одного типа) – например, шаблонные контейнеры;
- Контейнеры, содержащие разнородные объекты (объекты разных типов) – например, объектные контейнеры (в этом случае предусматриваются специальные механизмы для доступа к разнотипным объектам – наследование от базового элемента);

По типу создания контейнеры подразделяются на следующие типы:
- Шаблонные контейнеры, для создания которых достаточно типизировать используемый элементный класс;
- Объектные контейнеры, для создания которых используется базовый объектный класс, от которого наследуются рабочие элементные классы (см. универсальность контейнеров).

В некоторых контейнерных классах одновременно реализуются одновременно разные особенности рассмотренной классификации: есть контейнеры, обеспечивающие одновременно прямой доступ и последовательный доступ; контейнеры с фиксированной и наращиваемой размерностью и т.д..

Правильный выбор типа и свойств контейнеров необходим для грамотного решения поставленной задачи (программирования), включающей необходимость использования контейнеров. Ниже мы отдельно будем рассматривать особенности основных видов контейнеров: массивов (ЛР№7) и списков (ЛР№8).

ПКШ ЛР 7 (ООП)1-й курс

3.4. Контейнеры - массивы

Простейшим видом контейнерного класса является массив. В базовый язык C/C++ включен стандартный вид массивов. Для этого в скобках описания указывается его размерность (максимально допустимое число элементов для хранения в этом контейнере). Такие массивы вы изучали в основном курсе программирования. Примеры описания стандартных массивов (фиксированной размерности до начала компиляции), показаны ниже:

```
// Вспомогательный класс My_Obj
class My_Obj {
public:
	int Num;
	My_Obj operator =(My_Obj ob) { Num = ob.Num; return  *this; }; //для присваивания
};
// Стандартные массивы – примеры использования
int iMas[10]; // Массив с элементами одного типа int  с номерами от 0 до 9-ти.
#define MAX 15  // Переменная этапа компиляции для задания размерности
void main() {		//…
	char cMas[MAX]; //Массив с элементами одного типа char  с номерами от 0 до 14-ти.
			//…
	const int Razm = 5; // задание константной переменной для задания ращмерности
	My_Obj  oMas[Razm]; //Массив с элементами одного типа My_Obj с номерами от 0 до 4-ти.
	My_Obj  * poMas[Razm]; //Массив указателей на объекты типа My_Obj  с номерами от 0 до 4
				//…
	int iRaz = 10; // Простая переменная для задания динамической размерности
	float * pfMas = new float[iRaz];//Динамический массив типа float  с номерами от 0 до 9-ти
	// Прямой Доступ к элементам этих массивов выполняется с помощью индексных выражений
	iMas[5] = 5;
	int k = 1;
	My_Obj *pO = new My_Obj; // Объект создается по указатедю
	oMas[1] = *pO; // Значение объекта в oMas[1]
	oMas[0] = *new My_Obj; // Вычисление значения элемента массива (адреса объекта)  с
	cMas[k + 2] = 'A'; // Индексное выражение - [k + 2]
	pfMas[2] = 5.5f; // Вычисление значения 3-го элемента динамического массива номером 0
			//…
	delete[] pfMas;  // Освобождение памяти
	delete pO;
	//
	}
```

Индексные выражения стандартных типов задаются в квадратных скобках ([]), причем задаются для каждого индекса, если массив имеет несколько измерений (таблица, матрица и т.д.).

Первые четыре массива (см. выше) являются статическими (**iMas**, **cMas**, **oMas, poMas**). Размерность этих массивов определяется до начала компиляции. Изменить размерность этих массивов при выполнении программы невозможно. Четвертый массив, объявленный через указатель (**pfMas**), является динамическим, но с фиксированной размерностью при выделении памяти: при выполнении программы увеличить или изменить размерность можно, но достаточно сложно (Сохранение, освобождение, перераспределение и восстановление данных). Недостатками таких массивов как контейнеров является следующие особенности: заранее определенный тип однородных элементных объектов; сложность добавления новых элементов массивов в середину массива; невозможность доступа к такому массиву по имени массива и вычисление индекса по ключу (см. - ассоциативные массивы).

Для устранения перечисленных недостатков в современные системы программирования включены системы классов для работы с массивами, как с контейнерными объектами. Они определены в STL (класс - **vector**), ATL (класс - **CAtlArray**), MFC(классы - **CArray**, **CObArray**) и .NET (**Array**). Эти классы имеют различные методы и обеспечивают много удобных возможностей для описания массивов и работы с такими объектами типа массив. Однако, каждый из видов имеет и свои недостатки.

3.5. Операции с контейнерами

При работе с контейнерными объектами в целом можно выделить следующие основные операции или действия:

- Создание (описание) контейнеров, с указанием размерности контейнера и его заполнение контейнера;
- Добавление элементных объектов в контейнер;
- Удаление элементных объектов из контейнера;
- Доступ к отдельным элементным объектам контейнера;
- Очистка контейнера от элементных объектов;
- Распечатка содержимого контейнера (всех элементных объектов);
- Расширение размеров контейнера;
- Получение числа элементных объектов в контейнере;
- Обмен местоположения двух разных элементных объектов в контейнере.
- Сравнение контейнеров в целом (операции отношения контейнеров).

В большинстве видов стандартных классах контейнеров предусмотрены эти основные действия и, кроме этого, многие другие. По названию методы разных классов, выполняющие одинаковые действия, могут не совпадать.

3.6. Итераторы

Для удобства и универсализации работы с разными контейнерами используется специальная разновидность объектов для навигации (перемещению) по контейнеру. Они называются итераторами. Итераторы создаются для объектов контейнеров и позволяют обеспечить последовательный доступ (в прямом и обратном направлении) к элементным объектам контейнера. Доступ может выполняться как в прямом (**iterator**), так и в обратном порядке (**reverse_iterator**). Итераторы обеспечивают выполнения следующих основных операций: последовательного движения по элементам контейнера (++, --), проверки достижения конца контейнера и доступа к конкретному элементу контейнера. Использование итераторов характерно для контейнеров библиотеки STL и в большей степени актуально для контейнеров - списков.

