ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ БЮДЖЕТНОЕ ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ
«ОРЕНБУРГСКИЙ ГОСУДАРСТВЕННЫЙ АГРАРНЫЙ УНИВЕРСИТЕТ»

Методические рекомендации для самостоятельной работы обучающихся по дисциплине

Б1.В.16 Объектно-ориентированное программирование

Направление подготовки (специальность) 27.03.04 Управление в технических системах

Профиль образовательной программы Интеллектуальные системы обработки информации и управления

Квалификация (степень) выпускника бакалавр

Форма обучения очная

Содержание

1. ОРГАНИЗАЦИЯ САМОСТОЯТЕЛЬНОЙ РАБОТЫ

1.1 Организационно-методические данные дисциплины

№ п.п.	Наименование темы	Общий объем часов по видам самостоятельной работы (из табл. 5.1 РПД)				
		подготовка курсового проекта (работы)	подготовка реферата/эссе	индивидуальные домашние задания (ИДЗ)	самостоятельное изучение вопросов (СИВ)	подготовка к занятиям (ПкЗ)
1	2	3	4	5	6	7
1	Эволюция методологий программирования				1	1
2	Составные части объектного подхода				1	1
3	Понятие объекта. Свойства				1	1
4	Отношения между объектами				1	1
5	Разработка Visual Basic-приложений. Создание программного интерфейса пользователя				1	1
6	Интеграция приложений				1	1
8	Процедуры и функции. Рекурсивные подпрограммы Интерактивная форма				1	2
9	Функциональные возможности технологии доступа ADO в проектах для				1	1

	работы с локальными БД. Интерактивная форма					
10	Природа классов Интерактивная форма				2	2
11	UML-унифицированный язык моделирования. Четырехуровневая метамодель MOF Интерактивная форма				2	2
12	Отношения между классами Интерактивная форма				2	2
13	Отношения между классами и объектами				2	2
14	Представление объектов и классов				2	2
15	Реализация отношений между объектами и классами				2	2
16	Наследование как средство организации иерархий классов				4	4
17	Шаблоны классов, функций, специализация, наследование и шаблоны				4	4
18	Библиотека стандартных шаблонов. Библиотека ввода-вывода				8	6

2. МЕТОДИЧЕСКИЕ РЕКОМЕНДАЦИИ ПО

САМОСТОЯТЕЛЬНОМУ ИЗУЧЕНИЮ ВОПРОСОВ

2.1Дизайн и проектирование

Шаблон проектирования или паттерн (англ. design pattern) в разработке программного обеспечения — повторимая архитектурная конструкция, представляющая собой решение проблемы проектирования в рамках некоторого часто возникающего контекста.
Обычно шаблон не является законченным образцом, который может быть прямо преобразован в код; это лишь пример решения задачи, который можно использовать в различных ситуациях. Объектно-ориентированные шаблоны показывают отношения и взаимодействия между классами или объектами, без определения того, какие конечные классы или объекты приложения будут использоваться.
«Низкоуровневые» шаблоны, учитывающие специфику конкретного языка программирования, называются идиомами. Это хорошие решения проектирования, характерные для конкретного языка или программной платформы, и потому не универсальные.
На наивысшем уровне существуют архитектурные шаблоны, они охватывают собой архитектуру всей программной системы.
Алгоритмы по своей сути также являются шаблонами, но не проектирования, а вычисления, так как решают вычислительные задачи.

2.2 Разработка программного комплекса «Растровый графический редактор»

Для обработки изображений на компьютере используются специальные программы — графические редакторы. Графический редактор — это программа создания, редактирования и просмотра графических изображений. Графические редакторы можно разделить на две категории: растровые и векторные.

Растровые графические редакторы. Растровые графические редакторы являются наилучшим средством обработки фотографий и рисунков, поскольку растровые изображения обеспечивают высокую точность передачи градаций цветов и полутонов. Среди растровых графических редакторов есть простые, например стандартное приложение Paint, и мощные профессиональные графические системы, например Adobe Photoshop и CorelPhoto-Paint.

Растровое изображение хранится с помощью точек различного цвета (пикселей), которые образуют строки и столбцы. Любой пиксель имеет фиксированное положение и цвет. Хранение каждого пикселя требует некоторого количества бит информации, которое зависит от количества цветов в изображении.

Качество растрового изображения определяется размером изображения (числом пикселей по горизонтали и вертикали) и количества цветов, которые могут принимать пиксели.

Растровые изображения очень чувствительны к масштабированию (увеличению или уменьшению). Когда растровое изображение уменьшается, несколько соседних точек превращаются в одну, поэтому теряется разборчивость мелких деталей изображения. При укрупнении изображения увеличивается размер каждой точки и появляется ступенчатый эффект, который виден невооруженным глазом. Векторные графические редакторы. Векторные графические изображения являются оптимальным средством для хранения

высокоточных графических объектов (чертежи, схемы и т. д.), для которых имеет значение наличие четких и ясных контуров. С векторной графикой вы сталкиваетесь, когда работаете с системами компьютерного черчения и автоматизированного проектирования, с программами обработки трехмерной графики.

К векторным графическим редакторам относятся графический редактор, встроенный в текстовый редактор Word. Среди профессиональных векторных графических систем наиболее распространены CorelDRAW и Adobe Illustrator.

Векторные изображения формируются из объектов (точка, линия, окружность и т. д.), которые хранятся в памяти компьютера в виде графических примитивов и описывающих их математических формул.

Например, графический примитив точка задается своими координатами (X, У), линия — координатами начала (XI, У1) и конца (X2, У2), окружность — координатами центра (X, У) и радиусом (Я), прямоугольник — величиной сторон и координатами левого верхнего угла (XI, У1) и правого нижнего угла (X2, У2) и т. д. Для каждого примитива назначается также цвет.

Доистоинством векторной графики является то, что файлы, хранящие векторные графические изображения, имеют сравнительно небольшой объем. Важно также, что векторные графические изображения могут быть увеличены или уменьшены без потери качества.

Панели инструментов графических редакторов. Графические редакторы имеют набор инструментов для создания или рисования простейших графических объектов: прямой линии, кривой, прямоугольника, эллипса, многоугольника и т. д. После выбора объекта на панели инструментов его можно нарисовать в любом месте окна редактора. Выделяющие инструменты. В графических редакторах над элементами изображения возможны различные операции: копирование, перемещение, удаление, поворот, изменение размеров и т. д. Чтобы выполнить какую-либо операцию над объектом, его сначала необходимо выделить.

Для выделения объектов в растровом графическом редакторе обычно имеются два инструмента: выделение прямоугольной области и выделение произвольной области. Процедура выделения аналогична процедуре рисования.

Выделение объектов в векторном редакторе осуществляется с помощью инструмента выделение объекта (на панели инструментов изображается стрелкой). Для выделения объекта достаточно выбрать инструмент выделения и щелкнуть по любому объекту на рисунке.

Инструменты редактирования рисунка позволяют вносить в рисунок изменения: стирать его части, изменять цвета и т. д. Для стирания изображения в растровых графических редакторах используется инструмент Ластик, который убирает фрагменты изображения (пиксели), при этом размер Ластика можно менять.

В векторных редакторах редактирование изображения возможно только путем удаления объектов, входящих в изображение, целиком. Для этого сначала необходимо выделить объект, а затем выполнить операцию Вырезать.

Операцию изменения цвета можно осуществить с помощью меню Палитра,

содержащего набор цветов, используемых при создании или рисовании объектов.

