Министерство образования и науки РФ
Федеральное государственное бюджетное образовательное учреждение
высшего профессионального образования
ТОМСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ СИСТЕМ
УПРАВЛЕНИЯ И РАДИОЭЛЕКТРОНИКИ (ТУСУР)

Кафедра ЭМИС

Касимов В.З.

ОБЪЕКТНО-ОРИЕНТИРОВАННОЕ ПРОГРАММИРОВАНИЕ

Методические рекомендации по выполнению лабораторных работ

Учебно-методическое пособие
для студентов направления 230200
«Информационные системы»

2011

Касимов В.З.
Объектно-ориентированное программирование. Лабораторные работы.
Учебно-методическое пособие. – Томск: ТУСУР, 2011. – 28 с.

Учеьно-методическое пособие посвящено основам практического применения языка C++ как языка, исповедующего объектно-ориентированный стиль программирования.

Пособие рассчитано на студентов вузов.

Введение

Этот курс лабораторных работ предназначен для практического изучения основ применения языка C++ как языка программирования, исповедующего стиль объектно-ориентированного программирования. В методических указаниях основное внимание уделяется принципиальным моментам, которые необходимы для их успешного выполнения, а также трудным для понимания концепциям языка.

Предполагается, что обучающийся владеет навыками программирования на языке C.

Лабораторная работа №1

Изучение интегрированной среды разработки Visual Studio. Примеры простых программ. Средства отладки, включенные в среду разработки.

Интегрированная среда разработки MS Visual Studio включает различные средства разработки приложений (редактор исходного кода, редактор ресурсов, компилятор, линкер, отладчик), объединенные в рамках единого Windows-приложения. Разработка приложения происходит по следующей схеме:

1. организация проекта;
2. ввод и редактирование исходного кода программы;
3. компиляция и компоновка;
4. тестирование и отладка.

Создание проекта является обязательным шагом при разработке программы. В нем указывается, какие файлы исходного текста составляют проект, с какими параметрами будут вызваны компилятор и компоновщик при создании программы. На диске состояние проекта сохраняется в файле с расширением *dsp* (для версии Visual Studio 6), и в файлах с расширением *vcproj* (для версий Visual Studio выше 6). Для того, чтобы в рамках запуска одного экземпляра среды разработки был возможен доступ к нескольким

проектам, один из которых является активным, проекты ставятся под управление рабочего пространства (*workspace* для версии Visual Studio 6, файл с расширением *dsw*) или решения (*solution* для версии Visual Studio выше 6, файл с расширением *sln*). Файлы описания проектов и рабочего пространства (решения) разных версий несовместимы друг с другом, однако Visual Studio более высокой версии имеет в своем составе средства конвертации файлов более низкой версии, обратная операция невозможна. Таким образом, для переноса разрабатываемой программы с одного компьютера на другой необходимо скопировать файлы с расширением *dsp* (или *vcproj*), *dsw* (или *sln*), а также файлы исходного текста программы: заголовочные файлы (расширение *h*), файлы исходного текста на C++ (расширение *cpp*), а также файлы описания используемых программой ресурсов (расширение *rc*). В дальнейшем изложение ведется для версии Visual Studio 2008.

Создание проекта производится последовательным выбором из меню команд: Файл>Создать>Проект. В появившемся диалоговом окне необходимо выбрать расположение базовой директории создаваемого проекта, имя проекта, тип проекта (Visual C++>Win32) и шаблон создаваемого проекта. Для целей выполнения лабораторных работ следует выбирать шаблон проекта *Консольное приложение Win32*. После нажатия на Enter на вкладке *Параметры приложения* необходимо выбрать *Предварительно скомпилированный заголовок* и установить флажок *MFC*. В результате в базовой директории создаваемого проекта будет создана поддиректория с именем проекта, в которую будут помещены файлы описания решения, проекта и сгенерируемые средой файлы исходного текста программы.

Ввод и редактирование исходного кода программы производится с помощью текстового редактора, переключение между файлами исходного текста достигается посредством окна *Обозреватель решений*, расположенного слева от окна редактора. С помощью этого же окна можно управлять содержимым решения и проекта (вставлять новые или существующие файлы, удалять их), запускать проект на построение и другие операции.

По умолчанию создаются две конфигурации проекта: *Debug* – для производства отладки и *Release* – окончательная версия, содержащая оптимизированные по скорости выполнения программы опции создания программы. Для целей выполнения лабораторных работ достаточно построения проекта в конфигурации *Debug*. Построение проекта достигается посредством выбора в контекстном меню окна *Обозреватель решений* для нужного проекта пункта *Построение*.

Под построением проекта в данном случае понимается последовательное выполнение шагов компиляции и компоновки.