3.7. Операции с контейнерами типа массив

При работе с контейнерными объектами типа массив можно выделить следующие основные операции (в скобках показанные названия методов):

- Создание (описание) контейнеров, с указанием размерности контейнера и его заполнения (конструкторы: **vecror, CAtlArray, CArray, CObArray**);
- Добавление объектов в массив (например, методы: **push_back , add, InsertAt**);
- Удаление объектов из массива (например, методы: **pop_back , erase,RemoveAt**);
- Доступ к объектам массива (например, методы и операции**: [], at, GetAt**);
- Очистка контейнера-массива (например, методы: **clear, Removeall**);
- Расширение размеров контейнера (например, методы: **resize, SetAtGrow**);
- Получение числа элементных объектов в контейнере (например, методы: **size, GetSize, GetCount**);
- Обмен местоположения объектов в контейнере, методы (например: **swap**);
- Операции в целом над массивами (например, методы: **copy**, **append**, **assign**);
- Сравнение контейнеров (операции отношения контейнеров).

В данной лабораторной работе необходимо изучить и освоить основные действия над контейнерами. Это необходимо для того, чтобы в ДЗ/КЛР их использовать и не перепрограммировать заново (не “изобретать велосипед”).

3.8. Контейнерные классы в VS

В системе программирования VS предусмотрено значительное разнообразие контейнерных классов для разных применений. Они содержатся в разных библиотеках:

- Библиотеке STL (**vector, list, map, stack, set, multiset, map, multimap и queue**);
- Библиотеке MFC (**CArray, CList, CMap, CObArray, CObList, CMapPtrToPtr** и др);
- Библиотеке ATL (**CAtlArray, CAtlList, CAtlMap** и др.);
- Библиотеке платформы .NET (**Array, List** и др);

Рассмотрим работу с некоторыми из этих контейнеров и используемых для них методов.

3.9. Контейнерные классы массивов в STL (vector)

В библиотеке STL (напомним что это – библиотека шаблонных классов - **Standard Template Library**) описывается шаблонный класс **vector**, который позволяет описывать массивы переменных любого типа. Формальное определение шаблона дано ниже:

template < class Type, class Allocator = allocator<Type> > class vector {…};

Для работы объектами этого класса необходимо подключить заголовочный файл:

```
#include <vector>
```

Для описания вектора (массива) целых переменных нужно указать тип:

```
vector <int> v1; // Вектор/массив - переменных целого типа
v1.push_back( 1 ); // Добавить в конец
v1.push_back( 2 ); // Добавить в конец
v1.insert( v1.begin( ), 55 );// Добавить в начало
```

Для распечатки массива можно использовать следующий фрагмент программы (учтем, что для вектора уже перегружены операции индексирования и потоковые операции вывода целого типа):

```
for(int i = 0; (unsigned) i < v1.size( ) ; i++)
  { cout << "v1["<< i <<"] = " << v1[i] << endl;}; // Печать целого значения
```

В результате работы данного фрагмента, совместно с предыдущим, получим:

```
v1[0] = 55
v1[1] = 1
v1[2] = 2
```

Объявим функцию печати вектора-массива для упрощения дальнейшего текста:

```
// Функция печати вектора
void vPrint(vector<int>& v)
{
        int i;
        for (i = 0; (unsigned)i < v.size(); i++)
               cout << "v[" << i << "] = " << v[i] << endl;
}
```

Тогда печать вектора зададим так:

```
vPrint(v1);
```

В результате работы данного фрагмента, совместно с предыдущим, получим:

```
v1[0] = 55
v1[1] = 1
v1[2] = 2
```

Перечень всех методов класса вектор приведен в разделе Справочные материалы (см. Последний раздел данных МУ), подробное описание методов вы можете найти в документации, литературе и справочной системе MSDN[5]. Рассмотрим еще несколько примеров использования объектов вектор. Для очистки массива может быть использован метод удаления элементов из вектора-массива (`erase`):

```
v1.erase( v1.begin( ), v1. begin ( ) + 2 ); // Очистить 2 первых элемента
//
cout << "Печать функцией (2 первых удалено): " << endl;
vPrint(v1);
cout << "Печать функцией (пустой список): " << endl;
v1.erase( v1.begin( ), v1.end( ) ); // Очистить весь массив
vPrint(v1);
```

В результате работы данного фрагмента, совместно с предыдущим, получим:

```
Печать функцией (2 первых удалено):
```

```
v[0] = 2
Печать функцией (пустой список):
```

Снова заполним вектор - массив **v1**:

```
v1.push_back( 22 ); // Добавить в конец
v1.push_back( 33 ); // Добавить в конец
v1.insert( v1.begin( ), 11);// Добавить в начало
vPrint(v1);
```

Распечатаем содержимое **v1**:

```
v1[0] = 11
v1[1] = 22
v1[2] = 33
```

Получить текущее и предельное число элементов в массиве можно так

```
cout << "Размер массива v1 = "<<  v1.capacity( ) << endl;
cout << "Максимальный размер массива v1 = "<<  v1.max_size( ) << endl;
```

Проверить является ли массив пустым можно специальным методом:

```
if ( v1.empty( ) )
    cout << "Массив пуст." << endl;
 else
    cout << "Массив не пуст." << endl;
```

Получим для **v1** для предыдущего фрагмента:

```
Размер массива v1 = 3
Максимальный размер массива v1 = 1073741823
Массив не пуст.
```

Поменять содержимое двух массивов местами и проверку первого (v2 – первоначально пустой, а в v1 три элемента):

```
vector <int> v2; // Вектор/массив - переменных целого типа
v1.swap( v2 );
vPrint(v2);
if ( v1.empty( ) )
    cout << "Массив пуст." << endl;
 else
    cout << "Массив не пуст." << endl;
```

Получим для **v2** и **v1**:

```
v[0] = 11
v[1] = 22
v[2] = 33
Массив пуст.
```

Рассмотрим использование итераторов для массива типа вектор. Ля корректной работы с итератором, его нужно предварительно описать, а затем использовать в цикле для последовательной распечатки элементов массива. Для навигации по массиву можно применить перегруженную для итератора операцию - “++”. Например:

```
//Итераторы массивов
vector<int>::iterator iter; // Объявление итератора
// Для массива v2
cout << "Печать с помощью итератора:" << endl;
for( iter = v2.begin(); iter != v2.end()  ; iter++)
{ cout << "v2[] = " << *iter << endl;};
```

Итератор по сути является указателем на объект, поэтому для доступа можно воспользоваться выражением разименования (***iter**). Так как в векторе-массиве записаны целые переменные (**int**) печать возможна (перегрузка для **int** стандартно выполнена). Получим на экране после выполнения данного фрагмента:

```
Печать с помощью итератора:
v2[] = 11
v2[] = 22
v2[] = 33
```

Методы **begin** и **end** используются для управления циклом на основе итератора. Первый метод (**begin**) задает первый элемент вектора, а второй (**end**) указывает на последний элемент массива. Сравнение с последним элементом (**iter != v2.end()**) позволяет проверить условие

