МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ РОССИЙСКОЙ ФЕДЕРАЦИИ

федеральное государственное бюджетное образовательное учреждение высшего образования

**«МОСКОВСКИЙ АВТОМОБИЛЬНО-ДОРОЖНЫЙ ГОСУДАРСТВЕННЫЙ ТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ (МАДИ)»**
ВОЛЖСКИЙ ФИЛИАЛ

Кафедра гуманитарных и естественнонаучных дисциплин

# Методические указания к лабораторным работам по дисциплине ОБЪЕКТНО-ОРИЕНТИРОВАННОЕ ПРОГРАММИРОВАНИЕ

Направление подготовки

***09.03.01 Информатика и вычислительная техника***

Направленность (профиль, специализация) образовательной программы

***«Автоматизированные системы обработки информации и управления»***

Квалификация

бакалавр

Чебоксары
2019

# СОДЕРЖАНИЕ

**ТРЕБОВАНИЯ К ВЫПОЛНЕНИЮ ЛАБОРАТОРНЫХ РАБОТ** ..... 3

ТЕМАТИКА ЛАБОРАТОРНЫХ РАБОТ ..... 5

Лабораторная работа №1. Работа в среде Microsoft Visual Studio. Основы программирования на языке С++. ..... 5
Лабораторная работа №2. Файловый ввод-вывод, операторы манипулирования битами, работа с массивами. ..... 26
Лабораторная работа №3. Модульная организация программ. Работа со структурами. Указатели и динамическая память. ..... 32
Лабораторная работа №4 . Стандартная библиотека языка Си++. ..... 37
Лабораторная работа №5. Создание собственных классов. ..... 45
Лабораторная работа №6. Перегрузка операций, умные указатели. ..... 49
Лабораторная работа №7. Модульное тестирование ПО. Разработка в стиле TDD ..... 52
Лабораторная работа №8. Композиция, наследование, полиморфизм. ..... 58
Лабораторная работа №9. Модульное тестирование ПО. Разработка в стиле TDD. ..... 61
Лабораторная работа №10. Обобщенное программирование, шаблоны. ..... 66

# ТРЕБОВАНИЯ К ВЫПОЛНЕНИЮ ЛАБОРАТОРНЫХ РАБОТ

Все лабораторные работы курса “Интернет программирование” выполняются в едином порядке, в соответствии с едиными требованиями.

**Порядок выполнения работы.**

1. Изучить "Краткие теоретические сведения".
2. Получить у преподавателя условия задач (номер варианта).
3. Решить задачи (см. п.
4. **Порядок решения задачи.**).
5. Показать работу программ преподавателю.
6. Распечатать тексты полученных программ.
7. Оформить отчет (пояснительную записку) (см. п.
8. **Требования к отчету**).
9. Сдать отчет (пояснительную записку) преподавателю.
10. Подготовиться к ответам на контрольные вопросы.
11. Защитить работу.

**Порядок решения задачи.**

1. Изучить условие задачи.
2. Разработать алгоритм решения задачи.
3. Согласовать алгоритм решения задачи с ведущим преподавателем
4. Разработать порядок работы с программой.
5. Написать и ввести программу.
6. Отладить программу.
7. Скорректировать алгоритм и порядок работы программы по результатам отладки.
8. Ответить на вопросы задания (если есть).

**Общие требования к программам.**

- Программа должна выводить на терминал реквизиты авторов (фамилию, имя и группу).
- Программа, использующая ввод с клавиатуры, должна подсказывать пользователю, что ему делать.

**Требования к отчету.**

*Требования к оформлению.*

1. Отчет выполняется на листах формата А4 с использованием любого текстового процессора и распечатывается на принтере.
2. Титульный лист отчета выполняется по стандартной форме (см. Рисунок 1).
3. Рамка на последующих листах отчета необязательна.

4. Листы отчета необходимо скрепить.
5. Отчет должен быть подписан исполнителем.
6. Тексты программ должны содержать комментарии к использованию переменных и работе программы.
7. Тексты программ распечатываются на принтере.

*Содержание отчета.*

1. Титульный лист (см. Рисунок 1).
2. Цель работы.
3. Описание решений задач (см. п. 0).
4. Тексты программ (распечатки).

*Порядок описания решения задачи.*

Решение каждой задачи описывается в следующем порядке;

1. Условие задачи.
2. Физическое и математическое решение задачи.

Министерство образования и науки Российской Федерации
Федеральное государственное бюджетное образовательное учреждение
высшего профессионального образования
«МОСКОВСКИЙ АВТОМОБИЛЬНО-ДОРОЖНЫЙ ГОСУДАРСТВЕННЫЙ
ТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ (МАДИ)»
Волжский филиал

Факультет очный
Кафедра математики и информатики
Направление подготовки Автоматизированные системы обработки информации и управления

О Т Ч Е Т

к лабораторной работе №1

**«Доступ к базам данных. СУБД MySQL»**

Дата сдачи «__»_____20__г
Зачтено «__»_____20__г

Выполнил: студент гр. ОП-14,
Петров И.В.
Преподаватель: Семенов Б.И.

Чебоксары 2015

Рисунок 1. Пример оформления титульного листа.

# ТЕМАТИКА ЛАБОРАТОРНЫХ РАБОТ

**Лабораторная работа №1. Работа в среде Microsoft Visual Studio. Основы программирования на языке С++.**

**Основные сведения**

**1. Основы работы с Visual Studio C++**

Основные способы создания программы на языке высокого уровня С++.

**Общие сведения:**

Язык С (читается как Си) в основе своей был создан в 1972 г. как язык для операционной системы UNIX [1.2]. Автором этого языка считается Денис М. Ритчи (DENNIS M. RITCHIE).

Популярность языка С обусловлена, прежде всего тем, что большинство операционных систем были написаны на языке С. Его начальное распространение было задержано из-за того, что не было удачных компиляторов.

Несколько лет не было единой политики в стандартизации языка С. В начале 1980-х г. в Американском национальном институте стандартов (ANSI) началась работа по стандартизации языка С. В 1989 г. работа комитета по языку С была ратифицирована, и в 1990 г. был издан первый официальный документ по стандарту языка С. Появился стандарт 1989, т.е. С89 [1.3].

К разработке стандарта по языку С была также привлечена Международная организация по стандартизации (ISO). Появился стандарт ISO/IEC 9899:1990, или ANSI C99 языка С [1.3].

В данном пособии за основу принимается стандарт языка С от 1989 г. и написание программ будет выполняться в среде разработки Visual Studio 2010.

Язык С является прежде всего языком высокого уровня, но в нем заложены возможности, которые позволяют программисту (пользователю) работать непосредственно с аппаратными средствами компьютера и общаться с ним на достаточно низком уровне [1.3]. Многие операции, выполняемые на языке С, сродни языку Ассемблера. Поэтому язык С часто называют языком среднего уровня.

Для написания программ в практических разделах данного учебного пособия будет использоваться компилятор языка С++, апрограммирование будет вестись в среде Microsoft Visual Studio 2010. Предполагается, что на компьютере установлена эта интегрированная среда.

Microsoft Visual Studio 2010 доступна в следующих вариантах:

Express – бесплатная среда разработки, включающая только базовый набор возможностей и библиотек.

Professional – поставка, ориентированная на профессиональное создание программного обеспечения, и командную разработку, при которой созданием программы одновременно занимаются несколько человек.

Premium – издание, включающее дополнительные инструменты для работы с исходным кодом программ и создания баз данных.

Ultimate – наиболее полное издание Visual Studio, включающие все доступные инструменты для написания, тестирования, отладки и анализа программ, а также дополнительные инструменты для работы с базами данных и проектирования архитектуры ПО.

Отличительной особенностью среды Microsoft Visual Studio 2010 является то, что она поддерживает работу с несколькими языками программирования и программными платформами. Поэтому, перед тем, как начать создание программы на языке С, необходимо выполнить несколько подготовительных шагов по созданию проекта и выбора и настройки компилятора языка С для трансляции исходного кода

После запуска Microsoft Visual Studio 2010 появляется следующая стартовая страница, которая показана на рис. 1.1.

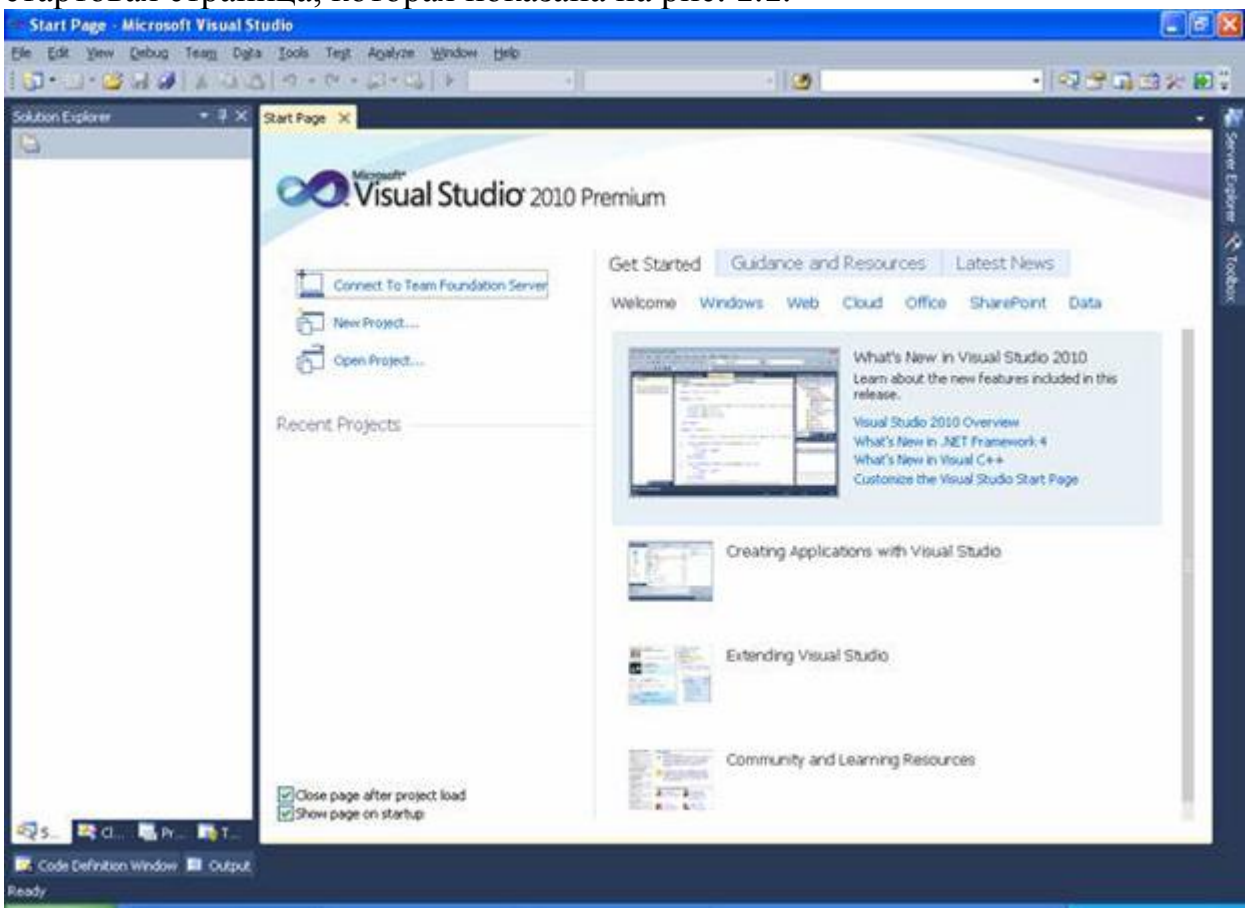


Рис. 1.1. Стартовая страница Visual Studio 2010

Следующим шагом является создание нового проекта. Для этого в меню File необходимо выбрать New – Project (или комбинацию клавишCtrl + Shift + N ). Результат выбора пунктов меню для создания нового проекта показан на рис. 1.2.

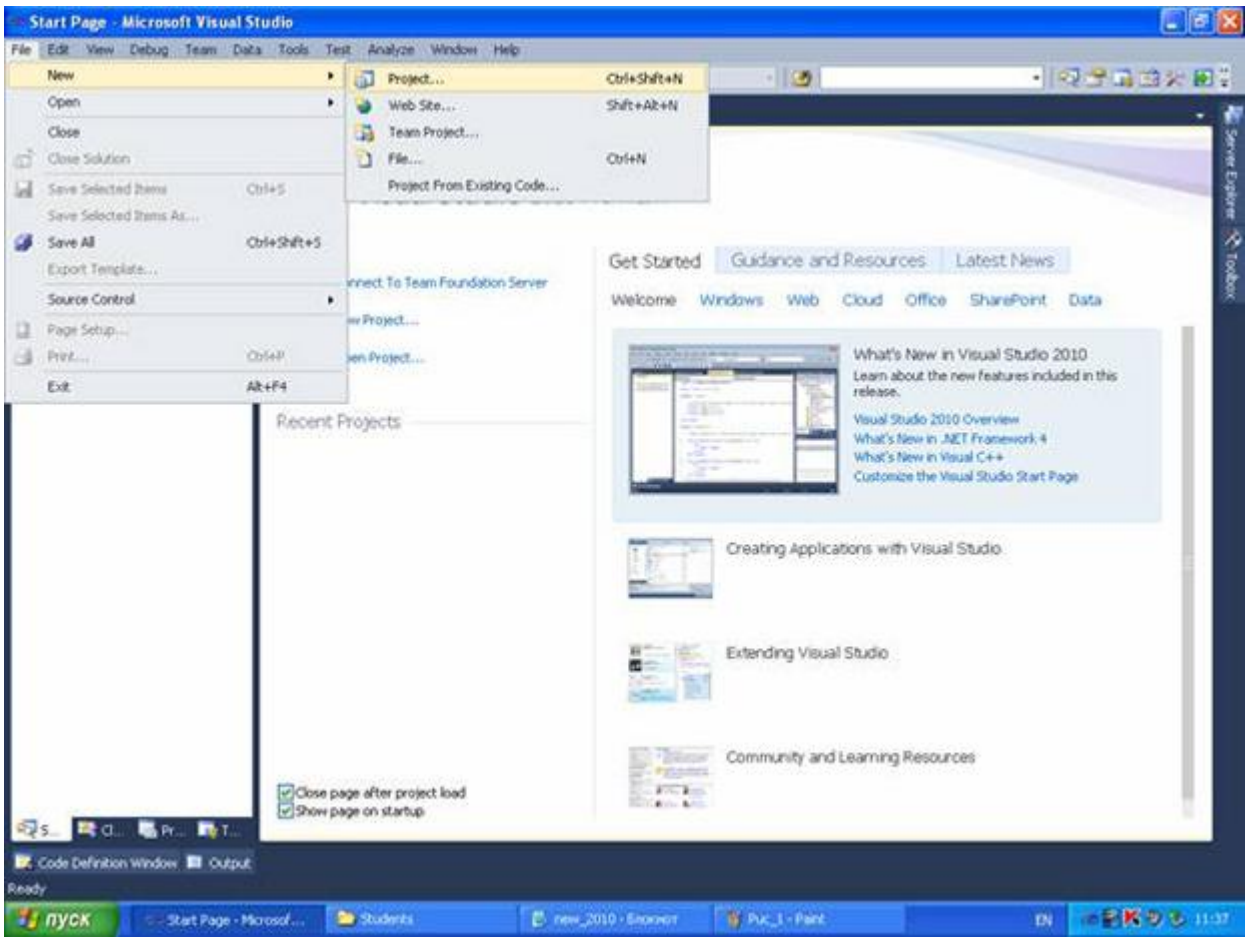


Рис. 1.2. Окно с выбором нового проекта

Среда Visual Studio отобразит окно New Project, в котором необходимо выбрать тип создаваемого проекта. Проект (project) используется вVisual Studio для логической группировки нескольких файлов, содержащих исходный код, на одном из поддерживаемых языков программирования, а также любых вспомогательных файлов. Обычно после сборки проекта (которая включает компиляцию всех входящих в проект файлов исходного кода) создается один исполняемый модуль.