Под компиляцией понимается преобразование исходного текста программы в формат объектного модуля, который записывается в файл с расширением *obj* (для исходного текста программы на C++) или *res* (для текста файла ресурса), расположенным в поддиректории проекта с именем конфигурации (*Debug* или *Release*). Применительно к тексту программы на C++ следует заметить, что компилируются только файлы с расширением *cpp* (причем независимо один от другого), заголовочные файлы с расширением *h* включаются в файлы с расширением *cpp* явно или неявно посредством директивы препроцессора *#include*. В процессе компиляции возможно выявление ошибок в исходном тексте программы, на которые указывают сообщения в окне *Вывод* нижней части экрана. Данные ошибки являются синтаксическими и связаны с неправильным использованием конструкций языка C++, для продолжения процесса построения программы они должны быть устранены.

После завершения компиляции всех файлов исходного текста начинается этап компоновки, т.е. связывания всех объектных модулей в единый исполняемый модуль (*exe*-файл). На этом этапе имеющаяся в каждом объектном модуле информация об именах, имеющих *внешний атрибут компоновки*, используется, чтобы однозначно их найти и связаться с ними в других объектных модулях, в том числе находящихся в библиотеках объектных модулей, расположенных в файлах с расширением *lib*. Данные библиотеки либо

являются стандартными, информация о которых помещается в объектный модуль компилятором автоматически, либо должны быть указаны явно в свойствах проекта, вызываемом в контекстном меню. При наличии ошибок на стадии компоновки выполняемый файл не создается, и требуется их устранение.

Следует отметить, что деление процесса построения программы на стадии (в данном случае – компиляции и компоновки, в других случаях стадий может быть больше) означает, что на каждой стадии есть входная и выходная информация, размещаемая в файлах. Утилита обслуживания процесса построения (nmake.exe) при определении необходимости выполнения какого-либо элементарного шага стадии руководствуется отметками даты-времени файлов входной и выходной информации, если выходная информация отсутствует, либо входная информация более свежая, то данный шаг выполняется. Применение такой технологии позволяет сэкономить время за счет отказа от выполнения ненужных элементарных шагов.

Устранение ошибок компиляции и компоновки не означает, что построенная программа работает правильно, практически всегда требуется ее отладка и тестирование. Запуск построенной программы производится путем нажатия на Ctrl-F5, а запуск под управлением отладчика – на F5.

Прежде, чем сделать обзор средств отладки, небольшие комментарии по поводу файлов проекта, сгенерированных средой разработки. Здесь мы не будем касаться файлов, связанных с ресурсами проекта. Пусть созданный проект называется *first*. Тогда будут сгенерированы:

1. targetver.h определяет макросы, содержащие минимальные требования к версии платформы, необходимые функции для выполнения приложения. Директива включения данного файла стоит первой в файле stdafx.h.

2. stdafx.h содержит директивы включения заголовочных файлов программных компонент Windows, необходимых для функционирования приложения. Сгенерированный файл соответствует некоторому «стандартному набору». Для подключения какой-либо новой возможности следует включить

соответствующий заголовочный файл. Директива #include «stdafx.h» должна присутствовать во всех *cpp*-файлах. Наличие в проекте данного файла делает возможным использование т.н. технологии *предкомпилированных заголовков*, когда заголовочные файлы, связанные с программными компонентами Windows, имеют каждый объем по несколько тысяч строк. Прямое включение этих заголовочных файлов в *cpp*-файлы проекта, которые требуют перекомпиляции в процессе создания программы чревато большими накладными расходами. Поэтому включение их в данный файл уменьшает эти накладные расходы путем предварительной компиляции и помещения его результата в файл с расширением *pch*. В результате включения #include «stdafx.h» в исходные *cpp*-тексты, отличные от stdafx.cpp, подключается только информация из *pch*-файла, который формально является результатом компиляции файла stdafx.cpp, содержащего единственную строку #include «stdafx.h». Таким образом, обновление *pch*-файла производится только в двух случаях: при его отсутствии или при редактировании stdafx.h.

3. stdafx.cpp

4. first.h содержит объявления типов и прототипы функций, используемых в first.cpp.

5. first.cpp содержит заготовку стартового кода приложения, которую желательно отредактировать в виде:

```
#include "stdafx.h"
#include "1.h"
#include <conio.h>

#ifdef _DEBUG
#define new DEBUG_NEW
#endif

CWinApp theApp;

using namespace std;

int _tmain(int argc, TCHAR* argv[], TCHAR* envp[]) {
int nRetCode = 0;
if (!AfxWinInit(::GetModuleHandle(NULL), NULL, ::GetCommandLine(), 0)) {
      _tprintf(_T("Критическая ошибка: сбой при инициализации MFC\n"));
      return 1;
}

// здесь помещается существенный для создаваемого приложения код
```

```
getch();    // не дает программе сразу завершиться в отладочном режиме
return nRetCode;
}
```

Обзор средств отладки

Прежде всего, простейшим средством отладки является выдача диагностических сообщений, например, функцией printf, которая осуществляет форматированный вывод на консоль. Данное средство имеет 2 недостатка: ограниченный размер буфера консольного окна и невозможность автоматического отключения для конфигурации Release. Этих недостатков лишен макрос TRACE, имеющий те же параметры, что и функция printf, вывод при этом производится в окно Вывод.