окончания цикла. Можно также объявить реверсивный итератор и просматривать в этом случае массив с конца до начала, тогда фрагмент программы может выглядеть так:

```
int i ;
v1.swap( v2 );  // восстановим v1
vector <int>::reverse_iterator riter;
//...
cout << "Печать с помощью reverse итератора:." << endl;

for( i= 0, riter = v1.rbegin(); riter != v1.rend()  ; riter++, i++)
{
cout << "v1["<<i <<"] = " << *riter << endl;};
```

Получим на экране после выполнения данного фрагмента:

```
Печать с помощью reverse итератора:.
v1[0] = 33
v1[1] = 22
v1[2] = 11
```

В рамках ЛР необходимо продемонстрировать использование методов класса вектор для объектов целого типа и собственного класса по групповому варианту ЛР№7. Использование собственного класса в качестве содержимого векторного массива имеет особенности. На примере простого класса **Point** покажем их. Пусть создан простой класс:

```
//Класс Point
class Point {
public:
	int x;
	int y;
	Point() { x = 0; y = 0; };                // Конструктор 1
	Point(int a, int b) { x = a; y = b; };   // Конструктор 2
	void Print()                              // Метод печати точки
	{
		cout << "{ x = " << x << " y = " << y << " } " << endl;
	};
	friend ostream & operator <<(ostream &  out, Point &  obj);
};
ostream & operator <<(ostream &  out, Point &  obj)
{
	out << "{ x = " << obj.x << " y = " << obj.y << " } ";
	return out;
};
```

Тогда мы можем описать вектор массив точек(**vP1**) для этих объектов и итератор(**PIter**) для этого типа масива:

```
vector<Point> vP1;
vector<Point>::iterator Piter;
```

Опишем объекты типа **Point** и добавим их в конец массива:

```
// Заполнение массива 3-мя точками (P1 , P2) и созданной динамически
Point P1(1, 2);
Point P2(51, 52);
vP1.push_back(P1);
vP1.push_back(P2);
vP1.push_back(*new Point(31, 32)); // Динамические создание
```

Цикл печати построим с помощью итератора и вспомогательной переменной (**i**), позволяющей выводить индекс массива (i):

```
for (int i=0, PIter=vP1.begin(); PIter!=vP1.end(); PIter++, i++)
		cout << "vP[" << i << "] = " << *PIter << endl;
```

В результате выполнения фрагмента программы получим в консольном окне:

```
vP[0] = { x = 1 y = 2 }
vP[1] = { x = 51 y = 52 }
vP[2] = { x = 31 y = 32 }
```

Особенностью использования собственных объектов является необходимость перегрузки операций ввода/вывода в поток. Другой особенностью класса **vector** является автоматическое корректное удаление динамических объектов, созданных в программе и включенных в массив (например, третья точка, созданная динамически). В ЛР необходимо

описать собственный класс объектов и продемонстрировать все возможности массива типа vector.

3.10. Контейнерные классы массивов в MFC

В библиотеке MFC (**Microsoft Foundation Class Library**) предусмотрено два основных класса для работы с массивами: **CArray** и **CObArray**. Кроме этих классов предусмотрены классы для определенного типа включаемых объектов: CWordArray, CByteArray, CPtrArray, CDWordArray, CStringArray, CUintArray. Класс **CArray** является шаблонным, для создания объектов необходимо указать тип, а класс **CObArray** не требует типизации, в него могут включаться любые объекты, наследованные от класса **CObject**. Методы первого и второго классов практически совпадают по названию и назначению. Рассмотрим первоначально класс **CArray**. Для добавления в массив объектов используется метод **Add**. Элементы всегда добавляются в конец массива. Формально массивы типа CArray являются шаблонами:

```
template < class TYPE, class ARG_TYPE = const TYPE& >
class CArray :
  public CObject { …};
```

Пример описания и добавления в массив переменных типа **int**.

```
CArray<int , int> IntMas;
for (i = 0 ; i < 5 ; i++ )
        IntMas.Add(i);
for (i = 0 ; i < IntMas.GetCount() ; i++ )
        cout << IntMas[i] << " "  ;
        cout << endl;
```

Метод **GetCount()** позволяет получить текущий величину заполнения массива, поэтому он может быть использован для проверки завершения цикла. После выполнения получим результат:

```
0 1 2 3 4
```

Метод **GetUpperBound()** позволяет получить последний занятый индекс в массиве:

```
cout << "GetUpperBound = " << IntMas.GetUpperBound() << endl;
```

Получим:

```
4
```

Метод **GetAt()** позволяет получить значение элемента списка по индексу (номеру):

```
// Получить 2-й элемент
i = IntMas.GetAt(2);
cout << "i = "<< i << "  IntMas[2] = " << IntMas[2] << endl;
```

Получим результат:

```
i = 2  IntMas[2] = 2
```

Метод **SetAt()** позволяет заменить элемент в массиве на другой:

```
// Замена 3-го на 10
IntMas.SetAt(3, 10); // Значение элемента номер 3 заменяется на 10
for (i = 0 ; i < IntMas.GetCount() ; i++ )
        cout << IntMas[i] << " "  ;       cout << endl;
```

Получим результат:

```
0 1 2 10 4
```

С помощью метода **InsertAt()** можно добавить в середину:

```
// Вставка после 1-го
IntMas.InsertAt(1, 100);
for (i = 0 ; i < IntMas.GetCount() ; i++ )
        cout << IntMas[i] << " "  ;     cout << endl;
```

Получим результат:

```
0 100 1 2 10 4
```

С помощью метода **RemoveAt()** можно удалить любой элемент по номеру:

```
//Удаление 2-го
IntMas.RemoveAt(2);
```

```
for (i = 0 ; i < IntMas.GetCount() ; i++ )
      cout << IntMas[i] << " "  ;     cout << endl;Получим результат:
  0 100 2 10 4
```

С помощью метода **IsEmpty()** можно проверить наличие в массиве элементов, а с помощью метода **RemoveAll()**, удалить все элементы:

```
// Удаление и проверка пустого
if ( IntMas.IsEmpty() )
{ cout << "Массив пуст!" <<endl; }
else
{ cout << "Массив не пуст!" <<endl; };
IntMas.RemoveAll(); // Удаление всех элементов
if ( IntMas.IsEmpty() )
{ cout << "Массив пуст!" <<endl; }
else
{ cout << "Массив не пуст!" <<endl; };
//
```