В окне New Project следует развернуть узел Visual C++, обратиться к пункту Win32 и на центральной панели выбрать Win32 Console Application. Выбор этой опции показан на рис. рис. 1.3.

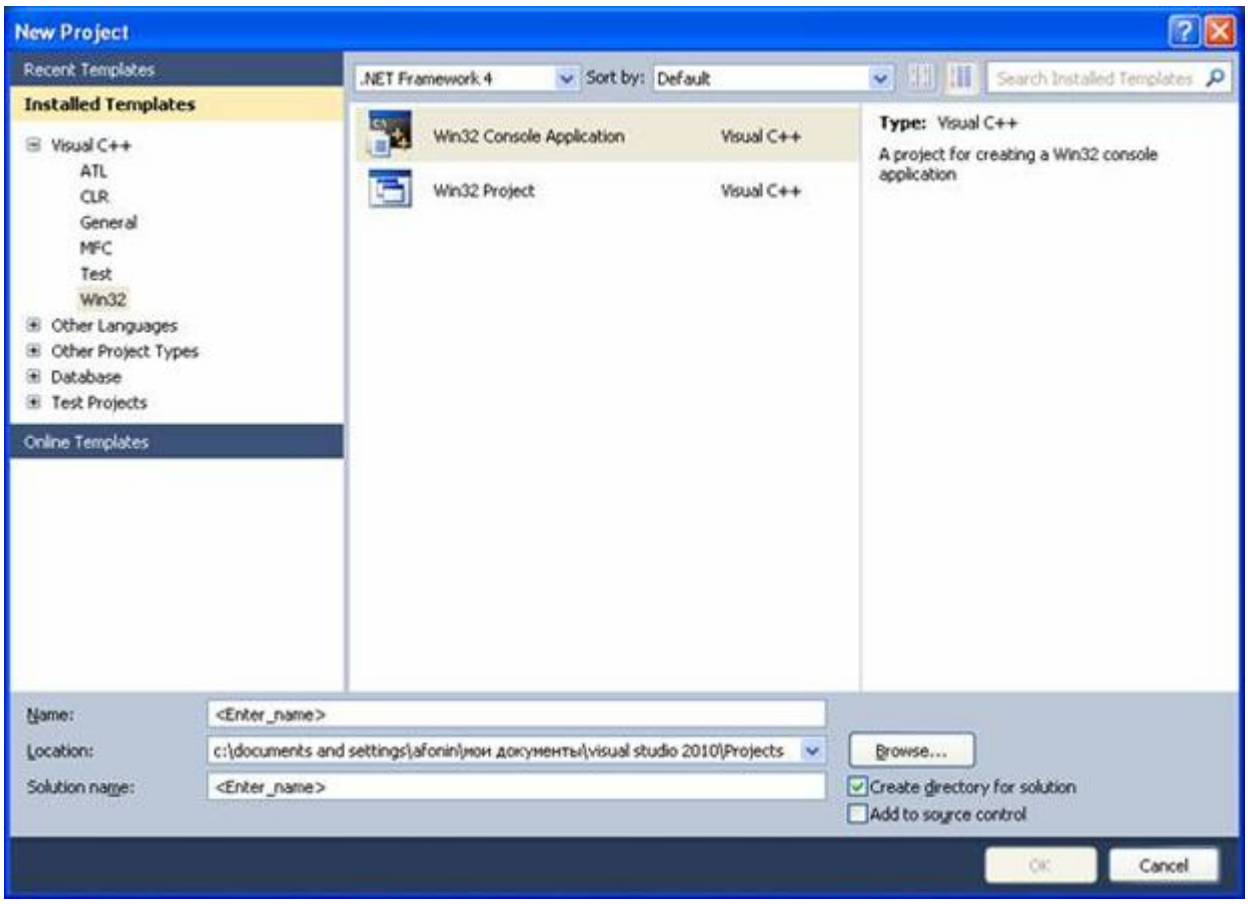


Рис. 1.3. Выбор типа проекта

Затем в поле редактора Name (где по умолчанию имеется <Enter_name>) следует ввести имя проекта, например, hello. В поле Locationможно указать путь размещения проекта, или выбрать путь размещения проекта с помощью клавиши (кнопки) Browse. По умолчанию проект сохраняется в специальной папке Projects. Пример выбора имени проекта показано на рис. 1.4.

Одновременно с созданием проекта Visual Studio создает решение. Решение (solution) – это способ объединения нескольких проектов для организации более удобной работы с ними.

После нажатия кнопки OK откроется окно Win32 Application Wizard (мастер создания приложений для операционных систем Windows), показанное на рис. 1.5.

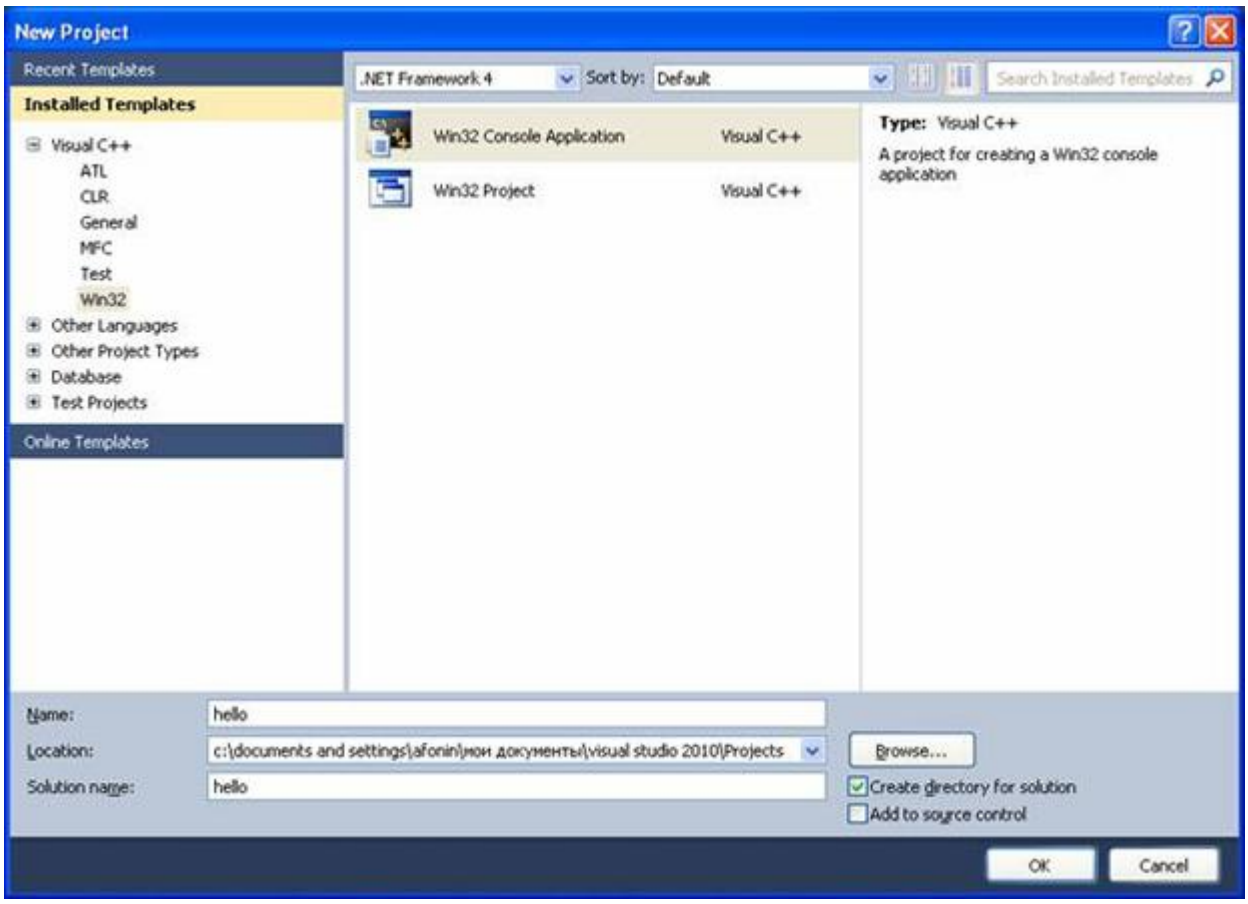


Рис. 1.4. Пример задания имени проекта

Выбор имени проекта может быть достаточно произвольным: допустимо использовать числовое значение, допустимо имя задавать через буквы русского алфавита.

В дальнейшем будем использовать имя, набранное с помощью букв латинского алфавита и, может быть, с добавлением цифр.


Рис. 1.5. Мастер создания приложения

На первой странице представлена информация о создаваемом проекте, на второй можно сделать первичные настройки проекта. После обращения к странице Application Settings, или после нажатия кнопки Next получим окно, показанное на рис. рис. 1.6.

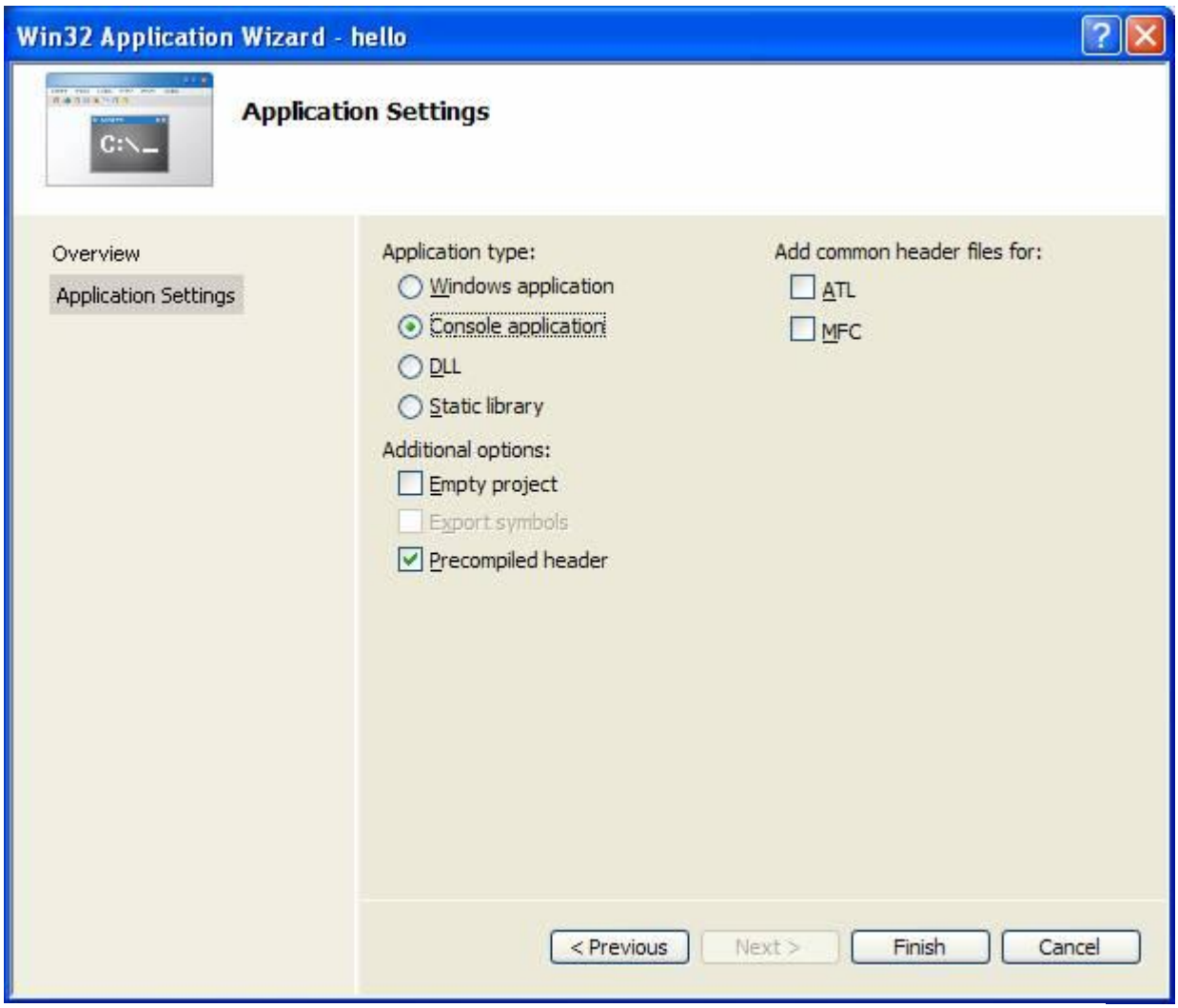


Рис. 1.6. Страница мастера настройки проекта по умолчанию

В дополнительных опциях ( Additional options ) следует поставить галочку в поле Empty project (пустой проект) и снять (убрать) галочку в поле Precompiled header. Получим экранную форму, показанную на рис. 1.7.

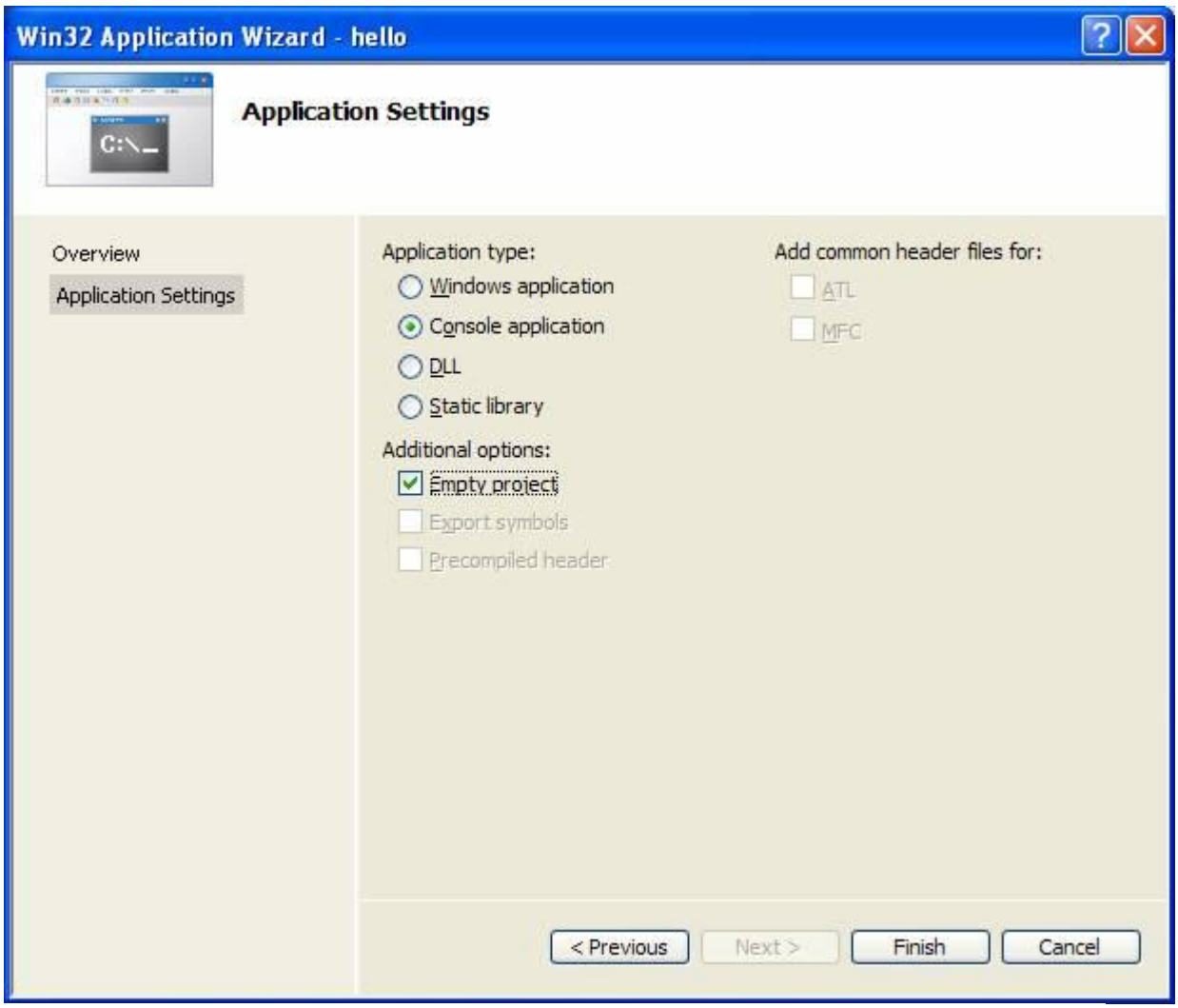


Рис. 1.7. Выполненная настройка мастера приложений

Здесь и далее будут создавать проекты по приведенной схеме, т.е. проекты в консольном приложении, которые должны создаваться целиком программистом (за счет выбора Empty project ). После нажатия кнопки Finish, получим экранную форму, показанную на рис. 1.8, где приведена последовательность действий добавления файла для создания исходного кода к проекту. Стандартный путь для этого: подвести курсор мыши к папке Source Files из узла hello в левой части открытого проекта приложения, выбрать Add и New Item (новый элемент).