Другим средством являются средство интерактивной отладки. В текстовом редакторе путем нажатия на F9 могут быть отмечены те точки программы, при достижении которых происходит останов процесса выполнениия (точки прерывания). Снятие точки прерывания также производится нажатием на F9. В меню Отладка возможно также снять все точки прерываний или временно включить/отключить их. После остановки на точке прерывания может быть исследован контекст точки прерывания (значения локальных, видимых объектов и т.д.), сменен контекст прерывания путем перемещения по стеку вызовов функций (данное средство ценно тогда, когда происходит прерывание по обнаружению ошибочной ситуации не в тексте разрабатываемой программы, а, например, в *Run-Time Library*). Перевод в режим прерывания возможен также программно путем помещения в текст программы вызова функции AfxDebugBreak(). После остановки отладчика на точке прерывания возможен пошаговый режим отладки (F10), вход (F11) и выход (Shift-F11) внутрь функции, переход на дальнейшее выполнение до достижения точки прерывания (F5) или до прерывания по обнаружению ошибочной ситуации. Окончание процесса отладки с завершением выполнения программы производится командой Shift-F5.

Изменения языка C++ по сравнению с C, не связанные с технологией объектно-ориентированного программирования

Язык программирования C++ был разработан на основе языка C, при этом в него были внесены изменения, целью которых было повышение удобства использования и надежности. Здесь дается краткий обзор данных усовершенствований, а также даются некоторые определения терминов и понятий, в том числе и пересекающимися с понятиями языка C, с тем, чтобы естественным образом рассматривать язык C как подмножество C++. При вдумчивом использовании обучающимся использование этих определений позволит ему прояснить для себе многие сложные моменты C++.

Понятие объекта и типа

Объектом называется размещенная в оперативной памяти компьютера информация, занимающая непрерывный участок памяти и обладающая двумя свойствами: типом и классом памяти.

Типом называется свойство объекта, определяющая две его основополагающие стороны: формат хранения информации в памяти и набор допустимых операций.

Класс памяти определяет место хранения объекта в памяти и время жизни объекта.

Существуют 3 класса памяти:

1. Статический.

Объекты этого класса памяти располагаются в сегменте статических данных и существуют во все время выполнения программы. Хотя последнее фактически неверно, но можно утверждать, что по крайней мере до передачи управления в функцию main и сразу после возврата из нее, это так.

2. Автоматический.

Объекты этого класса памяти располагаются в стеке – участке оперативной памяти, работа с которым производится по принципу LIFO (последний

вошел, первый вышел). Автоматический класс памяти имеют объекты, определенные в блоке, и для которых явно не указан статический класс памяти. После выхода из блока в силу специфики работы стека, объекты автоматически уничтожаются путем изъятия памяти. «Автоматическое» уничтожение справедливо только для типов данных языка С, поскольку в С++ компилятор до изъятия памяти генерирует вызов специального метода – деструктора (но об этом далее, в соответствующем разделе).

3. Динамический.

Объекты этого класса памяти располагаются в так называемой «куче», которая организуется в назанятом сегментами кода, статических данных и стека участке оперативной памяти компьютера, отведенной программе. Создание и уничтожение объекта динамического класса памяти производится путем явного вызова соответствующих конструкций языка: *new* и *delete*. Следует заметить, что таким образом созданный объект не имеет явного имени и для того, чтобы не потерять над ним управление, требуется предпринимать определенные усилия, например, сохранять его адрес в другом объекте, например, указателе.

Имена

Имя – не являющаяся служебным словом последовательность букв и цифр, начинающаяся с буквы (‘_’ – буква).

Имена, начинающиеся с подчерка, допустимы, но не рекомендуются, т.к. зарезервированы для использования реализацией языка. На длину имени не накладывается никаких ограничений.

Имена используются для обозначения пяти категорий языка:

1. объектов,

2. функций,

3. типов,

4. значений,

5. меток.

Все имена, кроме меток, до их использования должны быть объявлены.

Объявление может не только сопоставить с именем «что-то», но и некий элемент, идентифицируемый именем (в зависимости от контекста). Такие объявления называются определениями. Может существовать несколько непротиворечащих одно другому объявлений имени, но только одно определение.

Та часть исходного кода, в которой имя используется в связи со своим объявлением, называется областью существования.

Имеется 5 типов областей существования имени (ОСИ):

1. *блок*
2. *функция (только для меток),*
3. *прототип функции,*
4. *файл,*
5. *класс.*

Область видимости имени – подобласть ОСИ, в которой можно, используя имя, получить доступ к обозначаемому именем элементу.

Имя типа – объявление объекта такого типа, в котором опущено имя самого такого объекта.

```
int - целый
int *- указатель на целый
int *[10] – массив указателей
int (*)[5] – указатель на массив
int *(void) – функция, возвращающая указатель на int
int (*)(void) – указатель на функцию
```

Иногда в тексте программы на C++ синтаксически допустимо только простое имя типа. В этом случае для его получения требуется использовать конструкцию

```
typedef определение_объекта_типа;
```

При этом имя такого объекта становится *синонимом* имени типа.