Текстовые инструменты позволяют добавлять в рисунок текст и форматировать его.

В растровых редакторах инструментом Надпись (буква А на панели инструментов) создаются текстовые области на рисунках. Установив курсор в любом месте текстовой области, можно ввести текст. Форматирование текста производится с помощью панели Атрибуты текста. В векторных редакторах тоже можно создавать текстовые области для ввода и форматирования текста. Кроме того, надписи к рисункам вводятся посредством так называемых выносок различных форм.

Масштабирующие инструменты в растровых графических редакторах дают возможность увеличивать или уменьшать масштаб представления объекта на экране, не влияя при этом на его реальные размеры. Обычно такой инструмент называется Лупа.

В векторных графических редакторах легко изменять реальные размеры объекта с помощью мыши.

2.3 Интеграция приложений: Power Point и Word

Пакеты прикладных программ (ППП) − это специальным образом организованные программные комплексы, рассчитанные на общее применение в определенной проблемной области и дополненные соответствующей технической документацией.
В зависимости от характера решаемых задач выделяют различные виды ППП. Чтобы пользователь мог применить ППП для решения конкретной задачи, пакет должен обладать средствами настройки (иногда путём введения некоторых дополнений).
Каждый ППП обладает обычно рядом возможностей по методам обработки данных и формам их представления, полноте диагностики, что дает возможность пользователю выбрать подходящий для конкретных условий вариант.
ППП обеспечивают значительное снижение требований к уровню профессиональной подготовки пользователей в области программирования, вплоть до возможности эксплуатации пакета без программиста. Часто пакеты прикладных программ располагают базами данных для хранения данных и передачи их прикладным программам.

Как уже говорилось, самым распространенным в мире офисным пакетом является Microsoft Office. По данным International Data Corporation - одной из крупнейших компаний, работающих в области компьютерной аналитики, это приложение установлено более чем на 95% персональных компьютеров. Пакет Microsoft Office Standard Edition 2007 включает программы: Word 2007, Excel 2007, Outlook 2007, PowerPoint 2007 (возможно также включение в этот список программы Publisher). На возможностях этих программ мы кратко остановимся.
Microsoft Word - это многофункциональная система обработки текстов, обладающая полным набором средств, необходимых для быстрого создания и эффективной обработки документов практически любой степени сложности.
Microsoft Word - настолько широко распространенный текстовый процессор, что его файловый формат (.doc) стал стандартом дефакто для всех разработчиков офисных приложений.
Word обеспечивает редактирование текста, предоставляя пользователю разнообразные инструменты форматирования (на уровне символа, абзацев и разделов).
Форматированием называется изменение внешнего вида текста, при котором не изменяется его содержание.

На уровне символа программа обеспечивает символьное форматирование (шрифт, размер шрифта, расстояние между символами, полужирный текст, курсив, подчеркивание, цвет текста и т.п.).

На уровне абзаца осуществляется форматирование абзаца (способ выравнивания, межстрочные интервалы, обрамление абзацев, заливка абзаца, создание маркированных списков и т.д.).

Программа также позволяет форматировать разделы, то есть участки документа, в пределах которых сохраняют свой формат колонки, колонтитулы, нумерация страниц, сноски, поля и некоторые другие параметры. Совокупность форматирования символов, абзацев и разделов, а также параметров страницы (размер, фон и т.п.) определяет информацию о макете документа.

Word значительно облегчает работу при форматировании документа, предоставляя возможность использовать стили.

Стиль - это именованный и сохраненный набор параметров форматирования. Стиль может определять шрифт, его размер, межстрочный интервал, способ выравнивания текста по краям и т.д. Определив стиль, можно быстро применить его к любому фрагменту текста документа. Форматировать текст с помощью стилей намного быстрее, чем изменять вручную каждый элемент форматирования. Использование стиля гарантирует единообразие внешнего вида определенных элементов документа. При внесении изменений в определение стиля весь текст документа, к которому был применен этот стиль, изменится в соответствии с новым определением стиля. В Word есть целый ряд заранее определенных стилей, а также предусмотрена возможность создавать пользовательские. Кроме того, использование стилей позволяет автоматизировать такие процедуры, как составление оглавления и указателей.

Как правило, в организациях создается множество повторяющихся (типовых) документов, подготовку которых можно ускорить, используя специальные шаблоны. Шаблон позволяет быстро изготавливать новые, аналогичные по форме документы, не тратя времени на форматирование. Шаблоны могут содержать информацию о стилях, стандартных текстах и даже панели инструментов, что позволяет унифицировать и автоматизировать процесс подготовки документов. В комплект поставки Word входят шаблоны многих стандартных документов.

Таким образом, с помощью программы Word можно быстро оформить приказ, служебную записку, подготовить научно-технический отчет, письмо или любой другой документ, содержащий стандартные элементы.

Word предлагает встроенные средства графики, позволяющие включать в текст схемы, чертежи и диаграммы.

Важной особенностью программы является возможность поиска и замены фрагментов текста, сравнения версий документов, проверки правописания.

2.4 Запись файла

Класс **FileStream** представляет возможности по считыванию из файла и записи в файл. Он позволяет работать как с текстовыми файлами, так и с бинарными.

Рассмотрим наиболее важные его свойства и методы:

- Свойство **Length**: возвращает длину потока в байтах
- Свойство **Position**: возвращает текущую позицию в потоке
- Метод **Read**: считывает данные из файла в массив байтов. Принимает три параметра: int Read(byte[] array, int offset, int count) и возвращает количество успешно считанных байтов. Здесь используются следующие параметры:
    - array - массив байтов, куда будут помещены считываемые из файла данные
    - offset представляет смещение в байтах в массиве array, в который считанные байты будут помещены
    - count - максимальное число байтов, предназначенных для чтения. Если в файле находится меньшее количество байтов, то все они будут считаны.
- Метод **long Seek(long offset, SeekOrigin origin)**: устанавливает позицию в потоке со смещением на количество байт, указанных в параметре offset.
- Метод **Write**: записывает в файл данные из массива байтов. Принимает три параметра: Write(byte[] array, int offset, int count)
    - array - массив байтов, откуда данные будут записываться в файла
    - offset - смещение в байтах в массиве array, откуда начинается запись байтов в поток
    - count - максимальное число байтов, предназначенных для записи

FileStream представляет доступ к файлам на уровне байтов, поэтому, например, если вам надо считать или записать одну или несколько строк в текстовый файл, то массив байтов надо преобразовать в строки, используя специальные методы. Поэтому для работы с текстовыми файлами применяются другие классы.

В то же время при работе с различными бинарными файлами, имеющими определенную структуру FileStream может быть очень даже полезен для извлечения определенных порций информации и ее обработки.

2.5 Динамический список с произвольным запросом

чень часто, при разработке приложений, оперирующих с большим количеством входных данных, возникает вопрос об их хранении во время выполнения программы. Приводить все из них не имеет смысла, остановлюсь лишь на массивах. Несомненно, данный тип решает вопрос хранения данных, однако, очевидно, что он не лишен недостатков. Главным из них, несомненно, является его фиксированный размер. Это свойство не поддается изменению даже у динамически созданных массивов, что довольно часто заставляет программистов, использующих исключительно их, выделять память "с запасом". Ну а во-первых, даже "запас" ограничен, и никто не может дать гарантии, что и его будет достаточно, а во-вторых, наоборот, "запаса" может хватить настолько, что немалая часть отведенной программе памяти будет занята понапрасну.