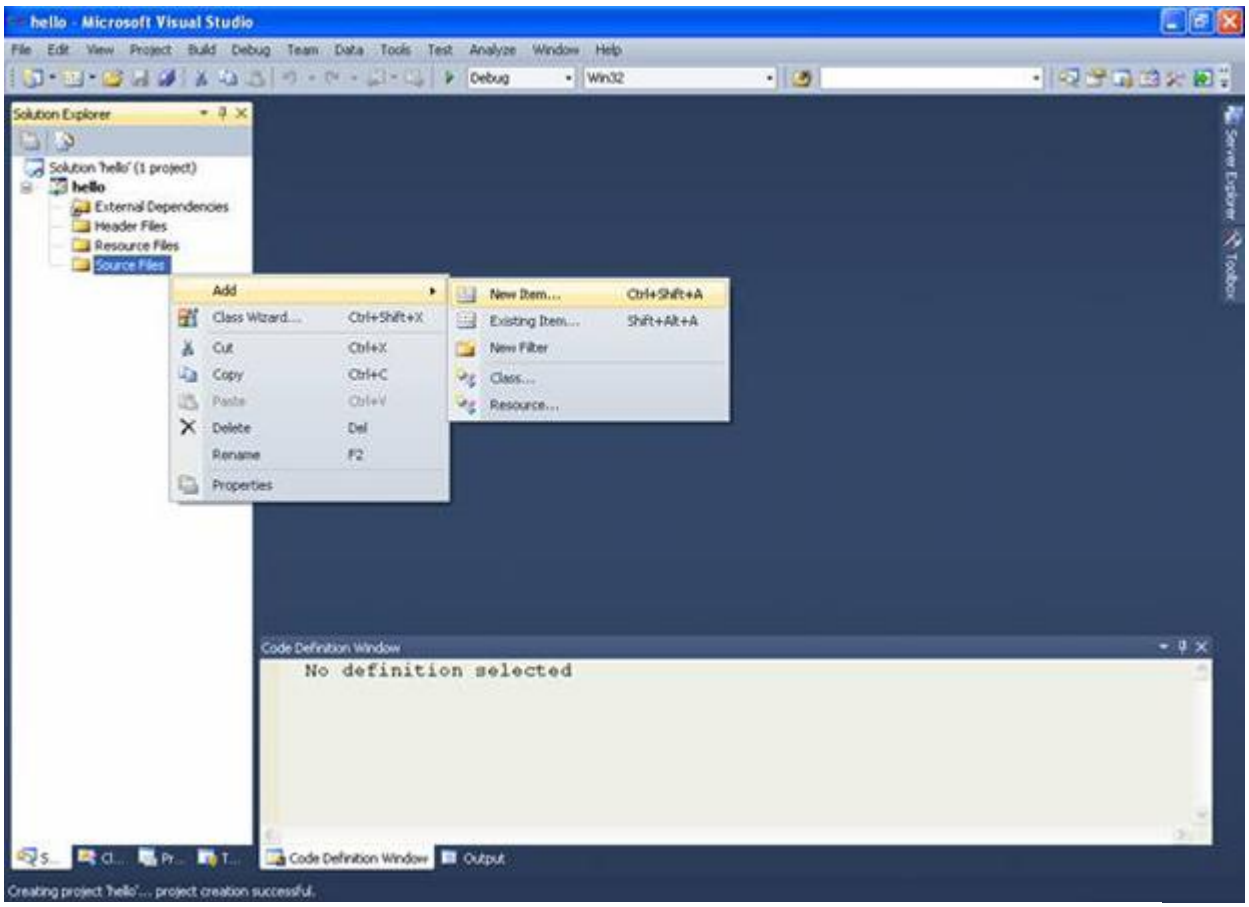


Рис. 1.8. Меню добавления нового элемента к проекту

После выбора (нажатия) New Item получим окно, показанное на рис. 1.9, где через пункт меню Code узла Visual C++ выполнено обращение к центральной части панели, в которой осуществляется выбор типа файлов. В данном случае требуется обратиться к закладкеC++ File (.cpp).

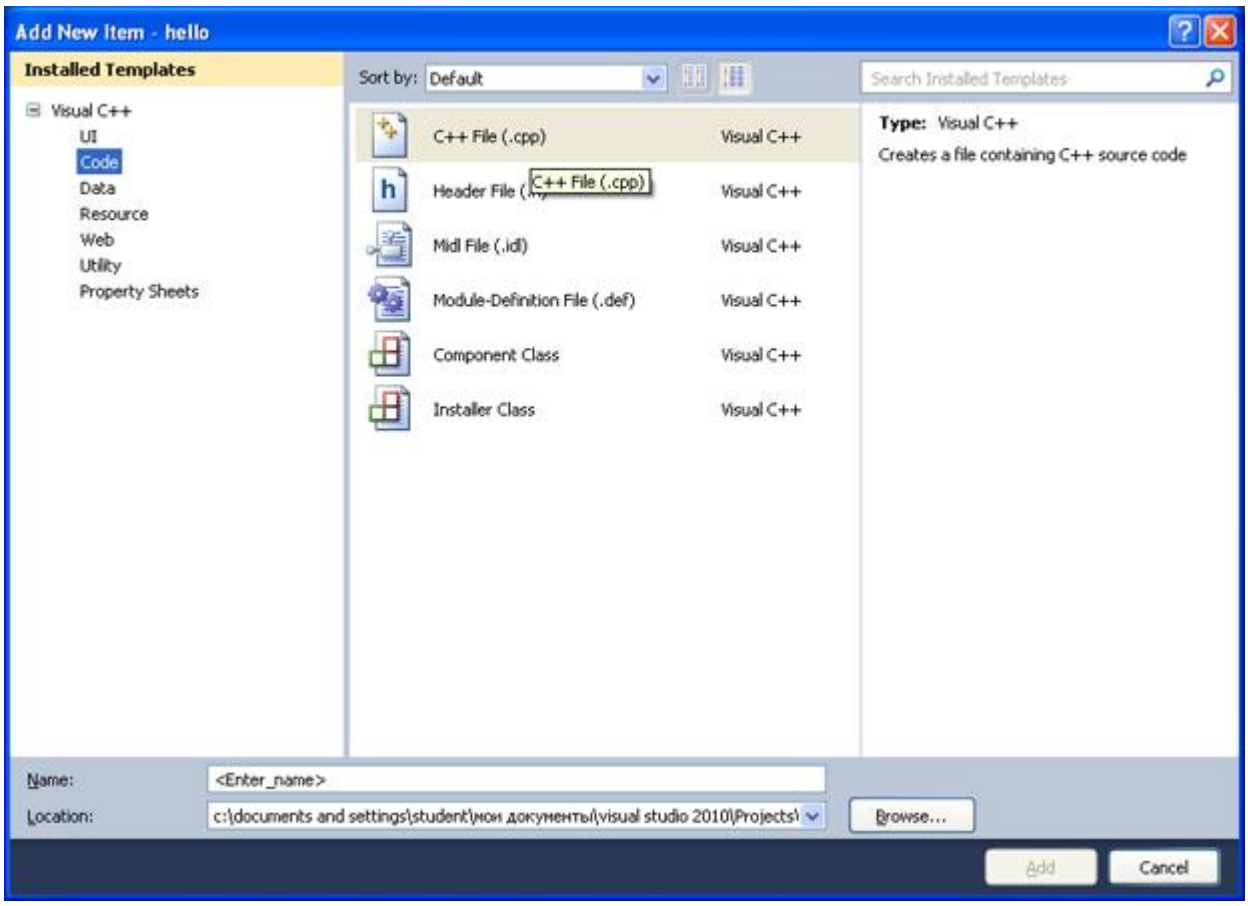


Рис. 1.9. Окно выбора типа файла для подключения к проект

Теперь в поле редактора Name (в нижней части окна) следует задать имя нового файла и указать расширение ".c". Например, main.c. Имя файла может быть достаточно произвольным, но имеется негласное соглашение, что имя файла должно отражать его назначение и логически описывать исходный код, который в нем содержится. В проекте, состоящем из нескольких файлов, имеет смысл выделить файл, содержащий главную функцию программы, с которой она начнет выполняться. В данном пособии такому файлу мы будем задавать имямain.c, где расширение .c указывает на то, что этот файл содержит исходный код на языке C, и он будет транслироваться соответствующим компилятором. Программам на языке C принято давать расширение .c. После задания имени файла в поле редактораName, получим форму, показанную на рис. 1.10.

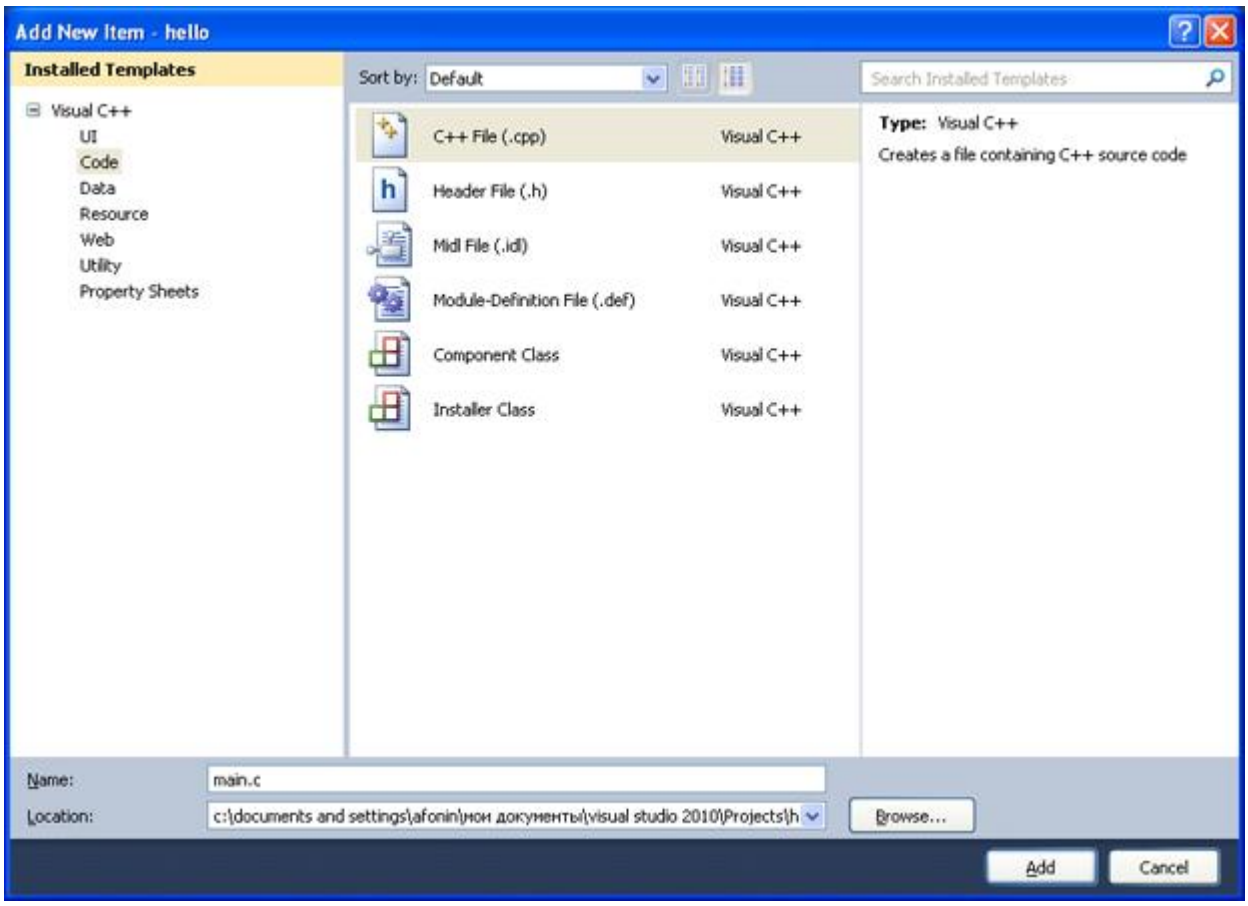


Рис. 1.10. Задание имени файла, подключаемому к проекту

Затем следует нажать кнопку Add. Вид среды Visual Studio после добавления первого файла к проекту показан на рис. 1.11. Добавленныйфайл отображается в дереве Solution Explorer под узлом Source Files (файлы с исходным кодом), и для него автоматически открывается редактор.