Получим результат:

```
Массив не пуст!
Массив пуст!
```

Методы **SetSize** и **FreeExtra** позволяют установить новую размерность массива и удалить свободную память:

```
//Изменение размера массива
IntMas.SetSize(32, 128); // Размер 32
cout << "GetSize = " << IntMas.GetSize() << endl;
cout << "GetUpperBound = " << IntMas.GetUpperBound() << endl;
IntMas.SetSize(10, 128); // Новый Размер 10!!!
IntMas.FreeExtra();
cout << "GetSize = " << IntMas.GetSize() << endl;
cout << "GetUpperBound = " << IntMas.GetUpperBound() << endl;
```

Получим результат:

```
GetSize = 32
GetUpperBound = 31
GetSize = 10
GetUpperBound = 9
```

Специальными методами можно копировать (**Copy()**) или добавлять к существующему массиву (**Append()**) однотипные массивы:

```
// Сложение массивов
CArray<int , int> IntMas1;
IntMas1.Add(55); // Добавим 1 элемент в массив IntMas1
IntMas1.Append( IntMas );   // Добавление массива IntMas с массиву IntMas1
for (i = 0 ; i < IntMas1.GetCount() ; i++ )
      cout << IntMas1[i] << " "  ;     cout << endl;
IntMas.Copy( IntMas1 );       // Копирование массива IntMas1 в массив IntMas
for (i = 0 ; i < IntMas.GetCount() ; i++ )
      cout << IntMas[i] << " "  ;     cout << endl;
```

Получим результат:

```
55 0 100 2 10 4
55 0 100 2 10 4
```

Для класса массивов **CObArray** (из библиотеки классов MFC) все методы аналогичны. Существенное отличие заключается в том, что такой массив поддерживает массив указателей на объекты типа **CObject**, или точнее, объекты классов наследованных от **CObject**. В этом случае их описание эквивалентно такому:

```
CArray< CObject *, CObject*> mArr; // Нужна настройка шаблона на CObject
//Массив класса CObArray
CObArray mArrObj;  // Настройки шаблона не нужно
```

Если класс **MPoint** является наследником класса **CObject**, то все предыдущие фрагменты проверки работы методов для нашего класса объектов работают нориально нормально, например, для класса:

```
class MPoint : public CObject {
public:
```

```
int x;
int y;
MPoint(){ x =0 ; y = 0;};
MPoint(int a , int b ){ x = a ; y = b;};
void Print()
{          cout <<"{ x = " << x <<" y = " << y << " } ";};
friend ostream & operator <<( ostream &  out  ,  MPoint &  obj );
};
ostream & operator <<( ostream &  out  ,  MPoint &  obj )
{
out  <<"{ x = " << obj.x <<" y = " << obj.y << " } " <<endl;
return out;
};
//...
```

Построим фрагмент программы:

```
//Циклы заполнения и печати массива типа CObArray
 CObArray mArrObj;
 for (i = 0 ; i < 5 ; i++ )
        mArrObj.Add(new MPoint(i,i));
 cout << " ";
 for (i = 0 ; i < mArrObj.GetCount() ; i++ )
        cout << *((MPoint *)(mArrObj[i])) << " "  ;
        cout << endl;
```

Получим результат:

```
{ x = 0 y = 0 }
{ x = 1 y = 1 }
{ x = 2 y = 2 }
{ x = 3 y = 3 }
{ x = 4 y = 4 }
```

Единственное отличие состоит в необходимости преобразования указателя к типу **Point** для автоматического вызова перегруженной операции вывода или, как в нашем примереЭ нужно эту операцию заново. Кроме этого, необходимо учесть что добавление, выборка объектов должны выполняться с указанием объектов соответствующих типов, наследованных от класса **CObject**.

3.11. Контейнерные классы массивов в ATL

Контейнерный класс массива в библиотеке ATL (библиотека активных шаблонов - **Active Template Library**) называется **CAtlArray**. Он является шаблонным классом и имеет следующее формальное описание:

```
template<   typename E,   class ETraits = CElementTraits< E >
> class CAtlArray { … };
```

В этом классе все очень похоже на класс **CArray** из MFC, но некоторые методы отсутствуют. В частности недоступны методы: **GetUpperBound**, **GetSize** и **SetSize**. Все остальные методы и свойства совпадают с массивами библиотек **MFC**.

При подключении классов библиотеки ATL в главный модуль нужно добавить следующий заголовочный файл:

```
#include <atlcoll.h>
```

Для примера использования класса **CAtlArray** рассмотрим фрагмент программы с массивом целых чисел:

```
CAtlArray<int> atlMas;
//Циклы заполнения и печати массива типа CAtlArray
for (i = 0 ; i < 5 ; i++ )
      atlMas.Add(i);
for (i = 0 ; i < atlMas.GetCount() ; i++ )
      cout << atlMas[i] << " "  ; //Печать в строку
```

```
    cout << endl;
//...
```

После выполнения этого фрагмента программы получим результат:

```
0 1 2 3 4
```

Если в предыдущих фрагментах программ закомментировать строки с недоступными функциями и переопределить название массивов (например, на **atlMas**), они будут работать.

3.12. Наследование для массивов контейнерных классов ДЗ/КЛР

Ниже приводиться упрощенное описание класса **Home** для примера из методических указаний по ДЗ (дома **Home** и улицы - **Street**)[14]. Отметим, что в ДЗ/КЛР необходимо предусмотреть прямое наследование от класса **CObject.** Ниже дан пример упрощенного описания элементного класса ДЗ. На примере простого этого класса **Home** покажем простой пример его использования. Изученные классы библиотек для списков необходимо использовать в домашнем задании (ДЗ) по дисциплине и комплексной ЛР (ЛР№12-15). Для контейнерного класса улиц - **Street**, ниже также дано максимально простое описание, для иллюстрации наследования от стандартных классов массивов.