Следует заметить, что используемые в программе на C++ имена сохраняют свой смысл не на всех стадиях построения программы. Имена, используемые для обозначения *типов, значений и меток*, используются только ком-

пилятором и информация о них не попадает в объектный модуль. Информация об именах *функций и объектов*, используемых в cpp-файле, сохраняется в объектном модуле, но только для тех из них, которые имеют *внешний атрибут компоновки имени (АКИ)*.

Правила определения АКИ

1. Для имени с ОСИ типа *файл* при наличии модификатора *static* – внутренний АКИ. Если при этом имеется объявление *extern*, то внешний АКИ. Для модификатора *const* – по умолчанию внутренний АКИ.
2. Если в объявлении локального имени присутствует *extern*, то атрибут тот же, что и глобальное имя, если оно есть. В противном случае – внешний.
3. Для функции неявно используется внешний АКИ.
4. Если для глобального объекта не указан *static*, то внешний.

Для имен с внешним АКИ на этапе компиляции производится модификация имени. Правила, по которым производится модификация, различны для языков C и C++.

В C++ возможно определение объекта *по мере необходимости*.

Инициализация – определение объекта вместе с заданием начального значения (в виде *выражения*). Имеется различие между инициализацией объекта и операцией *присваивания*. Последняя применяется для *уже созданного* объекта, в то время как инициализация может считаться частью процесса создания (конструирования) объекта.

Модификатор const

Данный модификатор означает, что объект не может быть изменен. Поэтому единственный способ задать ему значение – сделать это при инициализации. Поскольку одна из причин его появления – заменить обозначения констант в программе посредством определения макроса директивой #define на более безопасный способ, а обозначения констант именами производится, как правило, в заголовочных файлах, то для устранения коллизии – существует только одно определение, но может существовать сколько угодно не противоречащих одно другому объявлений, для объектов с модификатором

const по умолчанию используется внутренний АКИ.

Ссылки

Ссылки являются очень важным усовершенствованием языка С++.

Ссылкой называется такой константный указатель на объект, к которому при его использовании неявно применяется операция разадресации.

Этого определения достаточно, чтобы полностью разобраться с использованием ссылок.

```
int k=2; // создание объекта
int& rk=r; // инициализация ссылки
rk = rk*100; // использование ссылки
```

эквивалентный код

```
int k=2; // создание объекта
int * const pk=&r; // инициализация указателя
*pk = (*pk)*100;
```

Здесь важно следующее: сама ссылка как *константный указатель на объект* не может быть изменена, она может быть только *инициализирована* в момент создания и фактически *указывает* на объект, которым проинициализирована. Такое ее использование, как в примере выше, означает, что ссылка в rk является в некотором роде синонимом объекта k. Если бы это было единственной потребностью введения в язык С++ ссылки, то, наверное, данное усовершенствование не было бы сделано. Главное предназначение ссылки – это передача параметра в функцию по ссылке и возврат результата по ссылке. Рассмотрим первый случай: передача параметра в функцию по ссылке.

```
struct A { double m_f; … };
double fun(A& a) {
return a.m_f; // использование ссылки
}

A a1;
double f = fun(a1); // создание ссылки на a1
```

эквивалентный код

```
struct A { double m_f; … };
double fun(A * const pa) {
return (*pa).m_f;
}

A a1;
double f = fun(&a1);
```

В данном случае в функцию fun объект-формальный параметр передается по ссылке. Это означает, что инициализация ссылки производится при подготовке к вызову функции объектом-фактическим параметром, а фактически – его адресом. По-существу происходит так, как мы делали, если на языке С нам нужно было сделать некий параметр функции выходным: передавали этот параметр не по значению, а по значению его адреса, т.е. указатель, а внутри функции использовали операцию разадресации указателя. В

данном случае применение ссылки в качестве параметра функции позволяет эту разадресацию не делать, а только подразумевать.

Самый большой эффект от использования ссылки имеет место при возврате результата выполнения функции по ссылке. Рассмотрим следующий пример.

```
struct A { double m_f; … };
A& fun() {
A a;
return a; // возврат ссылки на a
}

// возврат результата по ссылке
A a1 = fun();
// использование ссылки производится в момент
// присваивания результата объекту a1
```

эквивалентный код

```
struct A { double m_f; … };
A * const fun() {
A a;
return &a; // возврат адреса a
}

// фактическая реализация
A a1 = *(fun());
```

В данном примере производится возврат из функции fun по ссылке. Фактически возвращается адрес объекта, созданного локально в теле функции fun и на момент использования ссылки фактически не существующего. Данное использование возврата по ссылке является неверным, поскольку должна возвращаться ссылка на объект, существующий на момент ее *использования*, когда *неявно применяется операция разадресации*. Таким образом, не принимая во внимание возможности, предоставляемые использованием объектно-ориентированного подхода, правильным возврат объекта по ссылке будет только в трех случаях:

1. Возврат объекта, имеющего статический класс памяти.
2. Возврат объекта, имеющего динамический класс памяти.
3. Возврат объекта, передаваемого в функцию по ссылке.