Данную проблему решает другой тип хранения данных, которому и посвящена эта статья - связанный список динамических переменных, или проще - динамический список. Компоненты добавляются и удаляются во время выполнения программы, и их количество зависит исключительно от размера доступной памяти. Однако, за это преимущество приходится расплачиваться недостатком - если в случае с массивом, мы в любой момент получаем доступ к любому компоненту, то в случае со списком, в один момент времени нам доступны максимум 3 компонента (это зависит от способа представления списка в программе). В большинстве случаев, это очень даже приемлемая цена ...

В статье на примере решения несложной задачи, я попытаюсь продемонстрировать работу с динамическими списками, реализую основные операции над ним и его компонентами. Статья рассчитана на программистов C\C++, хорошо знакомых с синтаксисом языка, типами данных, имеющих представление и опыт применения указателей. Я программирую в VS 2008, однако в данном случае, IDE не имеет особого значения.

2. Постановка задачи

Для демонстрации реализации работы с динамическим списком, решим задачу:

"В текстовом файле находятся идентификаторы переменных и их числовые значения (например: x 15 abc 12.098 z -1.23). Перенести их в динамический список, для которого реализовать следующие операции: поиск идентификатора в списке; изменение значения указанного идентификатора; удаление идентификатора из списка, добавление в список нового идентификатора с заданным значением. По окончании сеанса работы список идентификаторов и их значений переносится обратно в файл"

Задача позволит нам научиться реализовывать все основные операции над динамическим списком - создание списка, добавление и удаление компонентов (узлов) списка, поиск по компонентам списка. Для решения понадобятся знания о файловом вводе-выводе, указателях, а также структурированных типах данных. Я не буду создавать никакой оболочки для этой задачи, хотя она была бы кстати, так как имеется целый спектр возможных операций. Однако это не является моей основной задачей в рамках этой статьи. Я лишь реализую все операции в виде отдельных функций и для пробы обращусь к ним по разу :

Решение
Стандартное начало

Приступим к решению. Создаем консольный проект, и пишем довольно стандартный код - организовываем файловый ввод. Для чтения из файла используем метод getline(), где в качестве третьего параметра указываем пробел, являющийся разделителем в нашем файле.

```
#include <fstream>

#include <iostream>

#include <cstdlib>

//TODO: Описание списка

//TODO: Операции со списком

using namespace std;

int main()

{
```

```
char* fileName = new char[50];
char* buf_name = new char[20];
char* buf_value = new char [10];

cout << "Enter name of file -> ";
cin >> fileName;
ifstream* inp = new ifstream(fileName);
if (!inp->good())
{
	cout << "File opening error!\n";
	system("PAUSE");
	return 0;
}
while (!inp->eof())
{
	inp->getline(buf_name, 20, ' ');
	inp->getline(buf_value, 10, ' ');
```

Кажется, тут все ясно и ничего не требует особого внимания. В первом разделе TODO мы опишем типы, необходимые для работы со списком, во втором у нас будут функции для работы с ним. Теперь приступим непосредственно к описанию списка, т.к. в программе мы уже вплотную подошли к его заполнению.

Описание динамического списка

Для начала, вспомним (или впервые узнаем)), что динамический список представляет из себя некоторое количество компонентов (узлов), содержащих непосредственно информационную часть (число, строка или более сложные типы данных), а также ссылку на следующий компонент (возможно, что узел содержит 2 ссылки: на следующий и на предыдущий, в таком случае, список называется двусвязным). Таким образом, получаем следующую структуру, описывающую компонент списка для нашей задачи:

```
struct comp {
	char name[20]; // Имя переменной
	char value[10]; // Значение переменной
	comp* next; //Ссылка на следущий элемент списка
};
```

Сам же список представляет из себя совокупность его узлов и задается обычно одной или двумя ссылками, на первый элемент и последний. Нередко, хватает и одной из этих ссылок, но в нашем случае удобнее будет использовать обе. Итак, получаем структуру, описывающую наш список:

```
struct dyn_list {
```

```
        comp* head; // Первый элемент (голова) списка
        comp* tail; // Последний элемент (хвост) списка
    };
```

Несложно понять, что значит создать пустой список. Фактически делать ничего не нужно, разве что присвоить ссылке на первый элемент списка (head) значение NULL. Оформим это как функцию, параллельно создав еще одну, проверяющую по этому условию, пуст ли список.

```
// Создание пустого списка
void constr_list(dyn_list &l)
{
    l.head = NULL;
}
// Проверка списка на пустоту
bool chk_empty(dyn_list l)
{
    return (l.head==NULL);
}
```

Видим, что все просто. Конечно, тип, описывающий список можно было оформи как класс, а функцию, создающую его сделать конструктором данного класса, но я оставлю это вам в качестве домашнего задания :. Теперь в функции main() описываем переменную типа dyn_list и создаем пустой список. Затем переходим к следующему пункту.

```
dyn_list vars; // Динамический список
constr_list(vars);
```

3.3 Операции с компонентами списка

Ну, перед тем, как проводить какие-то операции с компонентами, их необходимо добавить в список, чем мы сейчас и займемся. Прежде всего, нам необходимо создать новый компонент и заполнить его информационную часть. Так как мы будем помещать его в конец списка, то ссылочная часть узла будет иметь значение NULL. Теперь давайте поймем, что должно происходить со ссылками head и tail при этой операции. Здесь рассматриваются 2 случая - наш список пуст или в нем есть хотя бы один компонент. Если пуст, то и head и tail будут указывать на только что созданный компонент. Если же узлы в списке присутствуют, очевидно, что сначала нужно последний узел связать с добавляемым (для этого ссылочной части компонента, на который указывает tail, присвоить адрес того, который хотим добавить), а затем "передвинуть" tail. Вот как просто все это выглядит на C++:

```
// Включение в список нового компонента
void comp_in(dyn_list &l, char* n, char* v)
{
```

```
	comp* c = new comp();
	strcpy_s(c->name, 20, n);
	strcpy_s(c->value, 10, v);
	c->next = NULL;
	if (chk_empty(l))
		l.head = c;
	else
		l.tail->next = c;
	l.tail = c;
}
```

Теперь займемся поиском компонента. Искать будет по имени переменной, по желанию вы можете искать и по значению. В качестве аргументов, функции поиска передаем сам список и искомый текст. Возвращать наша функция будет адрес найденного узла или NULL, если ничего не найдено. Искать начнем с компонента, на который указывает head и, двигаясь вперед, сравнивать имя текущей переменной с искомым. В функции поиска мы можем не боясь, передвигать ссылку head, так как передаем ее не по ссылке.

```
// Поиск компонента в списке по имени
comp* search(dyn_list l, char *n)
{
	while (l.head != NULL)
	{
		if (!strcmp(l.head->name,n))
			return l.head;
		l.head = l.head->next;
	}
	return l.head;
}
```

А сейчас удалим компонент. В качестве аргумента, передаем по ссылке список, а также ссылку на компонент, который собираемся удалить. В самой же функции рассматриваем 2 случая. Первый случай простой: если компонент является первым (то есть на него указывает head), то достаточно лишь переместить ссылку на первый элемент вперед. В противном случае, нам понадобится рабочая переменная-узел, которую мы будем использовать для движения по списку. Двигаться будем до тех пор, пока следующий за текущим узел не будет тем, который мы собираемся удалить. Ну а после этого, "перепрыгиваем" удаляемый компонент, присваивая ссылочной части предшествующего удаляемому компоненту адрес следующего за удаляемым. Все эти слова умещаются буквально в несколько строк исходного кода:

```
// Удаление компонента из списка
```

```
void comp_del(dyn_list &l, comp* c)
{
	if (c == l.head)
	{
		l.head = c->next;
		return;
	}
	comp* r = new comp();
	r = l.head;
	while (r->next != c)
		r = r->next;
	r->next = c->next;
	delete c;
}
```

Последняя и самая простая операция - это изменение значения информационной части узла. Менять будем поле value. В качестве параметров, по ссылке передадим адрес компонента, а также новое значение изменяемого поля. Ну и в одну строчку все изменим. Вот как просто это выглядит:

```
// Изменение значения компонента
void comp_edit(comp &c, char* v)
{
	strcpy_s(c.value, 10, v);
}
```

С операциями закончили, теперь осталось дописать программу.