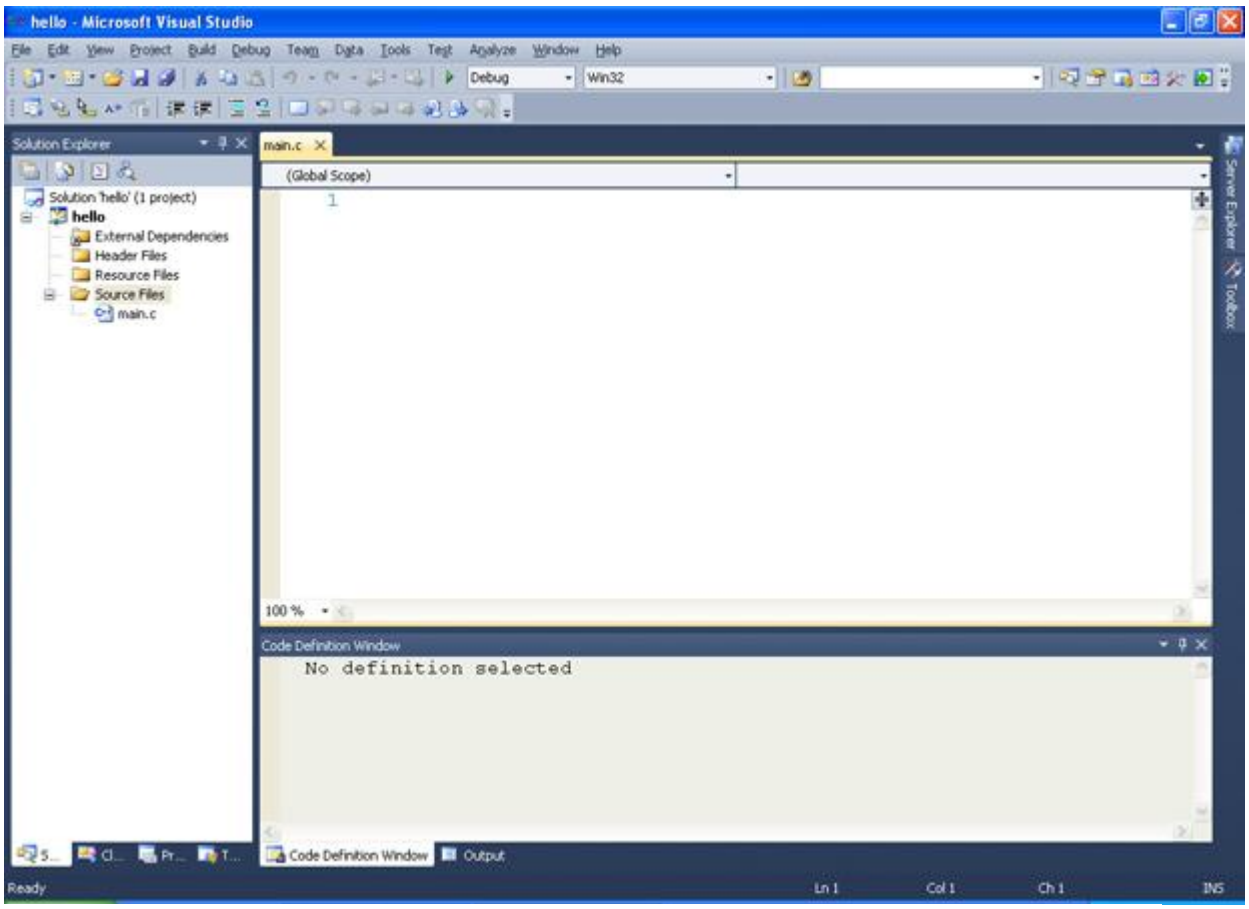


Рис. 1.11. Подключение файла проекта

На рис. 1.11 в левой панели в папке Solution Explorer отображаются файлы, включенные в проект в папках. Приведем описание.

Папка Source Files предназначена для файлов с исходным кодом. В этой папке отображаются файлы с расширением .c.

Папка Header Files содержит заголовочные файлы с расширением .h.

Папка Resource Files содержит файлы ресурсов, например изображения и т. д.

Папка External Dependencies отображает файлы, не добавленные явно в проект, но использующиеся в файлах исходного кода, например включенные при помощи директивы #include. Обычно в папке External Dependencies присутствуют заголовочные файлы стандартной библиотеки, использующиеся в проекте.

Следующий шаг состоит в настройке проекта. Для этого в меню Project главного меню следует выбрать hello Properties (или с помощью последовательного нажатия клавиш Alt+F7). Пример обращения к этому пункту меню показан на рис. 1.12.

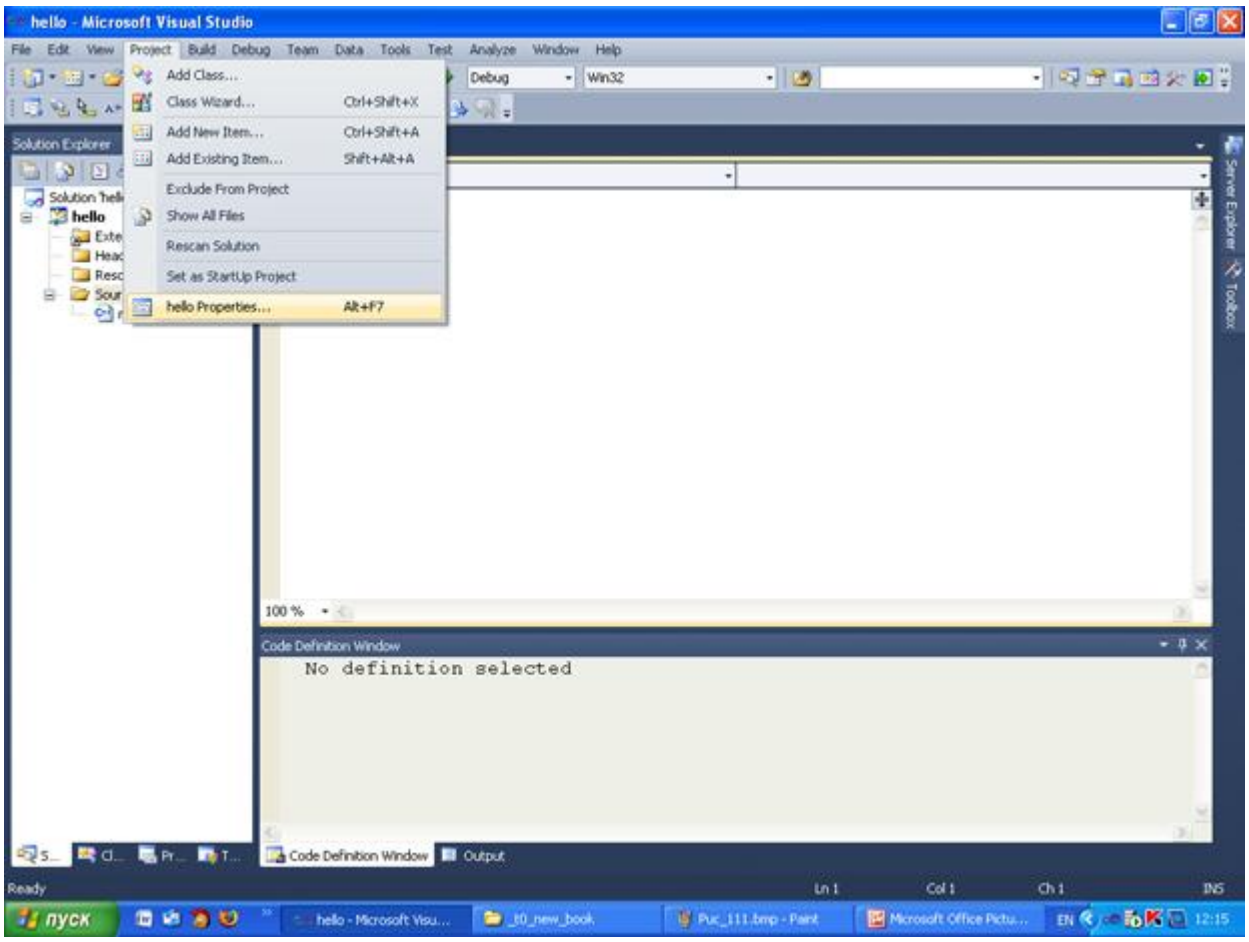


Рис. 1.12. Обращение к странице свойств проекта

После того как произойдет открытие окна свойств проекта, следует обратиться (с левой стороны) к Configuration Properties. Появится ниспадающий список, который показан на рис. 1.13. Выполнить обращение к узлу General, и через него в правой панели выбрать Character Set, где установить свойство Use Multi-byte Character Set. Настройка Character Set (набор символов) позволяет выбрать, какаякодировка символов – ANSI или UNICODE – будет использована при компиляции программы. Для совместимости со стандартом C89 мы выбираем Use Multi-Byte Character Set. Это позволяет использовать многие привычные функции, например, функции по выводу информации на консоль.

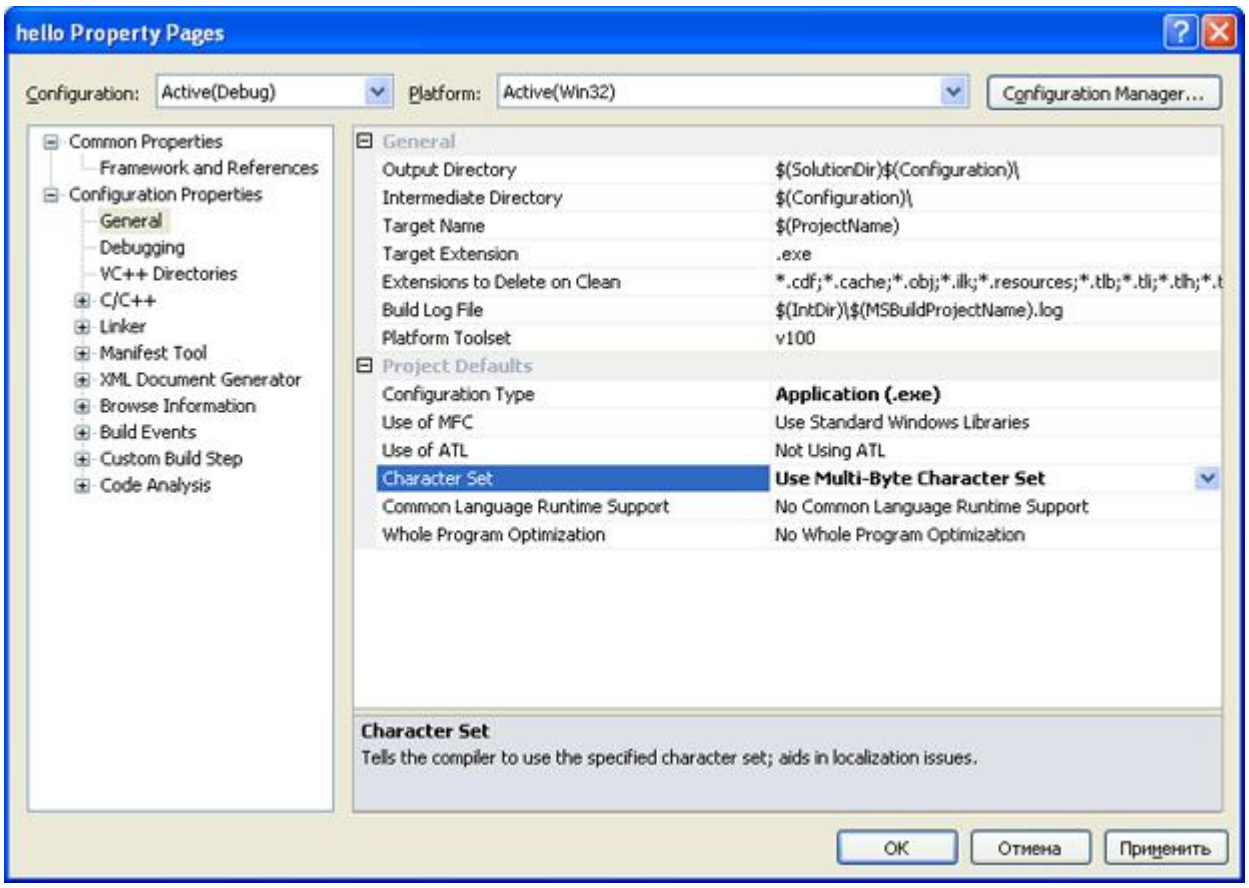


Рис. 1.13. Меню списка свойств проекта

После сделанного выбора, показанного на рис. 1.13, следует нажать кнопку Применить. Затем следует выбрать узел C/C++ и в ниспадающем меню выбрать пункт Code Generation, через который следует обратиться в правой части панели к закладке Enable C++ Exceptions, для которой установить No (запрещение исключений C++). Результат установки выбранного свойства показан на рис. 1.14. После произведенного выбора нажать кнопку Применить.

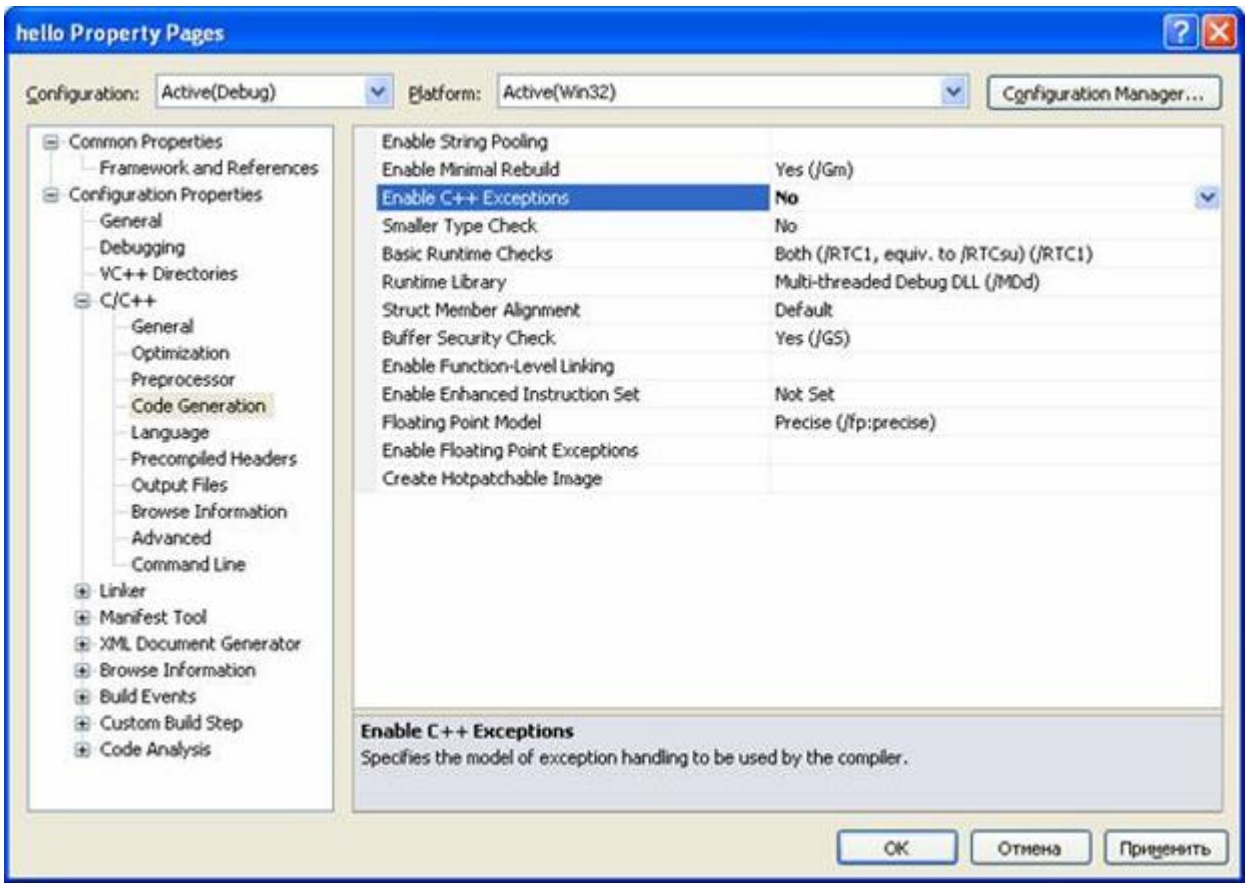


Рис. 1.14. Страница свойств для запрещения исключений C++

Далее в ниспадающем меню узла C/C++ необходимо выбрать пункт Language и через него обратиться в правую часть панели, где установить следующие свойства: свойство Disable Language Extensions (дополнительные языковые расширения фирмы Microsoft) вYes (/Za), свойство Treat wchar_t as Built-in Type (рассматривать тип wchar_t как встроенный тип) установить в No (/Zc:wchar_t–), свойство Force Conformance in For Loop Scope (соответствие стандарту определения локальных переменных в операторе цикла for ) установить в Yes(/Zc:forScope), свойство Enable Run-Time Type Info (разрешить информацию о типах во время выполнения) установить в No (/GR–), свойство Open MP Support (разрешить расширение Open MP – используется при написании программ для многопроцессорных систем) установить в No(/openmp–).

Результат выполнения этих действий показан на рис. 1.15.