По очевидным причинам, последний случай самый интересный, поскольку позволяет производить вложенный вызов:

```
struct A { double m_f; … };
A& fun(A& a) {
// что-то делаем, в том числе и модифицируем объект a
return a; // возврат ссылки на a
}

A a1, a2;
a2 = fun(fun(a1));
```

```
double f = fun(a1).m_f;
```

или что-то более экзотическое:

```
double f = fun(a1).m_f;
```

Перегрузка имен функций

Перегрузка функций обеспечивает возможность использовать несколько функций с одним именем. Этим средством можно воспользоваться для разработки семейства функций, выполняющих близкие задачи, но с различными списками аргументов.

Если 2 функции имеют одно и то же количество аргументов с одинаковыми типами и порядок следования их одинаков, то говорят, что функции имеют одну и ту же сигнатуру. Язык C++ предоставляет возможность определить несколько функций с одним именем при условии, что эти функции имеют разные сигнатуры. При вызове функции компилятор генерирует вызов той функции, сигнатура которой соответствует фактическим параметрам, при этом может производиться стандартное приведение типов. В такой ситуации может возникнуть неоднозначность выбора нужной версии функции. В этом случае вызов функции диагностируется как ошибочный.

Существуют также несовместимые сигнатуры, например версии функции, передающие один и тот же параметр по значению или по ссылке. Также в ходе сопоставления необходимо учитывать различие между параметрами с const и без него. Несовместимы также объявления с одинаковыми именами и сигнатурами, но с различными возвращаемыми значениями функций.

Аргументы, принимаемые по умолчанию

Это значения, используемые автоматически, если соответствующие фактические параметры в обращении к функции опущены.

Существуют правила их использования:

1. Задание аргументов функции по умолчанию производится только в прототипе.

2. Аргументы функции по умолчанию должны быть последними в спис-

ке аргументов.

Обработка исключений

Основная идея обработки исключительных ситуаций заключается в следующем. Если в текущем контексте программы (например, внутри функции f) не хватает информации для принятия решения о том, что делать при возникновении ошибки, то информация об ошибке передается во внешний по отношению к текущему контекст, т.е. туда, откуда был вызов f . При этом создается объект с информацией, описывающий ошибку, и он «выбрасывается» во внешний контекст, причем все созданные до этого момента локальные объекты уничтожаются. Генерация исключительной ситуации производится директивой

```
throw Object(Список параметров)
```

Во внешнем контексте (или если есть желание обработать ее в функции f), вызов f (или фрагмент текущего контекста) помещается в специальный блок try:

```
try {
// фрагмент текущего или внешнего контекста
}
```

Обработка исключительной ситуации производится в обработчике исключений, который представляет собой последовательность одного или нескольких блоков catch, непосредственно следующих за блоком try:

```
catch(type obj) {
}
```

Управление передается в тот блок catch, параметр type которого соответствует типу объекта в throw, при этом последний объект становится доступным в блоке catch под именем obj. Если хочется перехватить все исключения, то вместо `type obj` следует написать 3 точки, причем данный блок catch должен быть последним в последовательности блоков. Далее выполняются все действия, приведенные внутри блока catch. После этого управление

передается в точку непосредственно за последовательностью блоков catch. Если это неприемлемо, то можно перезапустить исключение, сгенерировав новую исключительную ситуацию или повторив обработанную, и отправив его на обработку в более внешний по отношению к текущему контекст. Повтор только что обработанного исключения производится вызовом throw без параметра.

Лабораторная работа №2
Использование объектов типа «указатель на функцию» для обеспечения полиморфного поведения программы (8 часов).

Цель работы: получение навыков работы по использованию технологии динамического связывания, т.е. принятия решения по выбору нужной функции для вызова во время выполнения программы.

Динамическое связывание. Компилятор превращает вызов функции в команду процессора, в которой присутствует адрес этой функции. Если же функция определена в другом файле исходного текста, то это же самое делает компоновщик на этапе сборки программы. Это называется **статическим связыванием** в том смысле, что в момент загрузки программы все связи между вызовами функций и самими функциями установлены. Динамическим связыванием называется связывание вызова функции с ее адресом во время работы программы. Си позволяет работать с архитектурной первоосновой динамического связывания - указателем на функцию.

Указатель на функцию - переменная, которая содержит адрес некоторой функции. Соответственно, косвенное обращение по этому указателю представляет собой вызов функции.

Следует обратить внимание на то, что в определении указателя присутствует прототип - указатель ссылается не на произвольную функцию, а только с заданной сигнатурой и типом возврата. Перед началом работы с указате-

лем его необходимо назначить на соответствующий объект, в данном случае - на функцию.

Указатель на функцию может рассматриваться как **средство параметризации алгоритма**. Если обычный указатель позволяет параметризировать алгоритм обработки данных, то указатель на функцию параметризирует сам алгоритм. То есть некоторая его часть может быть заранее не определена и будет подключаться к основному алгоритму только в момент его выполнения (**динамическое связывание**).

Задание. Написать программу для решения уравнения $f(x)=c$ на заданном промежутке $[a,b]$с точностью 0.0001 методом деления отрезка пополам. Значения *a*, *b* и *c*, а также ввод имени функции из некоторого фиксированного списка (sin, cos, tan, exp, atan, sqrt, log) задаются в диалоговом режиме и отображаются на экране.