Использование собственного класса в качестве содержимого массива ряд особенностей. Пусть создан простой класс **Home** (наследование от класса **CObject** нужно для других примеров):

```
//////////////
//Класс Домов (сильно упрощенный класс имя и координаты) =>  Home
class Home : public CObject{
public:
   int x;
   int y;
string Name;
Home(const Home & p){ x = p.x ; y = p.y ; Name = p.Name ;};
Home & operator=(const Home & p){ x = p.x ; y = p.y ; Name = p.Name ; return *this;};
Home(string S = ""){ Name = S; x= 0; y = 0; };
   Home(int a , int b , string S = "") {
   x =a; y = b;
   Name = S;};
friend ostream & operator <<( ostream &  out  ,  Home &  obj );
};
// Перегрузка операции для вывода объекта Типа Home
ostream & operator <<( ostream &  out  ,  Home &  obj )
{
// out  <<"{ x = " << obj.x <<" y = " << obj.y << " Name = " << obj.Name <<" } " <<endl;
out  <<"{ x = " << obj.x <<" y = " << obj.y << "  Имя = "<< obj.Name <<" } " <<endl;
return out;
};
//
```

Контейнерный класс ДЗ/КЛР – улицы - **Street** должен быть наследован от одного из библиотечных классов (списки и массивы – см. варианты задания). В нашем примере это класс **CAtlArray**. Его описание необходимо подключить в заголовочный файл проекта. Не забудьте подключить библиотеку (**atlcoll.h** – для этого класса):

```
// Класс улиц - наследник списка CAtlList
class Street: public CAtlArray <Home>
{
public:
string Name;
// другие свойства класса улиц
Street(string S = "" ){ Name = S; };
// другие методы класса улиц
};
```

На основе описаний объектов можно проверить простой текст программы для включения в список (улица) домов и распечатки содержимого списка:

```
// Описания объектов
Street S ("Моя улица - Массив!");
Home H11("Первый дом!");
Home H21(11 , 2 , "Второй дом!");
Home H31(21 , 2 , "Третий дом!");
S.Add (H11);
S.Add(H21);
S.Add (H31);
cout << endl << "Название улицы  = " << S.Name << endl;
cout << "Список домов улицы: " << endl;
for (i = 0 ; i < (int)S.GetCount() ; i++ )
      cout << S[i]  ;
      cout << endl;
 cout << endl;
```

После выполнения программы должны получить:

```
Название улицы  = Моя улица - Массив!
Список домов улицы:
{ x = 0 y = 0  Имя = Первый дом! }
{ x = 11 y = 2  Имя = Второй дом! }
{ x = 21 y = 2  Имя = Третий дом! }
```

3.13. Контейнеры в BC (дополнительные требования)

Для продвинутых студентов в комплект лабораторной работы включены исходные тексты классов **Array** и **List** из СП BC++ 3.1. Эти классы специально преобразованы для возможности работы в VS. Так как присутствуют все исходные описания классов и их методов, то можно изучить досконально организацию контейнеров этих типов на практике. Целесообразно вручную построить диаграммы классов и сформировать библиотеку объектных модулей, которую можно подключать в разные проекты.

3.14. Сравнение списков и массивов

В 7-й и 8-й ЛР вы изучаете контейнерные классы для списков и для массивов. В ДЗ вы должны научиться их использовать для построения собственной системы классов по вариантам задания. В вариантах предписано использование классов: **CArray**, **CList**, **CObArray** или **CObList**. Несомненно, использование типа зависит от содержания задания. Варианты используемых классов назначены для каждой из групп студентов. Считаю, что возможности классов из данных библиотек должны подойти для всех ваших заданий. В действительности для каждой задачи использование массивов и списков или имеют следующие существенные особенности:

- Массивы целесообразно использовать, если необходим прямой доступ к элементу контейнера по индексу.
- Списки предпочтительнее использовать, если содержимое, в первую очередь, последовательность элементов часто изменяется (сортировка).
- Массивы целесообразно использовать для больших объемах данных, заносимых в контейнер.
- Списки предпочтительнее использовать, если предполагается складывать или разделять контейнеры.
- Массивы целесообразно использовать для множеств упорядоченных по номеру.

В каждом конкретном случае необходимо оценить приемлемый тип контейнеров, который целесообразно использовать. Нельзя также забывать, что в библиотеках есть и другие

типы контейнерных классов: очереди, стеки, множества, ассоциативные массивы, мультимножества и т.д. Эти классы предлагаются для изучение в планах самостоятельной проработки материала.

3.15. Отладка программ с контейнерами типа массив

В данной лабораторной работе отладчик имеет особое значение. С его помощью мы можем проверить вызов типов перегруженных и задание параметров по умолчанию

При разработке программ важную роль играет отладчик, который встроен в систему программирования. В режиме отладки можно проверить работоспособность программы и выполнить поиск ошибок самого разного характера. Отладчик позволяет проследить ход (по шагам) выполнения программы и одновременно получить текущие значения всех переменных и объектов программы, что позволяет установить моменты времени (и операторы), в которые происходит ошибка и предпринять меры ее устранения. В целом, отладчик позволяет выполнить следующие действия:

- Запустить программу в режиме отладки без трассировки по шагам (**F5**);
- Выполнить программу по шагам (**F10**);
- Выполнить программу по шагам с обращениями к вложенным функциям(**F11**);
- Установить точку останова (**BreakPoint** – **F9**);
- Выполнить программу до первой точки останова - breakpoint (**F5**);
- Просмотреть любые данные в режиме отладки в специальном окне (**locals**);
- Просмотреть любые данные в режиме отладки при помощи наведения на них мышки;
- Изменить любые данные в режиме отладки в специальном окне (**locals** и **Watch**);
- Установить просмотр переменных в специальном окне (**Watch**);
- Остановить и выключить отладку при необходимости;
- Просмотреть последовательность и вложенность вызова функций.
- При выполнении всех лабораторных курса студенты должны знать и активно использовать отладчик VS любых версий системы программирования, знать его возможности и отвечать на контрольные вопросы, связанные с отладкой и тестированием программ.

При выполнении всех лабораторных курса студенты должны продемонстрировать использование отладчика VS, знать все его возможности, уметь воспользоваться технологиями отладки и отвечать на все контрольные вопросы, связанные с отладкой и тестированием программ (см. раздел контрольных вопросов).

Примечание. Подробное описание материала и понятий вы можете найти в литературе [1 - 6] или справочной системе MS VS[2]. Кроме того, не пропускайте лекции по курсу. Не рекомендую безоговорочно верить материалам из сети Интернет (например, в Википедии), так как там в некоторых статьях есть ошибки!

4. Порядок работы и методические указания (основные требования)

4.1.Создать и русифицировать консольный проект в VS 2005-12 (LAB7).

ПРОЕКТ VS. Последовательность действий для создания консольного проекта в VS описана в методических указаниях к ЛР № 1 для данного курса (ПКШ), смотрите на сайте дисциплины. Русификация ввода и вывода информации в проекте для консольного режима выполняется на основе приемов, также показанных в методических указаниях к ЛР №1.

4.2. Варианты для выполнения заданий ЛР

До начала выполнения ЛР необходимо познакомиться с таблицей вариантов для групп студентов (см. ниже) и выполнять задание следующих пунктов в соответствии со своим вариантом.