Тестируем функции и завершаем программу

Ну, тут все просто, в цикле нам осталось лишь обратиться к функции добавления компонента в список. После цикла, тестируем остальные наши функции и выводим весь список в файл. Привожу здесь весь код функции main()

```
int main()
{
	char* fileName = new char[50];
	char* buf_name = new char[20];
	char* buf_value = new char [10];
	dyn_list vars; // Динамический список
	cout << "Enter name of file -> ";
```

```
cin >> fileName;
ifstream* inp = new ifstream(fileName);
if (!inp->good())
{
	cout << "File opening error!\n";
	system("PAUSE");
	return 0;
}
constr_list(vars);
while (!inp->eof())
{
	inp->getline(buf_name, 20, ' ');
	inp->getline(buf_value, 10, ' ');
	comp_in(vars, buf_name, buf_value);
}
inp->close();
comp* p = new comp();
p = search(vars, "z");
if (p)
	comp_del(vars, p);
p = search(vars, "abc");
if (p)
	comp_edit(*p, "2");
ofstream* out = new ofstream(fileName);
while (vars.head != NULL)
{
	out->write(vars.head->name, strlen(vars.head->name));
	out->write(" ",1);
	out->write(vars.head->value, strlen(vars.head->value));
	out->write(" ",1);
	vars.head = vars.head->next;
}
out->close();
system("PAUSE");
```

```
    return 0;
}
```

Задача решена :

Выводы

Несмотря на то, что кому-то могла показаться сложной работа с динамическим списком, на деле все оказывается не только довольно просто, но еще и удобно - компоненты создаются и удаляются динамически, размерность списка ограничивается лишь доступной памятью, после того, как компонент больше не используется, память можно сразу же высвободить для других узлов.

В любом случае, использовать связанные списки в своих программах или нет, придется решать вам, надеюсь, моя статья хоть немного поможет вам в этом. Спасибо за внимание.

2.6 Иерархия классов первого и второго порядка

Объект - это абстрактная сущность, наделенная характеристиками объектов окружающего нас реального мира.

Создание объектов и манипулирование ими - это вовсе не привилегия языка C++, а скорее результат методологии программирования, воплощающей в кодовых конструкциях описания объектов и операции над ними. Каждый объект программы, как и любой реальный объект, отличается собственными атрибутами и характерным поведением. Объекты можно классифицировать по разным категориям: например, мои цифровые наручные часы "Cassio" принадлежат к классу часов. Программная реализация часов входит, как стандартное приложение, в состав операционной системы вашего компьютера.

Каждый класс занимает определенное место в иерархии классов, например, все часы принадлежат классу приборов измерения времени (более высокому в иерархии), а класс часов сам включает множество производных вариаций на ту же тему. Таким образом, любой класс определяет некоторую категорию объектов, а всякий объект есть экземпляр некоторого класса.

Объектно-ориентированное программирование, ООП - это методика, которая концентрирует основное внимание программиста на связях между объектами, а не на деталях их реализации.

В этой лекции основные принципы ООП (инкапсуляция, наследование, полиморфизм, создание классов и объектов) интерпретируются и дополняются новыми понятиями и терминологией, принятыми интегрированной средой визуальной обработки C++Builder. Приводится описание расширений языка новыми возможностями (компоненты, свойства, обработчики событий) и последних дополнений стандарта ANSI C++ (шаблоны, пространства имен, явные и непостоянные объявления, идентификация типов при выполнении программы, исключения).

Раздел носит обзорный характер, он призван познакомить читателя со специальной терминологией ООП, к которой лектор вынужден прибегать на протяжении всего курса. Это вызвано тем, что C++Builder является типичной системой ООП и претендует на кульминационную роль в истории его развития.

Инкапсуляция

Инкапсуляция - есть объединение в едином объекте данных и кодов, оперирующих с этими данными. В терминологии ООП данные называются членами данных (data members) объекта, а коды - объектными методами или функциями-членами (methods, member functions).

Инкапсуляция позволяет в максимальной степени изолировать объект от внешнего окружения. Она существенно повышает надежность разрабатываемых программ, т.к. локализованные в объекте функции обмениваются с программой сравнительно небольшими объемами данных, причем количество и тип этих данных обычно тщательно контролируются. В результате замена или модификация функций и данных, инкапсулированных в объект, как правило, не влечет за собой плохо прослеживаемых последствий для программы в целом (в целях повышения защищенности программ в ООП почти не используются глобальные переменные).

Другим немаловажным следствием инкапсуляции является легкость обмена объектами, переноса их из одной программы в другую. Простота и доступность принципа инкапсуляции ООП стимулирует программистов к расширению Библиотеки Визуальных Компонент (VCL), входящей в состав C++Builder.

Классы, компоненты и объекты

Класс не имеет физической сущности, его ближайшей аналогией является объявление структуры. Память выделяется только тогда, когда класс используется для создания объекта. Этот процесс также называется созданием экземпляра класса (class instance).

Любой объект языка C++ имеет одинаковые атрибуты и функциональность с другими объектами того же класса. За создание своих классов и поведение объектов этих классов полную ответственность несет сам программист. Работая в некоторой среде, программист получает доступ к обширным библиотекам стандартных классов (например, к Библиотеке Визуальных Компонент C++Builder).

Обычно, объект находится в некотором уникальном состоянии, определяемом текущими значениями его атрибутов. Функциональность объектного класса определяется возможными операциями над экземпляром этого класса.

Определение класса в языке C++ содержит инкапсуляцию членов данных и методов, оперирующих с членами данных и определяющих поведение объекта. Возвращаясь к нашему примеру, отметим, что жидкокристаллический дисплей часов "Casio" представляет член данных этого объекта, а кнопки управления - объектные методы. Нажимая кнопки часов, можно оперировать с установками времени на дисплее, т.е. следуя терминологии ООП, методы модифицируют состояние объекта путем изменения членов данных.

C++Builder вводит понятие компонент (components) - специальных классов, свойства которых представляют атрибуты объектов, а их методы реализуют операции над соответствующими экземплярами компонентных классов. Понятие метод обычно используется в контексте компонентных классов и внешне не отличается от терминафункция-член обычного класса. C++Builder позволяет манипулировать видом и функциональным поведением компонент не только с помощью методов (как это делают функции-члены обычных классов), но и посредством свойств и событий, присущих только классам компонент. Работая в среде C++Builder, вы наверняка заметите, что

манипулировать с компонентным объектом можно как на стадии проектирования приложения, так и во время его выполнения.

Свойства компонент, component properties - представляют собой расширение понятия членов данных и хотя не хранят данные как таковые, однако обеспечивают доступ к членам данных объекта.