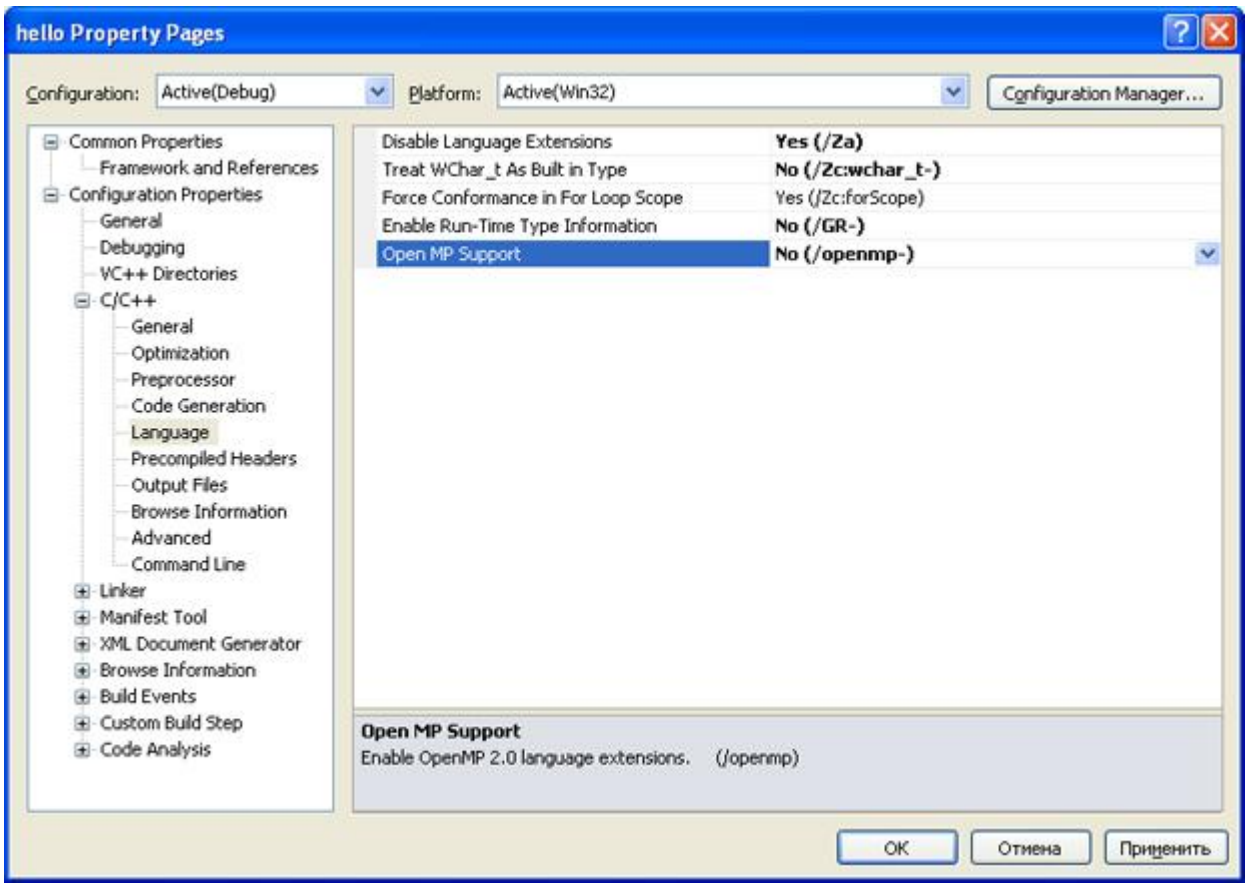


Рис. 1.15. Страница свойств закладки Language

После выполнения указанных действий следует нажать клавишу Применить. Далее в ниспадающем списке узла C/C++ следует выбратьпункт Advanced и в правой панели изменить свойство Compile As в свойство компиляции языка C, т.е. Compile as C Code (/TC). Результат установки компилятора языка C показан на рис. 1.16.

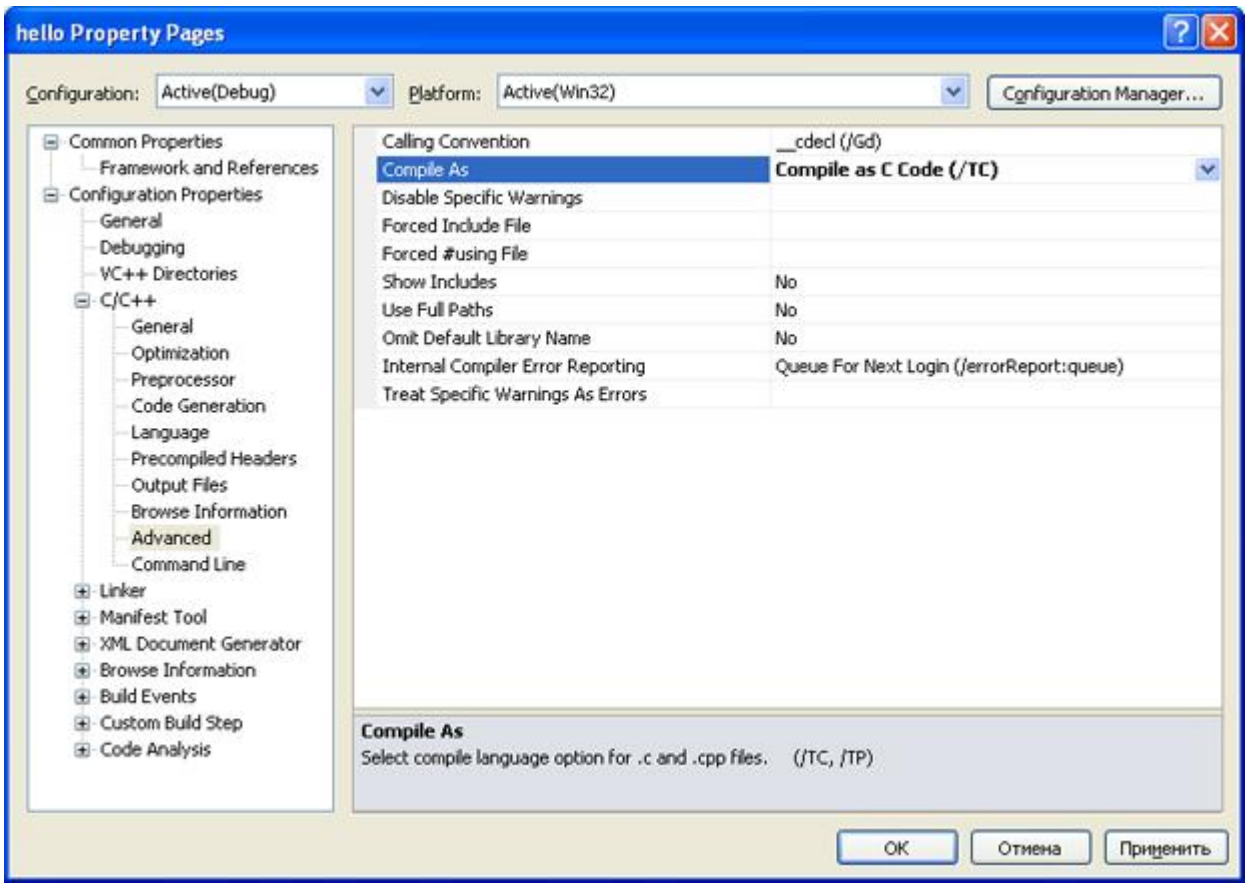


Рис. 1.16. Результат выбора режима компиляции языка С

После нажатия клавиш Применить и ОК сначала откроется подготовленный проект с пустым полем редактора кода, в котором можно начать писать программы. В этом редакторе наберем программу, выводящую традиционное приветствие "Hello World". Для компиляции созданной программы можно обратиться в меню Build, или, например, набрать клавиши Ctr+F7. В случае успешной компиляции получим следующую экранную форму, показанную на рис. 1.17.

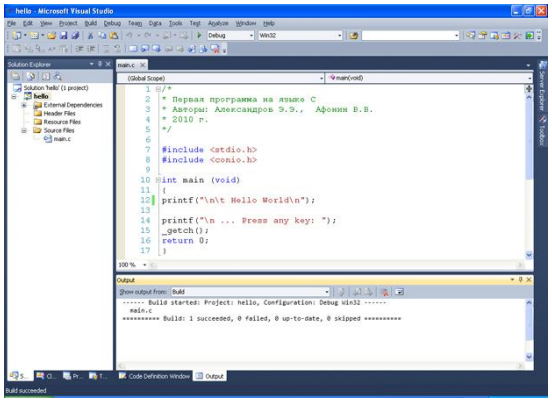

Рис. 1.17. Успешно откомпилированная первая программа на языке С

Для приведенного кода программы запуск на ее исполнение из окна редактора в Visual Studio 2010 можно нажать клавишу F5. рис. 1.18 показан результат исполнения первой программы.

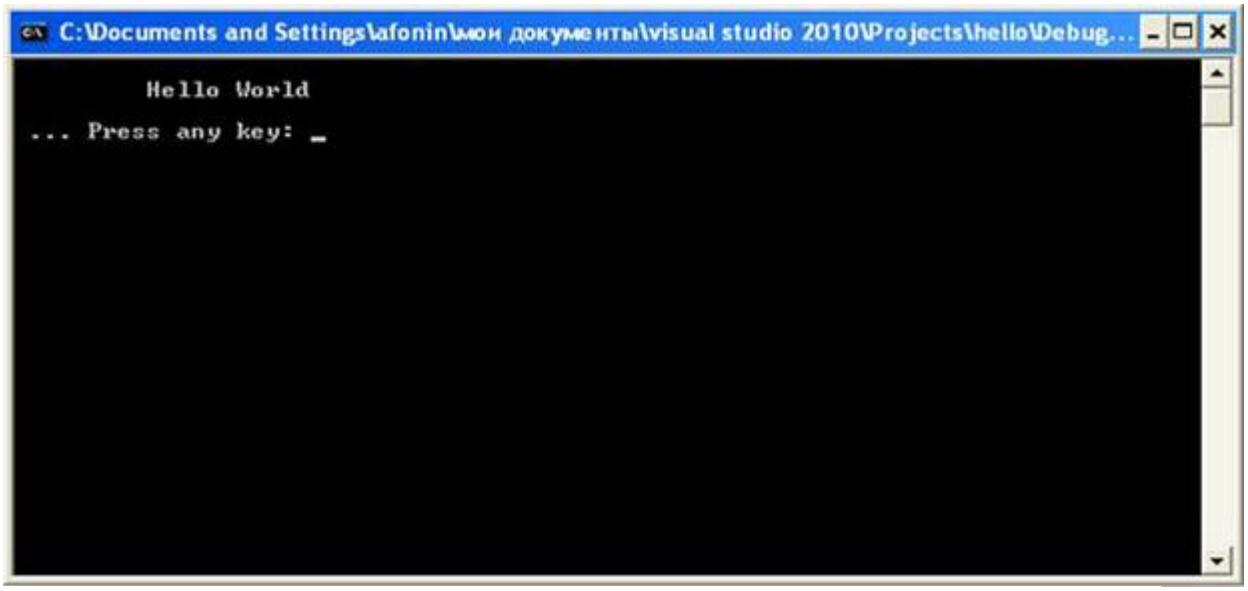


Рис. 1.18. Консольный вывод первой программы на языке

Произведем разбор первой программы. Во-первых, надо отметить, что в языке С нет стандартных инструкций (операторов) для вывода сообщений на консоль (окно пользователя). В языке С предусматриваются специальные библиотечные файлы, в которых имеются функции для этих целей. В приведенной программе используется заголовочный файл с именем stdio.h (стандартный ввод–вывод), который должен быть включен в начало программы. Для вывода сообщения на консоль используется функция printf(). Для работы с консолью включен также заголовочный файл conio.h., который поддерживает функцию _getch(), которая извлекает символ из потока ввода, т. е. она предназначенная для приема сообщения о нажатии какой-либо (почти любой) клавиши на клавиатуре. С другими компиляторами, возможно, потребуется getch(), т.е. без префиксного нижнего подчеркивания. Строка программы

int main (void)

сообщает системе, что именем программы является main() – главная функция, и что она возвращает целое число, о чем указывает аббревиатура "int". Имя main() – это специальное имя, которое указывает, где программа должна начать выполнение [1.1]. Наличие круглых скобок после слова main() свидетельствует о том, что это имя функции. Если содержимое круглых скобок отсутствует или в них содержится служебное слово void, то это означает, что в функцию main() не передается никаких аргументов. Тело функции main()ограничено парой фигурных скобок. Все утверждения программы, заключенные в фигурные скобки, будут относиться к функции main().

В теле функции main() имеются еще три функции. Во-первых, функции printf() находятся в библиотеке компилятора языка С, и они печатают или отображают те аргументы, которые были подставлены вместо параметров. Символ "\n" составляет единый символ newline(новая строка), т.е. с помощью этого символа осуществляется перевод на новую строку. Символ "\t"

осуществляет табуляцию, т.е. начало вывода результатов программы с отступом вправо.

Функция без параметров _getch() извлекает символ из потока ввода (т.е. ожидает нажатия почти любой клавиши). С другими компиляторами, возможно, потребуется getch(), т.е. без префиксного нижнего подчеркивания.

Последнее утверждение в первой программе

return 0;

указывает на то, что выполнение функции main() закончено и что в систему возвращается значение 0 (целое число). Нуль используется в соответствии с соглашением об индикации успешного завершения программы [1.3].

В завершение следует отметить, что все действия в программе завершаются символом точки с запятой.

Все файлы проекта сохраняются в той папке, которая сформировалась после указания в поле Location имени проекта (hello). На рис. 1.19 показаны папки и файлы проекта Visual Studio 2010..

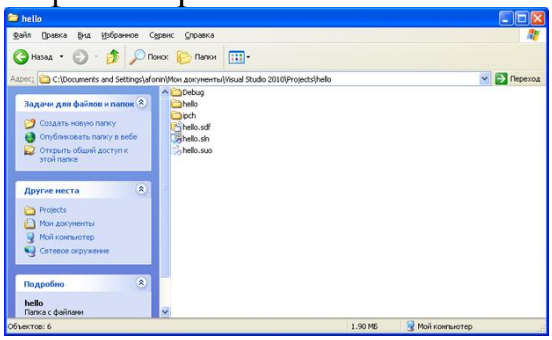


Рис. 1.19. Файлы и папки созданного проекта

На рис. 1.19 файлы с полученными расширениями означают:

hello.sln – файл решения для созданной программы. Он содержит информацию о том, какие проекты входят в данное решение. Обычно, эти проекты расположены в отдельных подкаталогах. Например, наш проект находится в подкаталоге hello;

hello.suo – файл настроек среды Visual Studio при работе с решением, включает информацию об открытых окнах, их расположении и прочих пользовательских параметрах.

hello.sdf – файл содержащий вспомогательную информацию о проекте, который используется инструментами анализа кода Visual Studio, такими как IntelliSense для отображения подсказок об именах и т.д.

Файлы папки Debug показаны на рис. 1.20.