4.3. Изучить раздел данного документа “Основные понятия”

ПОДГОТОВКА. Необходимо внимательно прочитать раздел МУ “Основные понятия” и на практике в отдельном проекте проверить выполнение предложенных фрагментов текста. Желательно в отладчике в пошаговом режиме просмотреть изменяющиеся значения переменных и объектов программы.

4.4. Выполнить описание вектора vector типа int/long

ОПИСАНИЕ ВЕНКТОРА. Выполнить описание массива типа **vector** для типа **int** или **long** по варианту группы. Присвоить имени массива уникальное и понятное имя. Имя массива должно хорошо запоминаться и быть осмысленным.

4.5. Выполнить заполнение массива - вектор

ЗАПОЛНЕНИЕ МАССИВА. Выполнить заполнение массива **vector** целыми переменными не менее 5-ти элементов. Метод для занесения **push_back**. Распечатать результат в цикле с операцией индексирования ([]).

4.6. Создать свою функцию распечатки векторов типа int/long

ФУНКЦИЯ ПЕЧАТИ МАССИВА. Создать в заголовочном файле проекта универсальную функцию распечатки векторов типа **int/long**. Прототип может быть таким:

```
void vPrint(vector<int>& v);
```

Распечатать вектор с помощью новой функции.

4.7. Использование итератора для векторов

ИТЕРАТОР. Распечатать свой массив - вектор с помощью итератора (**Iter**) и операции инкремента (++) для итератора. Проверить использование операции декремента для итератора. Результаты распечатать в отчет.

4.8. Создать функцию распечатки векторов типа int/long с итератором

ФУНКЦИЯ С ИТЕРАТОРОМ. Создать и проверить функцию распечатки вектора с помощью итератора.

4.9. Выполнить добавление элементов в массив по номеру

ДОБАВЛЕНИЕ В МАССИВ. Для добавления использовать метод **insert**. Номер добавляемого элемента определяется вариантом. Распечатать массив.

4.10. Выполнить удаление элементов из массива по номеру

УДАЛЕНИЕ ИЗ МАССИВА. Для удаления использовать метод **erase**. Номер удаляемого элемента определяется вариантом. Распечатать массив.

4.11. Выполнить очистку массива и проверку пустого массива

ОЧИСТКА МАССИВА. Выполнить проверку пустого массива (метод **empty**), очистить массив, и снова выполнить проверку пустого массива. Результат проверки вывести на экран.

4.12. Выполнить описание вектора для своего класса

ВЕКТОР КЛАССА ЗАДАНИЯ. Описать собственный класс (в заголовочном файле проекта), определенный вариантом группы, и описать массив на основе класса **vector** с указанием типа собственного класса. Предусмотреть перегрузку для класса операции вывода в поток (“<<”). Заполнить массив (не менее 5-ти элементов) и результат распечатать в **cout**.

4.13. Выполнить методы класса vector для собственного класса

РАБОТА С МАССИВАМ. Выполнить с массивом объектов собственного класса **все возможные действия** (см. выше п.4 - п.11): распечатку, добавление, удаление, очистку, использование итератора, проверку пустого массива. Промежуточные результаты распечатать и проверить в отладчике.

4.14. Создать функцию распечатки массивов своего класса с итератором

ФУНКЦИЯ С ИТЕРАТОРОМ. Создать в заголовочном файле проекта функцию распечатки векторов с наполнением собственного класса.

4.15. Выполнить описание класса MFC по варианту для своего класса

ОПИСАНИЕ MFC. По заданному варианту описать массив для хранения объектов собственного элементного класса (см. выше). Для варианта класса **CObArray** этот класс должен наследоваться от класса **CObject**. В новом классе перегрузить операцию вывода “<<”.

4.16. Выполнить заполнение массива для своего класса

ЗАПОЛНЕНИЕ МАССИВА. Заполнить массив объектами своего класса (методом **Add**). Результат заполнения распечатать.

4.17. Выполнить манипуляции в массиве

РАБОТА С МАССИВОМ. Выполнить действия со списком: выборку по номеру (метод **GetAt**), замену по номеру (метод **SetAt** – номер 2), Вставку (**InsertAt**) и удаление элемента с номером по варианту(**RemoveAt**) и очиску всего списка (**RemoveAll**). Проверить пустой список(). Выполнить копирование первого массива во второй (**Copy** и **Append**). Результат распечатать.

4.18. Создание массива для элементного класса ДЗ/КЛР

ЭЛЕМЕНТНЫЙ КЛАСС ДЗ. Создать и наполнить массив (не менее 5-и объектов) объектами **элементного** класса для вашего варианта ДЗ/КЛР, описанными в предыдущих лабораторных работах. Объект элементного класса нужно наследовать от **CObject**. Для описания этого массива использовать классы и шаблоны библиотек MFC, заданные в варианте группы (см. ниже). Распечатать массив в цикле по индексу и корректно очистить их массивы. Результаты поместить в отчет ЛР и в документацию по ДЗ/КЛР.

4.19. Создание массива/списка для контейнерного класса ДЗ/КЛР

КОНТЕЙНЕРНЫЙ КЛАСС ДЗ. Описать **контейнерный** класс типа массив для вашего варианта ДЗ/КЛР. Для описания этого класса использовать требования применения базового класса для вариантов групп по типу библиотек (MFC) соответственно (наследование от стандартного класса). Если в ДЗ у Вас задан массив, то выполняете этот пункт для списка, если записан список, то выполняете этот пункт для массива. Распечатать список/массив с помощью своей перегруженной операции вывода (“<<”). В данный массив включать объекты элементного класса из предыдущего пункта. Распечатать массив в цикле с помощью перегруженной операции вывода для объектов элементного класса. Очистить массив с помощью специального метода класса (**RemoveAll**).

4.20. Оформить отчет по ЛР

ОТЧЕТ. На основе шаблона, приведенного в конце документа оформить отчет по ЛР №7.

4.21. Дополнительные требования для сильных студентов

Для сильных студентов предлагаются дополнительные требования при выполнении ЛР №7. Эти требования могут быть выполнены в любой последовательности и в любом объеме. На титульном листе отчета по ЛР необходимо указать, что дополнительные требования выполнены. Нужно:

- Проверить использование итераторов всех типов для циклических заданий в пунктах ЛР (распечатка массивов – прямая и обратная печать).
- Создать собственный класс типа массив с перегруженными операциями индексирования, присваивания и сложения массивов, в котором предусмотрено множественное наследование.
- Изучить классы BC - тип массив (Array). Они усть в архиве МУ ЛР.
- Построить библиотеку объектных модулей для системы классов BC и подключить ее в проект на VS.
- Построить диаграмму классов для класса массив (Array).