C++Builder использует ключевое слово **_property** для объявления свойств. При помощи событий (events) компонента сообщает пользователю о том, что на нее оказано некоторое предопределенное воздействие. Основная сфера применения методов в программах, разрабатываемых в среде C++Builder - это обработчики событий (event handlers), которые реализуют реакцию программы на возникновение определенных событий. Легко заметить некоторое сходство событий и сообщений операционной системы Windows. Типичные простые события —нажатие кнопки или клавиши на клавиатуре. Компоненты инкапсулируют свои свойства, методы и события.

На первый взгляд компоненты ничем не отличаются от других объектных классов языка C++, за исключением ряда особенностей, среди которых пока отметим следующие:

• Большинство компонент представляют собой элементы управления интерфейсом с пользователем, причем некоторые обладают весьма сложным поведением.

• Все компоненты являются прямыми или косвенными потомками одного общего класса-прародителя (TComponent).

• Компоненты обычно используются непосредственно, путем манипуляции с их свойствами; они сами не могут служить базовыми классами для построения новых подклассов.

• Компоненты размещаются только в динамической памяти кучи (heap) с помощью оператора **new,** а не на стеке, как объекты обычных классов.

• Свойства компонент заключают в себе RTTI - идентификацию динамических типов.

• Компоненты можно добавлять к Палитре компонент и далее манипулировать с ними посредством Редактора форм интегрированной среды визуальной разработки C++Builder.

ООП интерпретирует взаимодействие с объектами как посылку запросов некоторому объекту или между объектами. Объект, принявший запрос, реагирует вызовом соответствующего метода. В отличие от других языков ООП, таких как SmallTalk, C++ не поощряет использование понятия "запрос". Запрос - это то, что делается с объектом, а метод - это то, как объект реагирует на поступивший запрос.

При ближайшем рассмотрении метод оказывается обычной функцией-членом, которая включена в определение класса. Чтобы вызвать метод, надо указать имя функции в контексте данного класса или в обработчике некоторого события.

Именно скрытая связь метода с включающим классом выделяет его из понятия простой функции. Во время выполнения метода он имеет доступ ко всем данным своего класса, хотя и не требует явной спецификации имени этого класса. Это обеспечивается передачей каждому, без исключения, методу скрытого параметра - непостоянного указателя **this** на

экземпляр класса. При любом обращении метода к членам данных класса, компилятор генерирует специальный код, использующий указатель **this.**

Наследование

Одной из самых восхитительных особенностей живой природы является ее способность порождать потомство, обладающее характеристиками, сходными с характеристиками предыдущего поколения. Заимствованная у природы идея наследования решает проблему модификации поведения объектов и придает ООП исключительную силу и гибкость. Наследование позволяет, практически без ограничений, последовательно строить и расширять классы, созданные вами или кем-то еще. Начиная с самых простых классов, можно создавать производные классы по возрастающей сложности, которые не только легки в отладке, но и просты по внутренней структуре.

Последовательное проведение в жизнь принципа наследования, особенно при разработке крупных программных проектов, хорошо согласуется с техникой нисходящего структурного программирования (от общего к частному), и во многом стимулирует такой подход. При этом сложность кода программы в целом существенно сокращается. Производный класс (потомок) наследует все свойства, методы и события своего базового класса (родителя) и всех его предшественников в иерархии классов.

При наследовании базовый класс обрастает новыми атрибутами и операциями. В производном классе обычно объявляются новые члены данных, свойства и методы. При работе с объектами программист обычно подбирает наиболее подходящий класс для решения конкретной задачи и создает одного или нескольких потомков от него, которые приобретают способность делать не только то, что заложено в родителе. Дружественные функции позволяют производному классу получить доступ ко всем членам данных внешних классов.

Кроме того, производный класс может перегружать (overload) наследуемые методы в том случае, когда их работа в базовом классе не подходит потомку. Использование перегрузки в ООП всячески поощряется, хотя в прямом понимании значения этого слова перегрузок обычно избегают. Говорят, что метод перегружен, если он ассоциируется с более чем одной одноименной функцией. Обратите внимание, что механизм вызовов перегруженных методов в иерархии классов полностью отличается от вызовов переопределенных функций. Перегрузка и переопределение - это разные понятия. Виртуальные методы используются для переопределения функций базового класса.

Чтобы применить концепцию наследования, к примеру, с часами, положим, что следуя принципу наследования, фирма "Casio" решила выпустить новую модель, дополнительно способную, скажем, произносить время при двойном нажатии любой из существующих кнопок. Вместо того, чтобы проектировать заново модель говорящих часов (новый класс, в терминологии ООП), инженеры начнут с ее прототипа (произведут нового потомка базового класса, в терминологии ООП). Производный объект унаследует все атрибуты и функциональность родителя. Произносимые синтезированным голосом цифры станут новыми членами данных потомка, а объектные методы кнопок должны быть перегружены, чтобы реализовать их дополнительную функциональность. Реакцией на событие двойного нажатия кнопки станет новый метод, который реализует произнесение последовательности цифр (новых членов данных), соответствующей текущему времени. Все вышесказанное в полной мере относится к программной реализации говорящих часов.

2.7 Структура хранения системы ограничений

Традиционные файловые системы, реализованные в рамках различных ОС, имеют ряд ограничений, рассмотренных ниже и препятствующих их широкому использованию для решения информационных задач.

Эти ограничения следующие:

- Разделение и изоляция данных.
- Дублирование данных.
- Зависимость от программ и данных.
- Несовместимость форматов файлов.
- Фиксированные запросы и быстрое увеличение количества приложений.

Разделение и изоляция данных. Данные изолированы в отдельных файлах, что требует значительных трудозатрат при извлечении соответствующей поставленным условиям выборки информации, часто - выполнения синхронной обработки нескольких файлов.

Дублирование данных. Из-за децентрализованной работы с информацией в файловых системах фактически выполняется бесконтрольное дублирование данных, что, во первых, является причиной неэкономного расходования ресурсов, во вторых, может привести к нарушению целостности данных (например, сведения о сотруднике организации, формируемые в отделе кадров и в бухгалтерии могут стать противоречивыми). .

Зависимость от данных. В файловых системах физическая структура и способ хранения записей файлов данных жестко зафиксированы в коде программ приложений. Это значит, что изменить существующую структуру данных достаточно сложно. Данная особенность файловых систем называется зависимостью от программ и данных (program-data dependence).

Несовместимость форматов файлов. Поскольку структура файлов определяется кодом приложений, она также зависит от языка программирования этого приложения. Например, структура файла, созданного программой на языке COBOL, может совершенно отличаться от структуры файла, создаваемого программой на языке С. Прямая несовместимость таких файлов затрудняет процесс их совместной обработки.

Фиксированные запросы и быстрое увеличение количества приложений. В процессе работы у пользователей постоянно возрастают требования к реализации новых запросов к данным, хранящимся в файлах. При этом реализация запросов осуществляется программистом в виде приложений, что, соответственно, ведет к увеличению их количества. В процессе реализации часто нарушаются меры по обеспечению безопасности или целостности данных, не предусматриваются средства восстановления в случае сбоя аппаратного или программного обеспечения, доступ к файлам часто ограничивается одним пользователем.

Системы с базами данных

Все перечисленные выше ограничения файловых систем являются следствием двух факторов.

- Определение данных содержится внутри приложений, а не хранится отдельно и независимо от них.
- Помимо приложений не предусмотрено никаких других инструментов доступа к данным и их обработки.

Для повышения эффективности работы был разработан подход, основанный на использовании баз данных (database) и систем управления базами данных, или СУБД (Database Management System - DBMS). В этом разделе представлено достаточно формальное определение этих терминов, а также рассмотрены компоненты среды СУБД.