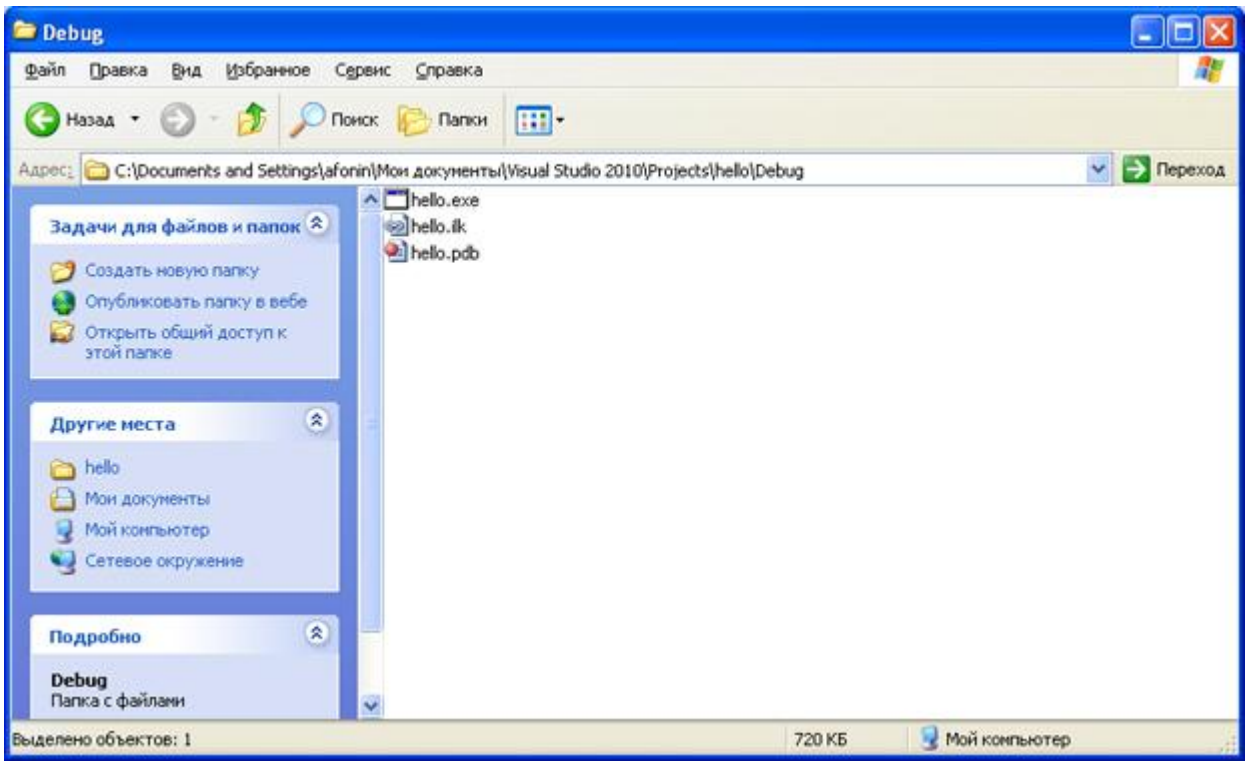


Рис. 1.20. Файлы папки Debug

Рассмотрим файлы в соответствии с рис. 1.20.

hello.exe – исполняемый файл проекта;

hello.ilk – файл "incremental linker", используемый компоновщиком для ускорения процесса компоновки;

hello.pdb – отладочная информация/информация об именах в исполняемых файлах, используемая отладчиком.

Файлы папки hello показаны на рис. 1.21.

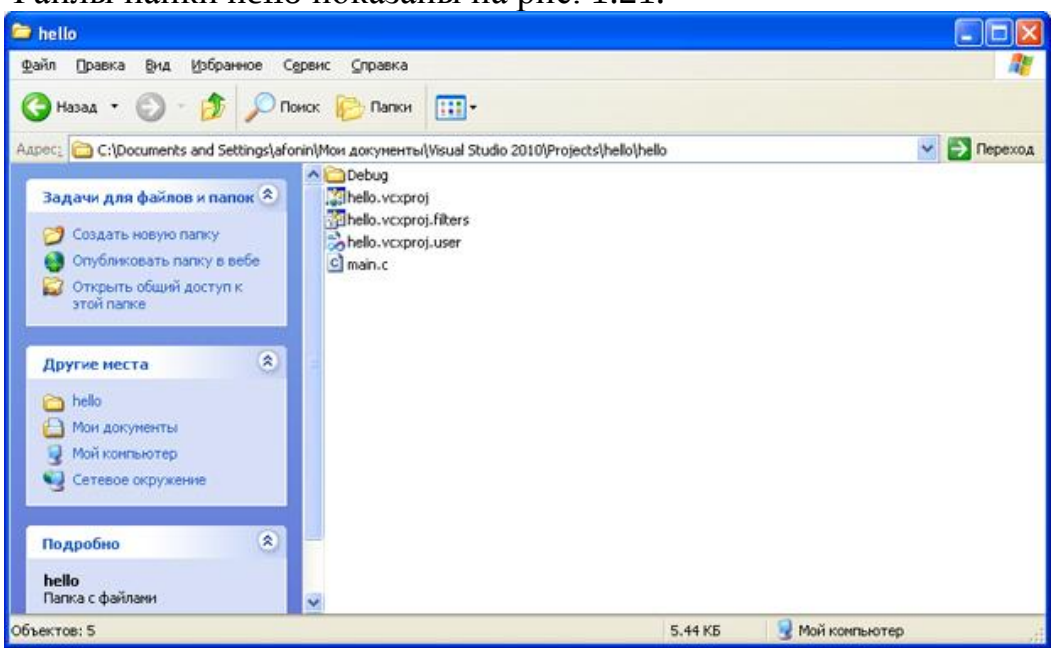


Рис. 1.21. Содержимое папки hello

Характеристика содержимого папки hello:

main.c – файл исходного программного кода,

hello.vcxproj – файл проекта,

hello.vcxproj.user – файл пользовательских настроек, связанных с проектом,

hello.vcxproj.filters – файл с описанием фильтров, используемых Visual Studio Solution Explorer для организации и отображения файлов с исходным кодом.

Практическая часть

В практической части выполните следующие задания на основе рассмотренной программы hello:

Напишите программу, которая выводила бы на консоль название факультета, где учитесь, номер группы, свою фамилию, имя и отчество в разных строках дисплея (консоли) с помощью одной функции printf().

Вывод выполните с помощью нескольких функций printf() (количество функций должно соответствовать каждой порции информации).

Для задания пункта 2 вывод информации выполните в различных строках подряд, т.е. без межстрочного пропуска.

Проверьте программу без ключевого слова void для функции main().

Примечание. Вывод требуемой информации осуществляется с помощью букв латинского алфавита. Комментарии в программе могут быть сделаны после символа "//" или внутри комбинации символов "/* */".

**Список индивидуальных заданий**

1. Напишите программу «Hello, world!» и преобразуйте ее в программу «Привет, мир!».
2. Напишите программу для перевода температуры в градусах по Фаренгейту в градусы по Цель- сию по формуле C = 5/9 (F – 32).
3. Напишите программу для вычисления площади треугольника по трем сторонам.
4. Заданы моменты начала и конца некоторого промежутка времени в часах, минутах и секундах (в пределах одних суток). Найти продолжительность этого промежутка в тех же единицах..

**Лабораторная работа №2. Файловый ввод-вывод, операторы манипулирования битами, работа с массивами.**

**Цель работы:** выработать практические навыки работы с VBScript изучить встроенные и внешние объекты. научиться создавать, вводить в компьютер, выполнять и исправлять простейшие программы на языке VBScript в режиме диалога, познакомиться с диагностическими сообщениями компилятора об ошибках при выполнении программ, реализующих линейные алгоритмы

**Общие сведения:**

Язык С++ также поддерживает концепцию объектно-ориентированного ввода-вывода, но потоки здесь представлены в виде объектов классов библиотеки *iostream*:

*ifsteram* - для ввода из файла;

*ofsteram* - для вывода в файл;

*fsteram* - для обмена с файлом в двух направлениях.

При этом стандартные потоки представлены объектами:

*cin* – стандартный поток ввода;

*cout* – стандартный поток вывода;

*cerr* – стандартный поток приема сообщений об ошибках.

Принцип работы с потоками аналогичен принятому в языке Си - программист создает объект-поток как экземпляр требуемого класса, связывая его с файлом на диске, после чего используя сервисные функции. Для чтения данных их потока можно использовать перегруженную операцию >>, для записи информации в поток используется операция <<. Принципиальное отличие объектно-ориентированного подхода Си++ заключается в том, что весь функционал работы с потоком сосредоточен в методах классов, которые должны вызываться в контексте объекта-потока. В листинге 4 приведен пример обмена данными программы с файлом на языке Си++.

```
#include <fstream.h>
…
 fstream f; char str[20];
 f.open("temp",ios::in|ios::out); //открываем поток на
//чтение и запись
 f.write(“Тест”,4);  //пишем в поток слово Тест
 f.seekg(0);    //перемещаем указатель потока в его начало
 f>>str;                 //читаем строку из потока
 cout<<str;
```

Таким образом, эффективное использование потокового ввода-вывода на языке Си++ должно опираться на знание методов и компонентных данных классов-потоков. Как видно из примера, названия методов во многом схожи с названиями функции библиотеки Си и разобраться в их назначении и способах применения будет совсем несложно. Ввод-вывод в Visual C++ выполняется с помощью подсистемы ввода- вывода и классов библиотеки.

Обмен данными реализуется с помощью по- токов. Поток (stream) – это абстрактное понятие, относящееся к любому перено- су данных от источника к приемнику.

Потоки обеспечивают надежную работу как со стандартными, так и с определенными пользователем типами данных, а также единообразный и понятный синтаксис. Поток определяется как последовательность байт и не зависит от конкретного устройства, с которым производится обмен (оперативная память, файл на диске, клавиатура или принтер). Обмен с потоком для повышения скорости передачи данных производит- ся, как правило, через специальную область оперативной памяти – буфер.

Буфер выделяется для каждого открытого файла. При записи в файл вся информация сначала направляется в буфер и там накапливается до тех пор, пока весь буфер не заполнится. Только после этого или после выполнения специальной команды сброса буфера происходит передача данных на внешнее устройство. При чтении из файла данные вначале считываются в буфер, причем не столько сколько запрашивается, а сколько помещается в буфер. Механизм буферизации позволяет быстро и эффективно обмениваться информацией с внешними устройствами. Для поддержки потоков библиотека содержит иерархию классов, которые определены в пространстве имен System.IO. Помимо классов там описано большое число перечислений для задания различных свойств и режимов. Классы библиотеки позволяют работать в различных режимах с файлами, каталогами и областями оперативной памяти.

Краткое описание исполь- зуемых для этих целей классов приведено в табл. 2.1. Как следует из содержимого табл. 2.1 выполнять обмен данными с внешними устройствами можно на 15 следующих уровнях: двоичного представления данных (BinaryReader, BinaryWriter), байтов (FileStream), текста (SreamWriter, StreamReader).

Таблица 2.1 Основные классы пространства имен System.IO

| Название класса | Назначение |
|---|---|
| BinaryReader, BinaryWriter | Чтение и запись значений простых встроенных типов (це- лочисленных, логических, встроенных и т.п.) во внутрен- ней форме представления |
| BufferedStream | Временное хранение потока байт (например, для после- дующего переноса в постоянное хранилище) |
| Directory, DirectoryInfo, File, FileInfo | Работа с каталогами или физическими файлами: создание, удаление, получение их свойств. Возможности классов |

| | |
|---|---|
| File и Directory | реализованы в основном в виде статических методов. Аналогичные классы DirectoryInfo и FileInfo используют обычные методы |
| FileStream | Произвольный (прямой) доступ к файлу, представленному как поток байт |
| MemoryStream | Произвольный доступ к потоку байт в оперативной памя- ти |
| SreamWriter, StreamReader | Чтение из файла и запись в файл текстовой информации (произвольный доступ не поддерживается) |
| StringWriter, StringReader | Работа с текстовой информации в оперативной памяти |

В .NET используется кодировка Unicode, в которой каждый символ коди- руется двумя байтами. Классы, работающие с текстом, являются оболочками классов, использующих байты, и автоматически выполняют перекодирование из байтов в символы и обратно. Двоичные и байтовые потоки хранят данные в том же виде, в котором они представлены в оперативной памяти, т. е. при об- мене с файлом происходит побитовое копирование информации. Двоичные файлы применяются не для просмотра их человеком, а для использования в программах. Доступ к файлам может быть последовательным, когда очередной эле- мент можно прочитать (записать) только после аналогичной операции с пре- дыдущим элементом, и произвольным (прямым), при котором выполняется чтение (запись) произвольного элемента по заданному адресу. Текстовые фай- лы позволяют выполнить только последовательный доступ, в двоичных и бай- товых потоках можно использовать оба метода. Прямой доступ в сочетании с отсутствием преобразований позволяет обеспечить высокую скорость получе- ния нужной информации. Методы форматированного ввода, с помощью которых можно выполнять ввод с клавиатуры или из текстового файла значений арифметических типов 16 (функции scanf и printf языка C) в языке Visual C# не поддерживаются. Для преобразования из символьного в числовое представление следует исполь- зовать либо методы класса Convert, либо метод Parse. Форматированный вывод, т.е. преобразование из внутренней формы представления числа в сим- вольную, понятную человеку, выполняются с помощью перегружаемых мето- дов ToString, результаты выполнения которых передаются в методы тексто- вых файлов. Использование классов файловых потоков в программе предполагает вы- полнение следующих операций: 1) Создание потока и связывание его с физическим файлом. 2) Обмен данными (ввод-вывод). 3) Закрытие файла. Каждый класс файловых потоков содержит несколько вариантов конст- рукторов, с помощью которых можно создавать объекты этих классов различ- ными способами и в различных режимах. Например, файлы можно открывать только для чтения, только для записи или для чтения и записи. Эти режимы доступа к файлу (константы) содержатся в перечислении FileAccess

(Read – открыть файл только для чтения; ReadWrite – открыть файл для чтения и записи, Write – открыть файл только для записи), определенном в пространстве имен System.IO. Возможные режимы открытия файла определены в перечислении FileMode (табл. 2.2).

Таблица 2.2 Значения перечисления FileMode

| Значение | Описание |
|---|---|
| Append | Открыть файл, если он существует, и установить текущий указатель в конец файла. Если файл не существует, создать новый файл |
| Create | Создать новый файл. Если в каталоге уже существует файл с таким же именем, он будет стерт. |
| CreateNew | Создать новый файл. Если в каталоге уже существует файл с таким же именем, то возникает исключение IOException. |
| Open | Открыть существующий файл |
| OpenOrCreate | Открыть файл, если он существует. Если нет, создать файл с таким же именем. |
| Truncate | Открыть существующий файл. После открытия он должен быть обрезан до нулевой длины |

Режим FileMode.Append можно использовать только совместно с доступом типа FileAccess.Write, т. е. для файлов, открываемых для записи. Режимы совместного использования файла различными пользователями определяет перечисление FileShare (табл. 2.3). Символьные потоки StreamWriter и StreamReader работают с Unicode-символами, поэтому ими удобнее всего пользоваться для работы с файла- 17 ми, предназначенными для восприятия человеком. Эти потоки являются наследниками классов TextWriter и TextReader соответственно, которые обеспечивают их большей частью функциональностью. В таблицах 2.4 и 2.5 приведены наиболее важные элементы этих классов.