5. Варианты по группам и студентам

Ниже предвтавлена таблица вариантов по группам студентов. В данной ЛР не назначаются индивидуальные варианты для каждого конкретного студента.

ПКШ ЛР 7 (ООП)1-й курс

Вар №	Группа	Тип элементного объекта	Тип контейнерного класса	Удаление/добавление в массив (номер объекта)	Тип для класса вектор
1	ИУ5 – 21	Circle	CObArray (MFC)	2-й / 3-й	long
2	ИУ5 – 22	Triangle	CArray (MFC)	Первый / Последний	int
3	ИУ5 – 23	Complex	CArray (MFC)	Первый / 3-й	int
4	ИУ5 - 24	Line	CObArray (MFC)	Первый / Последний	int
5	СУЦ	Rect	CObArray (MFC)	2-й / Последний	long

6. Диаграммы классов (дополнительные требования)

Построить диаграмму классов для системы классов массив **из ВС**. Исходные тексты модулей в комплекте документов ЛР №7.

7. Ошибки и их запоминание (требования)

Для накопления профессионального опыта в программировании рекомендуется запоминать и фиксировать ошибки, возникающие на различных стадиях разработки программ: разработки алгоритмов, подготовки текста, компиляции и отладки программ. С этой целью вводится требование размещения в отчете по ЛР фиксации ошибок в специальной таблице (см. шаблон отчета по ЛР в конце документа). При этом запоминается: тип и суть ошибки, этап возникновения и способ устранения. Такая работа является очень полезной и позволяет избавиться от ошибок и легче находить способы их устранения.

8. Контрольные вопросы

1. Что такое проект в VS 2005? Для чего нужны проекты и в чем их преимущество?
2. Что такое контейнерный объект?
3. Какие разновидности контейнеров вы знаете?.
4. Какие операции выполняются с контейнерами?
5. Приведите примеры контейнеров.
6. Что такое элементный класс?
7. В каком отношении он находиться с контейнерным классом?
8. Какие признаки классификации контейнеров вы знаете?
9. В чем заключаются недостатки стандартных массивов C++?
10. Что отличает массив от других контейнерных классов?
11. Что такое статический массив?
12. Что такое динамический массив?
13. Какие массивы вы знаете, и в каких библиотеках VS они включены?
14. Что такое итератор и для чего он используется?
15. Какие операции доступны для итераторов?
16. Какие операции с массивами вы знаете?
17. Перечислите основные методы класса vector.
18. Перечислите основные методы класса CArray.
19. Перечислите основные методы класса CObArray.
20. Перечислите основные методы класса CAtlArray.
21. Чем отличаются разные классы массивов из стандартных библиотек?

22. Какие действия можно предпринять во время отладки программы с контейнерами?

9. Оформление отчета (требования)

Студенты должны приходить на отработку лабораторной работы подготовленными. Подготовка включает в себя знакомство с данными методическими указаниями, осмысление поставленных задач. Усвоение основных понятий, связанных с данной темой. Полезным может быть предварительная подготовка отчета в электронном виде для дальнейшего его заполнения в процессе работы. Шаблон отчета размещен в конце документа (ссылка на него) и после доработки может быть использован при защите ЛР.

Отчет по ЛР должен включать следующие разделы:

– Титульный лист с указанием группы и фамилии студента и преподавателя.
– Цель лабораторной работы
– Порядок выполнения основных шагов ЛР (крупно)
– Диаграмма созданных классов.
– Перечень, обнаруженных ошибок (в таблице), возникших при разработке и отладки программы ЛР.
– Исходный текст всех модулей программы, включая и заголовочные файлы.
– Результаты работы программы в текстовом формате, скопированные с консольного окна при ее выполнении.
– Выводы по работе

Для защиты отчет предоставляется в распечатанном виде, кроме того, студент должен иметь все исходные и загрузочные файлы, включая и файлы проекта под VS 2005.

10. Сроки и порядок защиты ЛР

Основная часть работы должна быть выполнена студентом на занятиях в дисплейном зале. В конце занятия студенты должны продемонстрировать работоспособную программу, содержащую все основные задачи ЛР. Преподаватель в журнале проставляет результаты демонстрации тремя способами: демонстрация зачтена (+), демонстрация не зачтена (-), работа не выполнена полностью (±). Если проставлен (-), то работа не засчитывается и подлежит повторной отработке. В случае отметки (±), работа может быть доработана в домашних условиях, считается отработанной и предоставлена к защите по отчету. При защите в этом случае задаются дополнительные вопросы.

В течение 2-х недель, по отработанной лабораторной работе, проводится защита. Преподаватель проверяет правильность оформления отчета, соответствие вариантам групп и студентов и задает контрольные вопросы. При положительных ответах на вопросы ЛР считается сданной и защищенной (в журнале отмечается обведением знака +). При отрицательных ответах на вопросы отчет может быть представлен к повторной защите, но не более 2-х раз. Если работа после 2- кратной защиты не сдана, то она отрабатывается повторно во время специальных занятий в конце семестра.

Отработка невыполненных и пропущенных ЛР возможна только в конце семестра (на специальных занятиях и по специальному графику). Студенты, не отработавшие и не защитившие все ЛР, к экзамену не допускаются.

11. Литература

1. Г. Шилдт “C++ Базовый курс”: Пер. с англ.- М., Издательский дом “Вильямс”, 2011 г. – 672с
2. Г. Шилдт “C++ Руководство для начинающих” : Пер. с англ. - М., Издательский дом “Вильямс”, 2005 г. – 672с
3. Г. Шилдт “Полный справочник по C++”: Пер. с англ.- М., Издательский дом “Вильямс”, 2006 г. – 800с
4. Бьерн Страуструп "Язык программирования C++"- М., Бином, 2010 г.
5. MSDN Library for Visual Studio 2005 (Microsoft Document Explorer – входит в состав дистрибутива VS. Нужно обязательно развернуть при установке!)
6. Общее методическое пособие по курсу для выполнения ЛР и КЛР/ДЭ (см. на сайте 1-й курс www.sergebolshakov.ru) – см. кнопку в конце каждого раздела сайта!!!
7. Г.С.Иванова, Т.Н. Ничушкина, Е.К.Пугачев "Объектно-ориентированное программирование". – М., МГТУ, 2001 г.
8. Другие методические материалы по дисциплине с сайта www.sergebolshakov.ru.
9. Конспекты лекций по дисциплине “Программирование на основе классов и шаблонов”.
10. Эккель Б. Философия C++. Введение в стандартный C++. 2-е изд.- СПб.: Питер, 2004.- 572с.: ил.
11. Эккель Б. Эллисон Ч. Философия C++. Практическое программирование. - СПб.: Питер, 2004.- 608с.: ил.
12. Страуструп, Бьярн. Программирование: принципы и практика с использованием C++, 2-е изд. : Пер. с англ. - М . : ООО "И . Д. Вильяме", 2016. - 1328с. : ил . - Парал. тит. англ. ISBN 978-5-8459- 1 949-6 (рус .)
13. Страуструп Б. "Дизайн и эволюция C++. Классика CS" – СПб,: Питер , 2007. – 445с.