База данных - набор совместно используемых логически связанных данных, сопровождаемый описанием этих данных, предназначенный для удовлетворения информационных потребностей групп пользователей.
Рассмотрим это определение более внимательно. База данных - это совокупность данных, которые однократно определяются, а затем используются одновременно многими пользователями. Вместо разрозненных файлов с избыточными данными, что присуще файловым системам, здесь все данные собраны вместе с минимальной долей избыточности. При этом база данных хранит не только рабочие данные, но и их описания. По этой причине базу данных еще называют набором интегрированных записей с самоописанием. В совокупности, описание данных называется системным каталогом (system catalog), или словарем данных (data dictionary), а сами элементы описания принято называть метаданными (meta-data), т.е. "данными о данных". Именно наличие самоописания данных в базе данных обеспечивает в ней независимость между программами и данными (program-data independence). Структура данных при этом отделена от приложений и хранится в базе данных. Добавление новых структур данных или изменение существующих никак не влияет на приложения, при условии, что они не зависят непосредственно от изменяемых компонентов.
И, наконец, следует объяснить последний термин из определения базы данных, а именно понятие "логически связанный". При анализе информационных потребностей пользователей следует выделить сущности, атрибуты и связи. Сущностью (entity) называется отдельный тип объекта (человек, изделие, понятие или событие), который нужно представить в базе данных. Атрибутом (attribute) называется свойство, которое описывает некоторую характеристику описываемого объекта; связь (relationship) - это то, что объединяет несколько сущностей. Подобная БД представляет сущности, атрибуты и логические связи между объектами. Иначе говоря, база данных содержит логически связанные данные.
СУБД - это программное обеспечение, с помощью которого пользователи могут определять, создавать и поддерживать базу данных, а также осуществлять к ней контролируемый доступ. СУБД взаимодействует с прикладными программами пользователя и базой данных и обладает приведенными ниже возможностями:

- Позволяет определять базу данных, что обычно осуществляется с помощью языка определения данных (DDL - Data Definition Language). Язык DDL предоставляет пользователям средства указания типа данных и их структуры, а также средства задания ограничений для информации, хранимой в базе данных.
- Позволяет вставлять, обновлять, удалять и извлекать информацию из базы данных, что обычно осуществляется с помощью языка управления данными (DML - Data Manipulation Language). Наличие централизованного хранилища всех данных и их описаний позволяет использовать язык DML как общий инструмент организации запросов, который иногда называют языком запросов (query language). Наличие языка запросов позволяет устранить присущие файловым системам ограничения, при которых пользователям приходится иметь дело только с фиксированным набором запросов или постоянно возрастающим количеством программ. Существует две разновидности языков DML - процедурные (procedural) и непроцедурные (non-procedural) языки, которые отличаются между собой способом извлечения данных. Основное отличие между ними заключается в том, что процедурные языки обычно обрабатывают информацию в базе данных последовательно, запись за записью, а непроцедурные оперируют сразу целыми наборами записей. Поэтому с помощью процедурных языков

DML обычно указывается, как можно получить желаемый результат, тогда как непроцедурные языки DML используются для описания того, что следует получить. Наиболее распространенным типом непроцедурного языка является язык структурированных запросов (Structured Query Language - SQL), который в настоящее время определяется специальным стандартом и фактически является обязательным языком для любых реляционных СУБД.

- Предоставляет контролируемый доступ к базе данных с помощью: системы обеспечения безопасности, предотвращающей несанкционированный доступ к базе данных со стороны пользователей; системы поддержки целостности данных, обеспечивающей непротиворечивое состояние хранимых данных; системы управления параллельной работой приложений, контролирующей процессы их совместного доступа к базе данных; системы восстановления, позволяющей восстановить базу данных до предыдущего непротиворечивого состояния, нарушенного в результате сбоя аппаратного или программного обеспечения; доступного пользователям каталога, содержащего описание хранимой в базе данных информации.

При этом СУБД содержат в себе механизм создания представлений (view), который позволяет любому пользователю иметь свой собственный взгляд на базу данных. Язык DDL включает средства определения представлений, каждое из которых является некоторым подмножеством базы данных. Кроме того, представления обеспечивают:

- Дополнительный уровень безопасности за счет возможности исключения данных, которые не должны видеть некоторые пользователи. Например, можно создать некоторое представление, которое позволит руководителю организации и сотрудникам расчетного сектора бухгалтерии просматривать все данные о персонале, включая сведения об их зарплате. В то же время для организации доступа к данным других пользователей можно создать еще одно представление, из которого все сведения о зарплате будут исключены.
- Настройку внешнего интерфейса базы данных.
- Сохранение внешнего интерфейса БД непротиворечивым и неизменным даже при внесении изменений в ее структуру - например, при добавлении или удалении полей, изменении связей, разбиении файлов, их реорганизации или переименовании. Таким образом, представление обеспечивает полную независимость программ от реальной структуры данных, что позволяет устранить важнейший недостаток файловых систем.

При этом необходимо иметь в виду, что реальный объем функциональных возможностей, предлагаемых в различных СУБД, отличается от продукта к продукту. Например, в СУБД для персонального компьютера может не поддерживаться параллельный совместный доступ, а управление режимом безопасности, поддержанием целостности данных и восстановлением будет присутствовать только в очень ограниченной степени. Однако современные многопользовательские СУБД предлагают все перечисленные выше функциональные возможности и многое другое. Программное обеспечение СУБД постоянно совершенствуется и расширяется, чтобы удовлетворять растущим требованиям пользователей. Например, в некоторых приложениях теперь требуется хранить графику, видео, звук и т.д.

Банк данных - универсальные базы данных, предназначенные для обслуживания прикладных программ, вместе с соответствующими СУБД.
Информационно-поисковая система - частный случай банка данных, предназначенный для хранения, поиска и вывода на устройства ЭВМ необходимой информации.
Компоненты среды СУБД. В общем случае в среде СУБД можно выделить следующие пять основных компонентов: аппаратное обеспечение, программное обеспечение, данные, процедуры и пользователи.
Аппаратное обеспечение. Для работы СУБД и приложений необходимо некоторое аппаратное обеспечение. Оно может варьироваться в очень широких пределах - от единственного персонального компьютера или одного мейнфрейма до информационно-вычислительной сети. При этом одни СУБД предназначены для работы только с конкретными типами операционных систем или оборудования, другие могут работать с широким кругом аппаратного обеспечения и различными операционными системами.
Программное обеспечение. Этот компонент охватывает программное обеспечение самой СУБД и прикладных программ, вместе с операционной системой, включая и сетевое программное обеспечение, если СУБД используется в сети. Обычно приложения создаются на языках третьего поколения, таких как C, COBOL, Fortran, Ada или Pascal, или на языках четвертого поколения, таких как SQL, операторы которых внедряются в программы на языках третьего поколения. Кроме того, СУБД может иметь свои собственные инструменты четвертого поколения, предназначенные для быстрой разработки приложений с использованием встроенных непроцедурных языков запросов, генераторов отчетов, форм, графических изображений.
Данные. Вероятно, самым важным компонентом среды СУБД (с точки зрения конечных пользователей) являются данные, играющие роль моста между компьютером и человеком. База данных содержит как рабочие данные, так и мета-данные, структура базы данных называется схемой.
В системном каталоге содержатся следующие сведения:

- имена, типы и размеры элементов данных;
- имена связей;
- ограничения целостности данных;
- имена зарегистрированных пользователей, которым предоставлены некоторые права доступа к данным;
- используемые индексы и структуры хранения - например, инвертированные файлы.