Лабораторная работа №3

Разработка простейшего пользовательского типа данных (8 час).

Цель работы: получение навыков работы по созданию абстрактных типов данных.

При создании в программе объектов абстрактных типов всегда вызывается специальный метод, называемый конструктором. Конструктор также вызывается при создании объекта по уже сконструированному объекту (конструктор копирования). Если последний в классе не определен, то компилятор обеспечивает его автоматически, который работает как покомпонентное копирование. То же можно сказать по поводу оператора присваивания. Если в составе класса отсутствуют динамические данные, то эти методы по умолчанию вполне приемлемы в большинстве случаев. В данной лабораторной работе мы имеем как раз такую ситуацию.

В заданиях по лабораторной работе требуется написать нужные конструкторы, а также методы, обеспечивающие изменение отдельных составных частей объекта) и вывод его содержимого.

Задание 3.1. Дата, представленная целыми переменными: год, месяц и день.

Задание 3.2. Время, представленное целыми переменными: час, минута, секунда.

Задание 3.3. Несократимая дробь, представленная двумя длинными целыми: числитель и знаменатель.

ЛАБОРАТОРНАЯ РАБОТА №4. Выполнение индивидуального задания на тему «Создание класса с перегрузкой некоторых операций» (12 часов).

Цель работы: получение навыков работы по перегрузке операций с целью получения более удобного способа работы с объектами абстрактных типов данных.

Все нижеперечисленные задания базируются на использовании *динамических структур данных (ДСД)*. Этим термином обозначается способ организации данных, при котором не существует принципиальных ограничений на число одновременно хранимых в оперативной памяти экземпляров (элементов) данных. При этом хранимые элементы конструируются в динамическом классе памяти, а для организации самой структуры данных используются указатели на элементы. Примерами таких ДСД являются *линейные списки*, *стеки*, *очереди* и *бинарные деревья*. Отличие между ними заключается способами связи между элементами ДСД и допустимыми операциями. В настоящем задании проще всего использовать *линейный список*. Эта структура данных организуется таким образом, что каждый ее элемент (пусть это будет экземпляр класса *N*) содержит указатель на следующий элемент, при этом ес-

ли последний равен 0, то он является на самом деле последним элементом списка. Структура данных (класс с именем *L*), управляющий списком, содержит в своем составе компоненту типа *N *pFirst*, указывающий на первый элемент списка (равен 0, если в списке нет элементов).

Для всех заданий нужно определить конструктор, параметром которого является текстовая строка, означающая имя файла на диске, содержащего в текстовом формате информацию по всем входящим в состав управляющей структуры данных элементов. При уничтожении управляющей структуры ее содержимое также должно быть сохранено в файле на диске.

Должны быть перегружены операции + и – с правым операндом типа *N** для добавления и удаления соответствующего элемента из списка. Рекомендуется также перегрузить операцию индексации для поиска требуемого элемента по *ключу*. *Ключ* – это уникальное значение, однозначно идентифицирующее элемент списка, выбор поля структуры для ключа зависит от конкретного задания. В классе должны быть средства для выборки из управляющей структуры отсортированных элементов.

Все задания должны иметь меню для управления операциями и демонстрации методов класса.

Задание 4.1. Смоделировать понятие «студенческая группа».

Задание 4.2. Смоделировать понятие «записная книжка».

Задание 4.3. Смоделировать понятие «домашняя библиотека».

Задание 4.4. Смоделировать процесс сортировки ж/д состава на Т-образном сортировочном узле. Программа должна разделять на 2 направления состав, состоящий из вагонов двух типов (на каждое направление формируется состав из вагонов одного типа).

Задание 4.5. Смоделировать процесс наличия автобусов в автобусном парке. В начальный момент формируются данные о всех автобусах в парке. При работе программы формируются 2 списка: 1 – об автобусах, находящихся в парке, 2 – об автобусах, находящихся на маршруте. При выезде автобуса вводится его номер и программа удаляет его запись из списка автобусов,

находящихся в парке, и добавляет его в список автобусов, находящихся на маршруте. При въезде в парк производится обратная операция. По запросу выводится информация об автобусах, находящихся на маршруте и в парке с указанием времени выезда и заезда соответственно.

Задание 4.6. Задание №5, но реализация через единый список.

Задание 4.7. Смоделировать процесс заполнения диска емкостью 360К. В процессе работы файлы произвольной длины от 16К до 32К (выбираются случайным образом) записываются на диск или удаляются с него. При попытке записать файл длиной более длины непрерывного свободного участка файл не записывается и выдается сообщение. По запросу выдается информация о свободных и занятых участках.

Задание 4.8. Написать класс одномерного массива строк, каждая строка может иметь произвольную длину. Определить операции индексации, операции + и – с правым операндом типа char* для добавления и удаления строки в массив.

Задание 4.9. Определить класс многочленов произвольной длины с вещественными коэффициентами. Определить методы для вычисления значения многочлена для заданного аргумента, операции сложения, вычитания и умножения многочленов с получением нового объекта-многочлена.