Таблица 2.3 Значение перечисления FileShare.

| Значение | Описание |
|---|---|
| None | Совместное использование открытого файла запрещено. Запрос на открытие данного файла завершается обработкой сообще- ния об ошибке |
| Read | Позволяет открывать файл для чтения одновременно несколь- ким пользователям. Если этот флаг не установлен, то запросы на открытие файла для чтения завершаются сообщением об ошибке |
| ReadWrite | Позволяет открывать файл для чтения и записи одновременно нескольким пользователям |
| Write | Позволяет открывать файл для записи одновременно несколь- ким пользователям |

Таблица 2.4 Наиболее важные элементы базового класса TextWriter

| Элемент | Описание |
|---|---|
| | Close Закрыть файл и освободить связанные с ним ресурсы. Если в процессе записи используется буфер, то он будет автоматически очищен |
| Flush | Очистить все буферы для текущего файла и записать накоплен- ные в них данные в место их постоянного хранения. Сам файл при этом не закрывается |
| NewLine | Используется для задания последовательности символов, означающих начало новой строки. По умолчанию используется последовательность «возврат каретки» - «перевод строки »(\r\n) |
| Write | Записать фрагмент текста в поток |
| WriteLine | Записать строку в поток и перейти на другую строку |

Таблица 2.5 Наиболее важные элементы базового класса TextReader

| Элемент | Описание |
|---|---|
| Peek | Возвратить следующий символ, не изменяя позицию указателя в файле |
| Read | Считать данные из входного потока |
| ReadLine | Считать строку из из текущего потока и возвратить ее значение типа string. Пустая строка (null) означает конец файла (EOF) |
| ReadToEnd | Считать все символы до конца потока, начиная с текущей позиции и возвратить считанные данные как одну строку типа string |

Если, например, требуется записать значение переменной x типа double в текстовый файл, то текст программы будет выглядеть примерно следующим образом: …

```
// подключение пространства имен
System.IO using System.IO; …
// Создание потока и связывание его с физическим файлом
StreamWriter f = new StreamWriter("d:/f1.txt");
// запись значения переменной x в файл
f.WriteLine(Convert.ToString(x));
// закрытие потока
f.Close();
```

**Контрольные вопросы:**

1. Конструкторы и деструкторы классов.
2. Создание и вызов объектов.
3. Дружественные функции.
4. Перегруженные операции и функции.
5. Параметризированные классы.
6. Статические компоненты класса.
7. Производные классы. Наследование.

8. Доступ к наследуемым компонентам.
9. Виртуальные функции.
10. Виртуальные деструкторы.
11. Абстрактные классыю
12. Ввод-вывод данных, определенных пользователем.
13. Работа с дисковыми файлами.
14. Обработка исключений.
15. Общие сведения о MFC.
16. Объекты приложения.
17. Объекты главного окна, вида, документа.
18. Конструирование диалоговых окон и средств управления.
19. Обработка сообщений.
20. Структура прикладной программы с использованием MFC.
21. Контекст устройства.
22. Рисование с помощью CDC.
23. Метафайлы.
24. Система драйверов ODBC.
25. Источники данных.

**Список индивидуальных заданий:**

1.

**Задачи повышенной сложности**

**Лабораторная работа №3. Модульная организация программ. Работа со структурами. Указатели и динамическая память.**

**Цель работы:** научиться правильно использовать условный оператор if; научиться составлять программы решения задач на разветвляющиеся алгоритмы.

**Общие сведения.**

В C++ так же, как и в других языках программирования, класс – создаваемый программистом структурный тип данных, который используется для описания множества объектов предметной области, имеющих общие свойства и поведение.

Класс объявляется следующим образом:

class { private: protected: public: };

Описание предусматривает три секции.

Компоненты класса, объявленные в секции private, называются внутренними. Они доступны только компонентным функциям того же класса и функциям, объявленным дружественными описываемому классу.

Компоненты класса, объявленные в секции protected, называются защищенными. Они доступны компонентным функциям не только данного класса, но и его потомков. При отсутствии наследования – интерпретируются как внутренние.

Компоненты класса, объявленные в секции public, называются общими. Они доступны за пределами класса в любом месте программы. Именно в этой секции осуществляется объявление полей и методов интерфейсной части класса.

Если при описании класса тип доступа к компонентам не указан, то по умолчанию принимается тип private.

В качестве компонентов в описании класса фигурируют поля, применяемые для хранения параметров объектов, и функции, описывающие правила взаимодействия с этими полями. В соответствии со стандартной терминологией ООП функции – компоненты класса или компонентные функции можно называть методами.

Пример 1.1. Описание класса. А. Описание компонентных функций внутри класса.

#include class First { public: char c; int x,y;

/* компонентные функции, определенные внутри класса */

void print() { printf ("%c %d %d ",c,x,y); }

void set(char ach,int ax,int ay) { c=ach; x=ax; y=ay; } };

Б. Описание компонентных функций вне класса.

#include class First

{ public: char c;

```
int x,y;
void print();
void set(char ach,int ax,int ay); };
 /* компонентные функции, описанные вне класса */
void First::print() {
printf ("%c %d %d ",c,x,y); }
void First::set (char ach,int ax,int ay)
{
c=ach; x=ax; y=ay;
}
```

В программе, использующей классы, по мере необходимости объявляют объекты этих классов. Объекты – переменные программы, соответственно на них распространяются общие правила длительности существования и области действия переменных, а именно: внешние, статические и внешние статические объекты создаются до вызова

- функции main() и уничтожаются по завершении программы; автоматические объекты создаются каждый раз при вызове функции, в которой
- они объявлены, и уничтожаются при выходе из нее; объекты, память под которые выделяется динамически, создаются оператором
- new и уничтожаются оператором delete.

При объявлении полей в описании класса не допускается их инициализация, поскольку в момент описания класса память для размещения его полей еще не выделена.

Выделение памяти осуществляется не для класса, а для объектов этого класса, поэтому возможность инициализации полей появляется только во время или после объявления объекта конкретного класса.

Объявление объектов и способы инициализации их полей зависят от наличия или отсутствия в классе специального инициализирующего метода – конструктора, а также от того, в какой секции класса описано инициализируемое поле.

Конструктор, являясь методом класса, может инициализировать любое поле объекта при его создании.

Если в классе отсутствует конструктор, но описаны защищенные protected или скрытые private поля, то возможно создание только неинициализированных объектов.

Для этого используется стандартная конструкция объявления переменных или указателей на них.

Например:

```
 First a, // объект класса First
*b, // указатель на объект класса First
c[4]; // массив с из четырех объектов класса First
```

При объявлении указателя, как и для обычных переменных, память под объект не выделяется. Это необходимо сделать отдельно, используя операцию new, после работы с динамическим объектом память необходимо освободить:

b=new First; ... delete b;

Объект, созданный таким способом, называют динамическим. Значения полей неинициализированных статических и динамических объектов или массивов объектов задают в процессе дальнейшей работы с объектами: защищенных и скрытых – только в методах класса, а общедоступных – в методах класса или непосредственным присваиванием в программе. При отсутствии в классе конструктора и защищенных protected или скрытых private полей для объявления инициализированных объектов используют оператор инициализации, применяемый при создании инициализированных структур, например:

First a = {'A',3,4}, c[4] = {{'A',1,4},{'B',3,5},{'C',2,6},{'D',1,3}};

Инициализирующие значения при этом должны перечисляться в порядке следования полей в описании класса.

Пример. Различные способы инициализации общедоступных полей объекта.

```
#include <locale.h>
#include <string.h>
#include <stdio.h>
#include <conio.h>
class sstro
{
public:
char str1[80]; int x,y;
void set_str(char *vs) // инициализация полей
{
strcpy(str1,vs); x=0; y=0;
}
void print(void) // вывод содержимого полей
{
printf("x=%5d y=%5d str: ",x,y);
puts(str1);
}
};
void main()
{
setlocale(0,"russian");
sstro aa = {"Строка",200,400}; // инициализированный объект
sstro bb,cc; // неинициализированные объекты
bb.x=200; // инициализация посредством прямого обращения
bb.y=150;
strcpy(bb.str1,"Строка");
```

```
cc.set_str("Строка"); // вызов инициализирующего метода
aa.print();
bb.print();
cc.print(); _
getch();
}
```

Конструктор – метод класса, который автоматически вызывается при выделении памяти под объект.

По правилам C++ конструктор имеет то же имя, что и класс, не наследуется в производных классах, может иметь аргументы, но не возвращает значения, может быть параметрически перегружен.

Конструктор определяет операции, которые необходимо выполнить при создании объекта. Традиционно такими операциями являются инициализация полей класса и выделение памяти под динамические поля, если такие в классе объявлены. Явный вызов конструктора не возможен, что в некоторых случаях усложняет создание инициализированных объектов.

При освобождении объектом памяти автоматически вызывается другой специальный метод класса – деструктор. Имя деструктора по аналогии с именем конструктора, совпадает с именем класса, но перед ним стоит символ «~» («тильда»). Деструктор определяет операции, которые необходимо выполнить при уничтожении объекта. Обычно он используется для освобождения памяти, выделенной под динамические поля объекта данного класса конструктором, и при необходимости может быть объявлен виртуальным. Деструктор не возвращает значения, не имеет параметров и не наследуется производными классами. Класс может иметь только один деструктор или не иметь ни одного. В отличие от конструктора деструктор может вызываться явно.

Момент уничтожения объекта, а, следовательно, и автоматического вызова деструктора определяется типом памяти, выбранным для размещения объекта: локальная, глобальная, внешняя и т. д. Если программа завершается с использованием функции exit(), то вызываются деструкторы только глобальных объектов. При аврийном завершении программы, использующей объекты некоторого класса, функцией abort() деструкторы объектов не вызываются.

Пример. Создание, инициализация и уничтожение объекта при наличии в классе конструктора и деструктора.

```
#include <locale.h>
#include <iostream.h>

using namespace std;
class Num
{
int n;
public:
 Num(int an){ cout<<"Конструктор"<<endl; n=an;}
```

```
~Num() {cout<<"Деструктор"<<endl;}
};
void main(int argc, char* argv[])
{
 setlocale(0,"russian");
Num N(56);
system("pause"); }
};
```

**Контрольные вопросы.**

1. Что такое «класс»? В каких случаях используется эта конструкция?
2. Что такое «объект»?
3. Какие диаграммы используют для описания взаимодействия классов и объектов? 4. Какой синтаксис имеет объявление класса и объектов этого класса?
4. Объясните различие между описанием класса с конструктором и без конструктора?
5. Как можно инициализировать объекты класса, описанного без конструктора?

**Лабораторная работа №4 . Стандартная библиотека языка Си++.**

**Цель работы:** закрепить практические навыки работы со стандартными классами С++, научиться правильно использовать различные классы, научиться составлять программы решения задач с использование классов.

**Общие сведения.**

## Объектная система игры (игровой "движок")

Каким бы хорошим не был "движок" графический — сама игра работает благодаря "движку" игровому т.е. той части кода игры, которая отвечает за хранение и обработку разнообразных игровых объектов. В этой статье я бы хотел описать некоторую ретроспективу видов игровых "движков", начиная со времен "древних" игровых систем.

Сразу оговорюсь, что намеренно не включаю в этот обзор последние разработки по игровым "движкам" (в т.ч. свои) т.к. рассчитываю... описать их позднее, после того как вы, дорчитатели, оцените и обсудите эту статью.

Любая диалоговая программа, не исключая, конечно же, и игры, строиться на основе цикла (обычно — одного цикла). Краткая схема такого цикла выглядит так:

```
Инициализация
Цикл:
{
    Ввод
    Обработка
    Вывод
}
```

На этапе Инициализации производится подготовка к работе цикла — задаются начальные состояния объектов игры, запускается графическая, звуковая и другие подсистемы программы игры. Кроме того, инициализация может встречатся и внутри цикла (на этапеОбработки), если требуется, например, загрузить новый уровень или восстановить работоспособность графической подсистемы после переключения на другую задачу по Alt+Tab (в Windows

На этапе Ввода произовится чтение новых значений управляющих воздействий от игрока (клавиатура, мышь, джойстик, VR-костюм и др.) т.е. считывается новое положение курсора мыши, новые состояния клавиш на клавиатуре и др. Эти новые значение обрабатываются и записываются в память программы, чтобы позже (на других этапах) их смогли считать другие части программы и проверить — нажата ли клавиша, где рисовать курсор мыши и т.п.

На этапе Обработки происходит свмое интересное — здесь "думают" монстры, обрабатывается физика, а также создаются и уничтожаются разнообразные игровые объекты — т.е. работает игровой "движок". Кроме

того, очень важно сделать так, чтобы независимо от скорости выполнения всего цикла программы на разных компьютерах, объекты игры всегда работали именно с той скоростью, на которую раситаны по замыслу создателей игры.

Наконец, на этапе Вывода в дело вступает графический "движок", звуковая подсистема и другие системы, каждая из которых использует т.н. систему ресурсов (общую, локальную или комбинированную), чтобы загрузить текстуры, спрайты, звуки и т.п. ресурсы с медленного носителя информации (например, жесткого диска) в оперативную память и других подобных действий. То, что в результате появляется на экране, играется через колонки или наушники и т.д. определяется в том числе и объектами игры — если монстр ранен, то выводится моделька с другой текстурой и т.п.

Таким образом видно, что для игрового "движка" необходимо задать:

а) какие бывают объекты (типы объектов)

б) как они создаются/уничтожаются, обрабатываются, взаимодействуют

в) как они получают ввод от игрока (интерфейс системы ввода)

г) как они отображают себя для игрока (интерейс системы вывода)

Ретроспектива видов игровых "движков":

Хранение и обработка отдельных единичных объектов

Итак, самым простым и понятным способом задать объекты игры в программе является объявление отдельных переменных.

Пример:

```
    int playerX, playerY, playerHealth, playerScore;
int monsterX, monsterY, monsterHealth, monsterAnger;

playerX = playerY = 10;
playerHealth = 100;
playerScore = 0;

monsterX = 190; monsterY = 90;
monsterHealth = 1000;
monsterAnger = 10;
```

После того, как переменные были объявлены и инициализированы начальными значениями, их уже можно использовать для отрисовки игрока и монстра на экране, движения игрока при нажатии клавиш управления, движении монстра по какому-либо алгоритму. Это самый простой способ задать игровые объекты.

Однако для большего удобства лучше задать эти переменные как поля структур и создать переменные с типом этих структур.

Примерно так:

```
    struct player_t
{
    int x,y;
    int health;
    int score;
};
```

```
struct monster_t
{
    int x,y;
    int health;
    int anger;
};

player_t  player;
monster_t monster;

player.x = player.y = 10;
player.health = 100;
player.score = 0;

monster.x = 190; monster.y = 90;
monster.health = 2500;
monster.anger = 100;
```

### *2. Множества однотипных объектов*

Если задавать игровые объекты с помощью структур, то можно легко "тиражировать" однотиные объекты в игре, например, задавая массив.

Пример:

```
const int max_monsters = 10;
monster_t monsters[ max_monsters ];

for(int j = 0; j < max_monsters; j++)
{
   monsters[j].x = i * 10;
   monsters[j].y = 90 - i;
   monsters[j].health = 1000 + (random() % 1000);
   monsters[j].anger = 10;
}
```

Теперь вместо одиночной проверки на пересечение игрока с монстром будет цикл, в котором проверяется пересечение каждого монстра с игроком. Таким образом можно задать любое количество типов объектов и создавать любое требуемое количество однотипных объектов в игре.

Однако проблемы начинаются тогда, когда количество типов объектов становиться большим, а от игры требуется еще большая реалистичность. Код просто напросто превращается в нечто такое:

```
    //Этап обработки объектов
    ControlPlayer();                        //обрабатываем управление игрока
    ThinkMonsters();                        //монстры "думают"
    MovePlayer();                           //двигает игрока
    MoveMonsters();                         //двигаем монстров
    CheckCollisions_Player_Monsters();      //проверяем, если монстр поймал
игрока
    MoveMissles();                          //двигаем ракеты
```

```
    CheckCollisions_Player_Missles();       //проверяем, если ракета попала в
игрока (ракета взорвется)
    CheckCollisions_Monsters_Missles();     //проверяем, какие ракеты в каких
монстров попали (ракеты взрываюся)
    AnimateExplosions();                    //анимируем взрывы, которые
оставляют ракеты
    CheckCollisions_Player_Explosions();    //проверяем, если игрока задел
взрыв (уменьшаем здоровье и т.д.)
    CheckCollisions_Monsters_Explosions(); //проверяем, каких монстров какие
взрывы задели (...)
    ...
```

И так далее — чем больше типов объектов, тем сложнее код.