Процедуры.К процедурам относятся инструкции и правила, которые должны учитываться при проектировании и использовании базы данных. Пользователям и обслуживающему персоналу базы данных необходимо предоставить документацию, содержащую подробное описание процедур использования и сопровождения данной системы, включая инструкции о правилах выполнения действий по работе с СУБД и БД (регистрация в СУБД, запуск и останов СУБД, создание резервных копий БД и т.п.).
Пользователи. Последним компонентом среды СУБД являются пользователи системы. Этот компонент подробно обсуждается в разделе 5.1.3.
Предварительные замечания по разработке базы данных .До сих пор по умолчанию предполагалось, что данные в базе обладают некоторой структурой, определяемой во время ее проектирования. Однако сам процесс проектирования базы данных оказывается, как правило, чрезвычайно сложным. Для успешной реализации

системы на основе базы данных необходимо подумать прежде всего о данных и лишь потом о приложениях.

К сожалению, существующие методологии проектирования баз данных пока не получили широкого распространения. Именно это обстоятельство часто является основной причиной неудач при разработке информационных систем. Из-за отсутствия структурированных подходов к проектированию баз данных необходимые для проведения разработки время и ресурсы обычно недооцениваются, а созданные базы данных часто неэффективны или не отвечают требованиям прикладных приложений. Предоставляемая документация часто бывает недостаточна, что чрезвычайно затрудняет сопровождение созданной БД.

Распределение обязанностей в системах с базами данных

В этом разделе мы рассмотрим упомянутый выше пятый компонент среды СУБД - ее пользователей. Среди них можно выделить три различные группы: администраторы данных и баз данных, прикладные программисты и конечные пользователи.

Администраторы данных и администраторы баз данных. Обычно управление данными предусматривает управление и контроль за СУБД и помещенными в нее данными. Администратор данных, или АД (Data Administrator - DA), отвечает за управление данными, включая планирование базы данных, разработку и сопровождение стандартов, правил использования данных, а также за концептуальное и логическое проектирование базы данных.

При этом концептуальное проектирование базы данных выполняется независимо от таких деталей ее воплощения, как конкретная целевая СУБД, приложения, языки программирования или любые другие физические характеристики. Логическое же проектирование проводится с учетом особенностей выбранной модели данных: реляционной, сетевой, иерархической или объектно-ориентированной. АД консультирует и дает свои рекомендации руководству высшего звена, контролируя соответствие общего направления развития базы данных установленным корпоративным целям.

Администратор базы данных, или АБД (Database Administrator - DBA), отвечает за физическую реализацию базы данных, включая физическое проектирование и воплощение проекта, за обеспечение безопасности и целостности данных, за сопровождение операционной системы, а также за обеспечение максимальной производительности приложений. По сравнению с АД, обязанности АБД носят более технический характер, он должен быть профессиональным специалистом в области информационных технологий.

Прикладные программисты. Сразу после создания базы данных следует приступить к разработке приложений, предоставляющих пользователям необходимые им функциональные возможности. Именно эту работу и выполняют прикладные программисты. Обычно прикладные программисты работают на основе спецификаций, созданных системными аналитиками. Как правило, каждая программа содержит некоторые операторы, требующие от СУБД выполнения определенных действий с базой данных - например, таких как извлечение, вставка, обновление или удаление данных. Как уже упоминалось в предыдущем разделе, эти программы могут создаваться на различных языках программирования третьего или четвертого поколения.

Пользователи. Пользователи являются клиентами базы данных - она проектируется, создается и поддерживается для того, чтобы обслуживать их информационные потребности. Пользователей можно классифицировать по способу использования ими системы.

- Конечные пользователи обычно обращаются к БД с помощью специальных приложений, позволяющих в максимальной степени упростить выполняемые ими операции. Такие пользователи инициируют выполнение операций базы данных, вводя простейшие команды или выбирая команды меню.
- Опытные пользователи для выполнения требуемых операций могут использовать такой язык запросов высокого уровня, как SQL. А некоторые опытные пользователи могут даже создавать собственные прикладные программы.

2.8 Использование класса «Динамический список»

Для начала, вспомним (или впервые узнаем)), что динамический список представляет из себя некоторое количество компонентов (узлов), содержащих непосредственно информационную часть (число, строка или более сложные типы данных), а также ссылку на следующий компонент (возможно, что узел содержит 2 ссылки: на следующий и на предыдущий, в таком случае, список называется двусвязным). Таким образом, получаем следующую структуру, описывающую компонент списка для нашей задачи:

```
struct comp {
        char name[20]; // Имя переменной
        char value[10]; // Значение переменной
        comp* next; //Ссылка на следущий элемент списка
};
```

Сам же список представляет из себя совокупность его узлов и задается обычно одной или двумя ссылками, на первый элемент и последний. Нередко, хватает и одной из этих ссылок, но в нашем случае удобнее будет использовать обе. Итак, получаем структуру, описывающую наш список:

```
struct dyn_list {
        comp* head; // Первый элемент (голова) списка
        comp* tail; // Последний элемент (хвост) списка
    };
```

Несложно понять, что значит создать пустой список. Фактически делать ничего не нужно, разве что присвоить ссылке на первый элемент списка (head) значение NULL. Оформим это как функцию, параллельно создав еще одну, проверяющую по этому условию, пуст ли список.

```
// Создание пустого списка
void constr_list(dyn_list &l)
{
    l.head = NULL;
}
// Проверка списка на пустоту
```

```
bool chk_empty(dyn_list l)
{
	return (l.head==NULL);
}
```

Видим, что все просто. Конечно, тип, описывающий список можно было оформи как класс, а функцию, создающую его сделать конструктором данного класса, но я оставлю это вам в качестве домашнего задания :. Теперь в функции main() описываем переменную типа dyn_list и создаем пустой список. Затем переходим к следующему пункту.

```
 dyn_list vars; // Динамический список
constr_list(vars);
```

Операции с компонентами списка

Ну, перед тем, как проводить какие-то операции с компонентами, их необходимо добавить в список, чем мы сейчас и займемся. Прежде всего, нам необходимо создать новый компонент и заполнить его информационную часть. Так как мы будем помещать его в конец списка, то ссылочная часть узла будет иметь значение NULL. Теперь давайте поймем, что должно происходить со ссылками head и tail при этой операции. Здесь рассматриваются 2 случая - наш список пуст или в нем есть хотя бы один компонент. Если пуст, то и head и tail будут указывать на только что созданный компонент. Если же узлы в списке присутствуют, очевидно, что сначала нужно последний узел связать с добавляемым (для этого ссылочной части компонента, на который указывает tail, присвоить адрес того, который хотим добавить), а затем "передвинуть" tail. Вот как просто все это выглядит на C++:

```
 // Включение в список нового компонента
void comp_in(dyn_list &l, char* n, char* v)
{
	comp* c = new comp();
	strcpy_s(c->name, 20, n);
	strcpy_s(c->value, 10, v);
	c->next = NULL;
	if (chk_empty(l))
		l.head = c;
	else
		l.tail->next = c;
	l.tail = c;
}
```

Теперь займемся поиском компонента. Искать будет по имени переменной, по желанию вы можете искать и по значению. В качестве аргументов, функции поиска передаем сам список и искомый текст. Возвращать наша функция будет адрес найденного узла или NULL, если ничего не найдено. Искать начнем с компонента, на который указывает head