12. Справочные материалы

12.1. Методы vector

Конструкторы класса **vector**:

vector	Constructs a vector of a specific size or with elements of a specific value or with a specific **allocator** or as a copy of some other vector.

Определения typedefs vector:

allocator_type	A type that represents the **allocator** class for the vector object.
const_iterator	A type that provides a random-access iterator that can read a **const** element in a vector.
const_pointer	A type that provides a pointer to a **const** element in a vector.
const_reference	A type that provides a reference to a **const** element stored in a vector for reading and performing **const** operations.

const_reverse_iterator	A type that provides a random-access iterator that can read any **const** element in the vector.
difference_type	A type that provides the difference between the addresses of two elements in a vector.
iterator	A type that provides a random-access iterator that can read or modify any element in a vector.
pointer	A type that provides a pointer to an element in a vector.
reference	A type that provides a reference to an element stored in a vector.
reverse_iterator	A type that provides a random-access iterator that can read or modify any element in a reversed vector.
size_type	A type that counts the number of elements in a vector.
value_type	A type that represents the data type stored in a vector.

Методы класса vector:

assign	Erases a vector and copies the specified elements to the empty vector.
at	Returns a reference to the element at a specified location in the vector.
back	Returns a reference to the last element of the vector.
begin	Returns a random-access iterator to the first element in the container.
capacity	Returns the number of elements that the vector could contain without allocating more storage.
clear	Erases the elements of the vector.
empty	Tests if the vector container is empty.
end	Returns a random-access iterator that points just beyond the end of the vector.
erase	Removes an element or a range of elements in a vector from specified positions.
front	Returns a reference to the first element in a vector.
get_all	Returns an object to the **allocator** class used by a vector.

ocator	
insert	Inserts an element or a number of elements into the vector at a specified position.
max_size	Returns the maximum length of the vector.
pop_back	Deletes the element at the end of the vector.
push_back	Add an element to the end of the vector.
rbegin	Returns an iterator to the first element in a reversed vector.
rend	Returns an iterator to the end of a reversed vector.
resize	Specifies a new size for a vector.
reserve	Reserves a minimum length of storage for a vector object.
size	Returns the number of elements in the vector.
swap	Exchanges the elements of two vectors.

Операторы класса vector:

operator[]	Returns a reference to the vector element at a specified position.

12.2. Методы CArray

Методы класса CArray.

CArray	Constructs an empty array.
GetCount	Gets the number of elements in this array.
GetSize	Gets the number of elements in this array.
GetUpperBound	Returns the largest valid index.
IsEmpty	Determines whether the array is empty.
SetSize	Sets the number of elements to be contained in this array.
FreeExtra	Frees all unused memory above the current upper bound.

RemoveAll	Removes all the elements from this array.
RelocateElements	Relocates data to a new buffer when the array should grow or shrink.
ElementAt	Returns a temporary reference to the element pointer within the array.
GetAt	Returns the value at a given index.
GetData	Allows access to elements in the array. Can be **NULL**.
SetAt	Sets the value for a given index; array not allowed to grow.
Add	Adds an element to the end of the array; grows the array if necessary.
Append	Appends another array to the array; grows the array if necessary
Copy	Copies another array to the array; grows the array if necessary.
SetAtGrow	Sets the value for a given index; grows the array if necessary.

InsertAt	Inserts an element (or all the elements in another array) at a specified index.
RemoveAt	Removes an element at a specific index.

Операторы:

operator []	Sets or gets the element at the specified index.

12.3. Методы CObArray

Методы класса CObArray.

CObArray	Constructs an empty array for CObject pointers.
GetCount	Gets the number of elements in this array.
GetSize	Gets the number of elements in this array.
GetUpperBound	Returns the largest valid index.

	SetSize	Sets the number of elements to be contained in this array.
	FreeExtra	Frees all unused memory above the current upper bound.
	IsEmpty	Determines if the array is empty.
	RemoveAll	Removes all the elements from this array.
	ElementAt	Returns a temporary reference to the element pointer within the array.
	GetAt	Returns the value at a given index.
	GetData	Allows access to elements in the array. Can be NULL.
	SetAt	Sets the value for a given index; array not allowed to grow.
	Add	Adds an element to the end of the array; grows the array if necessary.
	Append	Appends another array to the array; grows the array if necessary.
	Copy	Copies another array to the array; grows the array if necessary.
	SetAtGrow	Sets the value for a given index; grows the array if necessary.
	InsertAt	Inserts an element (or all the elements in another array) at a specified index.
	RemoveAt	Removes an element at a specific index.

Операции

	operator []	Sets or gets the element at the specified index.

12.4. Методы CAtlArray

Методы CAtlArray

	Add	Call this method to add an element to the array object.

Append	Call this method to add the contents of one array to the end of another.
AssertValid	Call this method to confirm that the array object is valid.
CAtlArray	The constructor.
~CAtlArray	The destructor.
Copy	Call this method to copy the elements of one array to another.
FreeExtra	Call this method to remove any empty elements from the array.
GetAt	Call this method to retrieve a single element from the array object.
GetCount	Call this method to return the number of elements stored in the array.
GetData	Call this method to return a pointer to the first element in the array.
InsertArrayAt	Call this method to insert one array into another.
InsertAt	Call this method to insert a new element (or multiple copies of an element) into the array object.
IsEmpty	Call this method to test if the array is empty.
RemoveAll	Call this method to remove all elements from the array object.
RemoveAt	Call this method to remove one or more elements from the array.
SetAt	Call this method to set the value of an element in the array object.
SetAtGrow	Call this method to set the value of an element in the array object, expanding the array as required.
SetCount	Call this method to set the size of the array object.

Операции

oper	Call this operator to return a reference to an element in the

ator []	array.