Задание 4.10. Смоделировать процесс выдачи книг из библиотеки. Каждая книга характеризуется автором (одним), названием, годом издания и общим количеством экземпляров. Кроме списка книг требуется создать список читателей. Для каждого читателя и книги требуется хранить информацию о соответственно книгах, которые читатель взял, и читателях, взявших книгу. Требуется выдавать информацию: по книге - о возможности выдачи, списке читателей, взявших книгу; по читателю – о книгах у читателя.

Задание 4.11. Смоделировать базу данных клиентов бензоколонки. Каждый клиент имеет индивидуальный код, имя. База должна хранить код клиента, дату и время заправки, количество заправленного топлива. Требуется выдавать информацию о заправках как по дате, так и по клиенту.

ЛАБОРАТОРНАЯ РАБОТА №5. Создание классов для работы с динамическими данными переменной размерности (12 часов).

Цель работы: получение навыков работы по созданию типов данных, для реализации которых используются данные в динамической памяти.

Если объект содержит динамические данные или связанные с ним ресурсы, то простое копирование в конструкторе копирования и операторе присваивания не совсем корректно: оба объекта содержат указатели на динамические данные, для обоих вызываются деструкторы, при разрушении одного из них другой окажется с некорректными данными (будет ссылаться на уже освобожденные динамические переменные).

Конструктор копирования должен выполнять корректное копирование содержимого объекта-параметра в текущий объект (уместно назвать его «конструктором клонирования»). Он должен создать их копию, «независимую от оригинала». При передаче объектов в качестве формальных параметров и результата по значению (в виде копии) транслятор автоматически формирует вызов этого конструктора и тогда функция получает и возвращает корректную и независимую копию объекта.

Класс матриц произвольной размерности. Конструктор, деструктор и работа с данными объекта. Для объекта-матрицы необходимо прежде всего обеспечить ее неограниченную размерность. Сразу же заметим, что двумерные массивы имеют хотя бы одну статическую размерность (число столбцов матрицы). Поэтому резонно выбрать массив указателей на массивы - строки матрицы (неограниченная вторая размерность). Для неограниченной первой размерности массив указателей тоже должен быть динамическим. «Самодостаточный» объект включает в себя целые переменные - текущие размерности и массив указателей. Память для структуры данных резервируется в момент конструирования объекта, тогда же задаются и размерности матрицы, в дальнейшем они не должны меняться. Для обеспечения

широких возможностей задания матриц необходим набор конструкторов: конструктор, заполняющий матрицу заданным значением, значениями из линейного массива коэффициентов, конструкторы, заполняющие матрицу списком коэффициентов.

Поскольку объект содержит динамические данные, ему необходим конструктор копирования. Деструктор объекта разрушает динамические данные объекта.

Для обеспечения доступа к содержимому матрицы достаточно реализовать метод, возвращающий ссылку на выбранный коэффициент матрицы, что позволяет работать с ним как по чтению, так и по записи.

Задания.

Разработать класс, объект которого реализует «пользовательский» тип данных. Обеспечить его произвольную размерность за счет использования в объекте динамических структур данных. Разработать необходимые конструкторы, деструктор, конструктор копирования, методы, обеспечивающие изменение отдельных составных частей объекта (например, коэффициентов полинома), перегрузку некоторых операций и вывод содержимого в поток ostream.

1. Целое произвольной длины во внешней форме представления в виде строки символов-цифр. Перегрузить операции сложения и вычитания с типом long и с данным типом, оператор преобразования к типу long.

2. Целое произвольной длины в двоично-десятичной форме представления в виде: десятичная цифра - тетрада, по две тетрады в одном байте, признак отрицательного цисла – тетрада OxE. Последовательность цифр хранится, начиная с младшей, и ограничена тетрадой с кодом OxF. Перегрузить операции сложения и вычитания с типом long и с данным типом, оператор преобразования к типу long.

3. Степенной полином, представленный размерностью и динамическим массивом коэффициентов. Перегрузить операции сложения и вычитания с данным типом, оператор преобразования к типу long.

4. Степенной полином, представленный односвязным списком ненулевых коэффициентов. Элемент списка содержит показатель степени (индекс) и само значение коэффициента. Перегрузить операции сложения и вычитания с данным типом, оператор преобразования к типу long.

5. Матрица произвольной размерности, представленная размерностями и линейным динамическим массивом коэффициентов матрицы, в котором она разложена по строкам. Перегрузить операции сложения, вычитания, умножения с данным типом.

6. Матрица произвольной размерности, представленная размерностями и динамическим массивом указателей на динамические массивы — строки матрицы. Перегрузить операции сложения, вычитания, умножения с данным типом.

7. Разреженная матрица, представленная динамическим массивом структур, содержащих описания ненулевых коэффициентов: индексы местоположения коэффициента в матрице (целые) и значение коэффициента (вещественное). Перегрузить операции сложения, вычитания, умножения с данным типом.