и, двигаясь вперед, сравнивать имя текущей переменной с искомым. В функции поиска мы можем не боясь, передвигать ссылку head, так как передаем ее не по ссылке.

```
// Поиск компонента в списке по имени
comp* search(dyn_list l, char *n)
{
	while (l.head != NULL)
	{
		if (!strcmp(l.head->name,n))
			return l.head;
		l.head = l.head->next;
	}
	return l.head;
}
```

А сейчас удалим компонент. В качестве аргумента, передаем по ссылке список, а также ссылку на компонент, который собираемся удалить. В самой же функции рассматриваем 2 случая. Первый случай простой: если компонент является первым (то есть на него указывает head), то достаточно лишь переместить ссылку на первый элемент вперед. В противном случае, нам понадобится рабочая переменная-узел, которую мы будем использовать для движения по списку. Двигаться будем до тех пор, пока следующий за текущим узел не будет тем, который мы собираемся удалить. Ну а после этого, "перепрыгиваем" удаляемый компонент, присваивая ссылочной части предшествующего удаляемому компоненту адрес следующего за удаляемым. Все эти слова умещаются буквально в несколько строк исходного кода:

```
// Удаление компонента из списка
void comp_del(dyn_list &l, comp* c)
{
	if (c == l.head)
	{
		l.head = c->next;
		return;
	}
	comp* r = new comp();
	r = l.head;
	while (r->next != c)
		r = r->next;
	r->next = c->next;
	delete c;
```

```
}
```

Последняя и самая простая операция - это изменение значения информационной части узла. Менять будем поле value. В качестве параметров, по ссылке передадим адрес компонента, а также новое значение изменяемого поля. Ну и в одну строчку все изменим. Вот как просто это выглядит:

```
 // Изменение значения компонента
void comp_edit(comp &c, char* v)
{
	strcpy_s(c.value, 10, v);
}
```

С операциями закончили, теперь осталось дописать программу.

Тестируем функции и завершаем программу

Ну, тут все просто, в цикле нам осталось лишь обратиться к функции добавления компонента в список. После цикла, тестируем остальные наши функции и выводим весь список в файл. Привожу здесь весь код функции main()

```
 int main()
{
	char* fileName = new char[50];
	char* buf_name = new char[20];
	char* buf_value = new char [10];
	dyn_list vars; // Динамический список
	cout << "Enter name of file -> ";
	cin >> fileName;
	ifstream* inp = new ifstream(fileName);
	if (!inp->good())
	{
		cout << "File opening error!\n";
		system("PAUSE");
		return 0;
	}
	constr_list(vars);
	while (!inp->eof())
	{
		inp->getline(buf_name, 20, ' ');
		inp->getline(buf_value, 10, ' ');
```

```
                comp_in(vars, buf_name, buf_value);
        }
        inp->close();
        comp* p = new comp();
        p = search(vars, "z");
        if (p)
                comp_del(vars, p);
        p = search(vars, "abc");
        if (p)
                comp_edit(*p, "2");
        ofstream* out = new ofstream(fileName);
        while (vars.head != NULL)
        {
                out->write(vars.head->name, strlen(vars.head->name));
                out->write(" ",1);
                out->write(vars.head->value, strlen(vars.head->value));
                out->write(" ",1);
                vars.head = vars.head->next;
        }
        out->close();
        system("PAUSE");
        return 0;
}
```

2.9 Наследование на основе списка классов стека и очереди

Стек является общей структурой данных для представления данных, которые должны обрабатываться в определенном порядке. Например, когда функция вызывает другую функцию, которая, в свою очередь, вызывает третью функцию, важно, чтобы третья функция вернулась на вторую функцию, а не первую.

Один из способов реализации такого порядка обработки данных — это организовать своего рода очередь вызовов функций. Последняя добавленная в стек функция, будет завершена первой и наоборот, первая добавленная в стек функция будет завершена последней. Таким образом, сама структура данных обеспечивает надлежащий порядок вызовов.

Концептуально, структура данных — стек очень проста: она позволяет добавлять или удалять элементы в определенном порядке. Каждый раз, когда добавляется элемент, он

попадает на вершину стека, единственный элемент, который может быть удален из стека — элемент, который находится на вершине стека. Таким образом, стек, как принято говорить, «первым пришел, последним ушел — FILO» или «последним пришел, первым ушел — LIFO». Первый элемент, добавленный в стек будет удален из него в последнюю очередь.

Так в чем же дело? Зачем нам нужны стеки? Как мы уже говорили, стеки — удобный способ организации вызовов функций. В самом деле, «стек вызовов» это термин, который часто используют для обозначения списка функций, которые сейчас либо выполняются, либо находятся в режиме ожидания возвращаемого значения других функций.

В некотором смысле, стеки являются частью фундаментального языка информатики. Когда вы хотите реализовать очередь типа — «первый пришел, последним ушел», то имеет смысл говорить о стеках с использованием общей терминологии. Кроме того, такие очереди участвуют во многих процессах, начиная от теоретических компьютерных наук, например функцииpush-down и многое другое.

Стеки имеют некоторые ассоциируемые методы:

- **Push** — добавить элемент в стек;
- **Pop** — удалить элемент из стека;
- **Peek** — просмотреть элементы стека;
- **LIFO** — поведение стека,
- **FILO** Equivalent to LIFO

Давайте теперь разработаем программу, которая будет реализовывать работу структуры данных Стек. Создадим универсальный стек, который будет соответствовать методике обобщенного программирования, то есть создадим шаблон класса Stack. Если вы плохо знакомы с шаблонами, то прочитайте статью: Шаблоны функций, там подробно описан механизм работы с шаблонами. Если же вы знакомы с шаблонами, но забыли как работать с шаблонами классов, прочитайте статью ошаблонах классов.

Этот стек был реализован с шаблонами, чтобы его можно было использовать практически для любых типов данных. Причем размер стека определяется динамически, во время выполнения программы. В стек добавлена также дополнительная функция:peek(), которая показывает n-й элемент от вершины стека.

Шаблон класса Stack реализован в отдельном *.h файле, да, именно реализован, я не ошибся. Все дело в том, что и интерфейс шаблона класса и реализация должны находиться в одном файле, иначе вы увидите список ошибок похожего содержания:

ошибка undefined reference to «метод шаблона класса»

Интерфейс шаблона класса объявлен с 9 по 28 строки. Все методы класса содержат комментарии и, на мой взгляд, описывать их работу отдельно не имеет смысла. Обратите внимание на то, что все методы шаблона класса Стек объявлены как inlineфункции. Это сделано для того, чтобы ускорить работу класса. Так как встроенные функции класса работают быстрее, чем внешние.

Сразу после интерфейса шаблона идет реализация методов класса Стек, строки 32 — 117. В реализации методов класса ничего сложного нет, если знать как устроен стек, шаблоны и классы. Заметьте, в классе есть два конструктора, первый объявлен в строках 32-33, — это конструктор по умолчанию. А вот конструктор в строках 41-5, — это конструктор

копирования. Он нужен для того, чтобы скопировать один объект в другой. Метод Peek, строки 80 — 88 предоставляет возможность просматривать элементы стека. Необходимо просто ввести номер элемента, отсчет идет от вершины стека. Остальные функции являются служебными, то есть предназначены для использования внутри класса, конечно же кроме функции printStack(), она вывод элементы стека на экран.

Теперь посмотрим на драйвер для нашего стека, под драйвером я подразумеваю программу в которой тестируется работа класса. Как всегда это main функция, в которой мы и будем тестировать наш шаблон класса Stack.