8. Разреженная матрица, представленная списком, каждый элемент которого содержит описание ненулевого коэффициента: индексы местоположения коэффициента в матрице (целые) и значение коэффициента (вещественное). Перегрузить операции сложения, вычитания, умножения с данным типом.

9. Разреженная матрица, представленная динамическим массивом указателей на структуры, определяющие ненулевые коэффициенты. Структура содержит индексы местоположения коэффициента в матрице и само значение коэффициента. Перегрузить операции сложения, вычитания, умножения с данным типом.

ЛАБОРАТОРНАЯ РАБОТА №6. Выполнение индивидуального задания на тему «Наследование классов» (8 часов).

Цель работы: получение навыков работы по созданию иерархий типов и перегрузке виртуальных функций.

Рассмотрим группу классов - пользовательских типов данных. Допустим, проектируется структура данных, предназначенная для хранения произвольных объектов пользовательских типов данных. Прежде всего, определяется ряд общих действий, которые обязательно должны быть выполнимы в объекте любого класса, чтобы он мог включаться в структуру данных, и выделяются в абстрактный (пустой) базовый класс.

```
// Абстрактный базовый класс для пользовательских типов
данных

class ADT {
public:
virtual int Get(char ")=0;        // Загрузка объекта
из строки
virtual char *Put()=0;        // Выгрузка объекта в
строку
virtual long Append(File*)=0; // Добавить объект в файл
virtual int Load(File*)=0;        // Загрузить объект
из файла
virtual int Type()=0;     // Возвращает идентификатор
                          типа
                          // объекта
virtual char 'Name))=0;       // Возвращает имя типа
объекта
virtual int Cmp(ADT *)=0;         // Сравнивает значе-
ния объектов
```

```
virtual ADT *Copy()=0;    // Создает динамический объект
                          -копию // с себя (клонирова-
                          ние)
virtual -ADT{){ };             // Виртуальный деструктор
};
```

Как видим, базовый класс получился абстрактным, то есть его объект не содержит данных, а функции «пустые». Это значит, что объекты базового класса не могут создаваться в программе, а сам класс - это исключительно «объединяющая идея». В принципе, базовый класс может содержать данные и непустые функции, если в самой группе классов выделяется некоторая общая часть.

Естественно, что при проектировании любого производного класса в первую очередь в нем должны быть реализованы виртуальные функции, которые поддерживают в нем перечисленные действия. Остальная часть класса может быть какой угодно, естественно, что она уже не может использоваться в общих функциях работы со структурой данных.

Все эти методы конкретизируют действия, которые должны быть произведены в производном классе при доступе к его объектам через интерфейс ADT. Например, метод добавления объекта к заданному файлу вызывает в этом файле метод добавления строки, содержащейся в объекте, в виде записи переменной длины.

Базовый класс и набор виртуальных функций используются как общий интерфейс доступа к объектам - типам данных при проектировании структуры данных. Любое множество объектов, для которых осуществляются основные алгоритмы (хранение, включение, исключение, сортировка, поиск и т.д.) будет представлено как множество указателей на объекты базового класса ADT, а за ними способны «скрываться» объекты любых производных классов. Все действия, выполняемые над объектами, осуществляются уже в производных классах через перечисленные виртуальные функции.

Отдельного обсуждения заслуживают проблемы уничтожения объектов из группы производных классов, указатели на которые хранятся в структуре данных. Если предположить, что хранимые объекты динамические и при ее разрушении возможно разрушение этих объектов, то деструктор, вызываемый для объекта базового класса ADT, должен быть виртуальным.

Еще одна тема - работа с файлом, в который «вперемешку» записываются объекты, хранимые в структуре данных. Очевидно, что для идентификации объектов в файле перед каждым из них потребуется сохранять его тип (результат виртуальной функции Type), после чего вызывать виртуальную функцию сохранения объекта в заданном файле Append.

При загрузке структуры данных из файла необходимо создавать динамические объекты различных производных классов, в которые «подгружать» содержимое при помощи виртуальной функции Load. Здесь уже динамические возможности Си++ достигли своего потолка: создать произвольный объект по его идентификатору, полученному из файла, можно только явно, с использованием переключателя.

Задания.

Сделать разработанный в лабораторных работах 3 и 5 тип данных производным от класса ADT, переопределив в нем соответствующие методы. Разработать программу, демонстрирующую возможность хранения в одной структуре данных объектов различного типа.

ЛАБОРАТОРНАЯ РАБОТА №7. Создание шаблона класса для описания обобщенного массива (12 часов).

Цель работы: получение навыков работы по созданию шаблонов класса и их практическому использованию.

Задание.

Создать обобщенный класс массива для хранения элементов произвольного типа Type. Класс должен быть реализован в виде списка элементов, каждый из которых предназначен для хранения kPart элементов. Для доступа к элементам массива должен быть перегружен оператор []. При выходе индекса index за пределы должно быть одно из двух: при index<0 генерируется исключение, при превышении индекса за текущее значение верхнего предела последний увеличивается таким образом, чтобы значение index было допустимым.

Шаблон должен иметь следующий вид: template ArbArray<class Type, int kPart>