3. Единая структура для всех объектов

Чтобы упростить программирование объектов, можно задать одну структуру для всех объектов игры. Тогда внутри этой структуры необходимо будет хранить код типа объекта, чтобы проверив этот код можно было однозначно сказать — что же, собственно, задает этот объект (игрока, монстра, ракету, взрыв и др.).

```
struct game_object_t
{
    int kind; //это код типа объекта
    int x,y;
    int health;
    int score;
    int anger;
};

enum game_object_kinds
{
    kind_Player,
    kind_Monster,
    kind_Missle,
    kind_Explosion,
};
```

Также видно, что в таком случае приходится объявлять все виды полей, которые могут потребоваться объекту, если его поле kind будет принимать то или иное значение.

Чтобы избежать таких затрат памяти можно объединить поля разных объектов в безымянные union'ы.

```
struct game_object_t
{
    int kind; //это код типа объекта
    int x,y;
    int health;
    union
    {
        int score;
        int anger;
    };
};
```

Объединение полей потребует некоторых дополнительных усилий со стороны программиста, но это надо будет сделать только один раз, а потом всего лишь подправлять и дополнять единую структуру объектов.

Однако где-же здесь упрощение при обработке объектов?

А упрощение состоит в том, что мы можем написать обработчик объектов, в котором будет проверяться поле kind объекта и будут выполняться соответствующие именно этому типу объектов действия.

Пример:

```
void Think( game_object_t *p )
{
    switch(p->kind)
    {
        case kind_Player:
            ...
            break;
        case kind_Monster:
            ...
            break;
        case kind_Missle:
            ...
            break;
        case kind_Explosion:
            ...
            break;
        default:
            //ошибка - тип объекта неверен
            ...
            break;
    }
}
```

Однако есть способ получше — мы может добавить в единую структуру игровых объектов поля — указатели на процедуры обработки этого объекта.

Пример:

```
    struct game_object_t
{
    int kind; //это код типа объекта
    int x,y;
    int health;
    union
    {
        int score;
        int anger;
    };
    void (* Think)( game_object_t *self );
};
```

Затем в программе, при инициализации объекта надо будет заполнить это поле определенным значением, либо нулем (ноль — отсутствие обрабочика)

```
game_object_t objects[ max_game_objects ];
...
void Monster_Think( game_object_t *self )
```

```
{
    //здесь "думает" монстр
}
...
void Missle_Think( game_object_t *self )
{
    //здесь "думает" ракета
}
...
   objects[0].Think = Monster_Think;
   objects[1].Think = Missle_Think;
...
```

Теперь можно вызывать обработчики "дум" объектов единообразным образом.

```
for(int j = 0; j < num_game_objects; j++)
{
    void (* proc)() = objects[j].Think;
    if(proc) proc( &objects[j] );
}
```

Причем для каждого объекта будет вызван именно тот обработчик, который был задан ему в поле Think.

Может возникнуть вопрос — а зачем после этого нужно поле kind объекта? Очень просто — чтобы можно было идентифицировать объект по номеру его типа. Для чего? Например, для того, чтобы можно было сохранять игровые объекты в файл (ведь в файле нельзя хранить адрес обработчика "думания" т.к. этот адрес может поменяться при новом запуске программы, номер же не изменится пока мы этого не захочем).

### *4. Класс объекта, наследование и виртуальные методы*

Следующей ступенью на пути совершенстования игрового "движка" будет решение заменить структуру на класс т.к. в языке C++ классы лучше соответствуют ООП (объектно-ориентированному подходу в программировании), чем "чистые" структуры в стиле pure C.

Пример:

```
class GameObject
{
    public:
        int x,y;
        int health;
        virtual int GetType() { return -1; }
        virtual void Think() {}
};
```

В этом примере нет описания собственно типов объектов, а описывается т.н. _базовый класс_, от которого могут _наследоваться_ классы-потомки.

Примеры:

```
class GameObject_Player: public GameObject
{
    private:
        int score;

    public:
        int GetType() { return kind_Player; }
        void Think()  {}
};

class GameObject_Monster: public GameObject
{
    private:
        int anger;

    public:
        int GetType() { return kind_Monster; }
        void Think()  {}
};
```

Чтобы любой объект класса-потомка GameObject "подумал" нужно все лишь написать:

```
GameObject *p = ...; //здесь подставляется адрес конкретного объекта
                     //в том числе можно подставить адрес объекта,
                     //класс которого - класс-потомок класса GameObject
p->Think();          //здесь вызывается метод Think того класса, к
                     //которому пренадлежит объект
```

Чтобы получить код типа объекта можно написать:

```
... = p->GetType();
```

Таким образом в C++ уже встроены те средства, которые помогают в создании системы игровых объектов.

Как правило в классе-родителе (таком как GameObject) не должно быть никаких полей данных — только объявление общих для всех объектов _виртуальных_ методов.

Пример:

```
class GameObject
{
    public:
        virtual int GetType() { return -1; }

        virtual void Think  ()                                {}
        virtual void Push   ( float force_x, float force_y ) {}
        virtual void Damage ( float damage )                  {}
        ...
};
```

Таким образом, если надо толнуть объект, то достаточно лишь вызвать его метод Push(...), независимо от того, к какому классу-потомку пренадлежит этот объект:

```
      GameObject *p = GetAnyDerivedClassObject();
p->Push( force_x, force_y );
```

Если в классе-потомке метод Push(...) не будет описан, то будет вызван метод класса-родителя т.е. в данном случае пустой метод.

5. Вынос физики за рамки объекта

После серии опытов в написании системы игровых объектов, можно прийти к выводу, что хранить координаты и размеры объекта внутри самого объекта — неправильно т.к. эти данные требуются, чтобы проверять столкновения объектов и только при обнаружении столновения должен вызываются соответствующий метод объекта (Push, Damage или др.).

Можно реализовать игровой "движок" так, что объекты не будут "знать" своего точного положения и размеров в том "мире", в котором они находяться. Эта информация будет храниться отдельно от объектов самим "миром". "Мир" будет заведовать физикой объектов т.е. находить какие объекты пересекаются и вызывать соответствующие методы объектов. Если объекту потребуется узнать какую-то физическую информацию, создать другой объект или подвинуть себя — он должен будет вызвать соответствующий метод "мира".

Таким образом "мир" — тоже объект, хотя и непохожий на те объекты, что "храняться" внутри него. Если еще немного подумать, то можно реализовать более универсальную систему игровых объектов, в которой "мир" сам будет обычным игровым объектом (только более сложно устроенным внутри) и тогда, к примеру, поместить в объект "сундук" объекты-предметы будет проще простого (а попробуйте это сделать без организации иерархии объектов).

## Лабораторная работа №5. Создание собственных классов.

### Цель работы:

изучение средств языка C++, используемых при наследовании классов.

### Общие сведения

Наследованием называют конструирование новых более сложных производных классов (классов-потомков) из уже имеющихся базовых классов (классов-родителей) посредством добавления полей и методов. Это – эффективное средство расширения функциональных возможностей существующих классов без их перепрограммирования и повторной компиляции существующих программ. По определению компонентами производного класса являются: компоненты базового класса, за исключением конструктора, деструктора и компонентной функции, переопределяющей операцию «присваивания» (=) (см. раздел 5.4); компоненты, добавляемые в теле производного класса.

В функциональном смысле производные классы являются более мощными по отношению к базовым классам, так как, включая поля и методы базового класса, они обладают еще и своими компонентами. Ограничение доступа к полям и функциям базового класса при наследовании осуществляется с помощью специальных описателей, определяющих вид наследования: class : {}; где вид наследования определяется ключевыми словами: private, protected, public. Видимость полей и функций базового класса из производного определяется секцией, в которой находится объявление компонента и видом наследования (см. таблицу 1).

Таблица 1. Видимость компонентов базового класса в производном

| Вид наследования | Объявление компонентов в базовом классе | Видимость компонентов в производном классе |
|---|---|---|
| private | private | не доступны |
| | protected | private |
| | public | private |
| protected | private | не доступны |
| | protected | protected |
| | public | protected |
| public | private | не доступны |
| | protected | protected |
| | public | public |

Если вид наследования явно не указан, то по умолчанию принимается private. Однако хороший стиль программирования требует, чтобы в любом случае вид наследования был задан явно.

В языке C++ конструкторы и деструкторы базового класса в производных классах не наследуются. Однако если базовый класс содержит хотя бы один конструктор и деструктор, то производный класс также должен включать собственные конструктор и деструктор. При этом C++ поддерживает определенные правила взаимодействия между этими компонентами базовых и производных классов.

При создании объектов производного класса предусмотрен автоматический вызов конструктора базового класса для инициализации его полей. Однако следует помнить, что по умолчанию осуществляется вызов конструктора базового класса без параметров. Если такой конструктор в базовом классе отсутствует, то компилятор выдает сообщение об ошибке error C2512.

Поскольку явный вызов конструктора базового класса в программе невозможен, чтобы передать этому конструктору аргументы для инициализации полей базового класса, следует:

- добавить соответствующие параметры к собственным параметрам конструктора• производного класса;
- вызвать конструктор базового класса в списке инициализации конструктора• производного класса, передав ему соответствующие аргументы.

```
class A
{
int x;
 public:
A(int ax):x(ax){} // конструктор базового класса
};
class B
{
int y;
public:
B(int ax,int ay):A(ax),y(ay){} // конструктор производного класса
};
```

При этом соблюдается строгий порядок конструирования полей базового и производного классов: не зависимо от порядка указания в списке инициализации сначала вызывается конструктор базового класса, а затем – конструкторы полей, объявленных в производном классе.

Пример 2.2. Проектирование классов с использованием наследования (классы Целое число и Вещественное число). Пусть требуется разработать классы для реализации объектов Целое число и Вещественное число. Объект Целое число должен хранить длинное целое в десятичной записи и уметь выводить его значение. Объект Вещественное число должен хранить вещественное число, задаваемое в виде ccccc.dddddd, и его символьное представление. Он также должен уметь выводить свое значение на экран.

Для обоих объектов необходимо предусмотреть возможность инициализации как в момент объявления переменной, так и в процессе функционирования. Поскольку вещественное число включает длинное целое как целую часть, класс для его реализации можно наследовать от класса, реализующего длинное целое число.

```
#include <locale.h>
#include <stdlib.h>
#include <iostream.h>
#include <string.h>
 using namespace std;
typedef unsigned long dlong;
class
class Tlong // Класс Целое число
{
public:
dlong num; // числовое поле класса
Tlong(){} // неинициализирующий конструктор
Tlong(dlong an):num(an){} // конструктор
 ~Tlong(){} // деструктор
void print(void) // вывод значения поля
{ cout<<" Целое число : "<<num<<endl;}
void setnum(dlong an) //инициализация поля
{ num=an;}
};
```

```
class Treal // Класс вещественное число
{
public:
dlong drob; // дробная часть числа
char *real;
Treal (){real=NULL} // конструктор без параметров
Treal (char *st) : TLong()  // инициализирующий конструктор
{ setnumv (st); }
 ~ Treal (){} // деструктор
{ if (real!=NULL) delete [] real;}
void printr(); // вывод  вещественного числа
 void setnumv (char *st); //инициализация  полей класса
};
void Treal:: setnumv (char *st)
{
int l=strlen(st);
char *ptr;
real=new char[l+1];
strcpy(real,st);
ptr=strchr(real,'.');
*ptr='\0'; drob=dlong(atol(ptr+1));
num=dlong(atol(real));
*ptr='.';
}

void Treal::printr()
{
cout<<"Вещественное число: "<<real<<endl;
cout<<"Целая часть: "; print();
cout<<"Дробная часть: "<<drob<< endl;
}
void main ()
{
setlocale(0,"russian");
Treal a("456789.1234321"), // объект производного класса
*pa=new Treal("456789.1234321"), // указатель
mask[3]= // инициализированный массив объектов
{
Treal("1748.5932"),
Treal("4567.34321"),
Treal("18689.9421")
};
a.printr();
pa->printr();
delete pa;
for(int i=0;i
{ cout<<"Элемент массива "<<(i+1)<<": "<<endl;
mask[i].printr();
}
 system("pause");
 }
```

## Контрольные вопросы

Что такое «наследование»? В каких случаях используется этот механизм?

2. Как показать наследование на диаграмме объектом и на диаграмме классов?

3. Как описывается наследование классов в программе?

4. В чем заключаются особенности работы с конструкторами базового и производного классов?

5. В каких случаях фиксируются ошибки при выполнении конструкторов производных классов?

**Лабораторная работа №6. Перегрузка операций, умные указатели.**

**Цель работы:** познакомить с понятием *"множество"* в языке программирования Pascal; выработать навыки работы со структурой данных множество.

**Общие сведения**

Ключевое слово operator предоставляет объявление функции, указывающей что operator-symbol означает при его применении к экземплярам класса. Это предоставляет оператору многозначность или "перегружает" его. Компилятор различает разные виды одного и того же оператора, изучая типы его операндов.

type operator operator-symbol ( parameter-list )

Заметки

Можно переопределять большинство встроенных операторов глобально или в классе. Перегруженные операторы реализуются в виде функции.

Перегруженный оператор имеет имя operatorx, где x означает оператор из следующей таблицы. Например, для перегрузки оператора сложения необходимо определить функцию operator+. Аналогично, для сложения/присвоения += необходимо определить функцию operator+=.

Переопределяемые операторы

| operator | name | Тип данных |
|---|---|---|
| , | запятая (разделитель элементов) | binary |
| ! | Логическое НЕ | Унарные |
| != | Неравенство | binary |
| % | Остаток от деления | binary |
| %= | Назначение остатка от деления | binary |
| & | Побитовое И | binary |
| & | Взятие адреса | Унарные |
| && | Логическое И | binary |
| &= | Назначение побитового И | binary |
| ( ) | Вызов функции | — |
| ( ) | Оператор приведения типа | Унарные |
| * | Умножение | binary |
| * | Разыменование указателя | Унарные |
| *= | Присваивание умножения | binary |
| + | Сложение | binary |
| + | Унарный плюс | Унарные |
| ++ | Инкремент1 | Унарные |

| | | |
|---|---|---|
| += | Присваивание сложения | binary |
| – | Вычитание | binary |
| – | Унарное отрицание | Унарные |
| — | Декремент | Унарные |
| –= | Присваивание вычитания | binary |
| –> | Выбор члена | binary |
| –>* | Выбор указателя на член | binary |
| / | Деление | binary |
| /= | Присваивание деления | binary |
| < | Меньше | binary |
| << | Сдвиг влево | binary |
| <<= | Сдвиг влево с присваиванием | binary |
| <= | Меньше или равно | binary |
| = | Назначение | binary |
| == | Равенство | binary |
| > | Больше | binary |
| >= | Больше или равно | binary |
| >> | Сдвиг вправо | binary |
| >>= | Сдвиг вправо с присваиванием | binary |
| [ ] | Операция взятия индекса | — |
| ^ | Исключающее ИЛИ | binary |
| ^= | Исключающее ИЛИ/присваивание | binary |
| \| | Побитовое включающее ИЛИ | binary |
| \|= | Назначение побитового включающего ИЛИ | binary |
| \|\| | Логическое ИЛИ | binary |
| ~ | Дополнение до единицы | Унарные |
| delete | Delete | — |
| new | New | — |
| conversion operators | операторы преобразования | Унарные |

**Пример**

В следующем примере перегружен оператор +, что позволяет сложить два комплексных числа и получить результат.

```
// operator_overloading.cpp
// compile with: /EHsc
#include <iostream>
using namespace std;
```

```
struct Complex {
   Complex( double r, double i ) : re(r), im(i) {}
   Complex operator+( Complex &other );
   void Display( ) {   cout << re << ", " << im << endl; }
private:
   double re, im;
};

// Operator overloaded using a member function
Complex Complex::operator+( Complex &other ) {
   return Complex( re + other.re, im + other.im );
}

int main() {
   Complex a = Complex( 1.2, 3.4 );
   Complex b = Complex( 5.6, 7.8 );
   Complex c = Complex( 0.0, 0.0 );

   c = a + b;
   c.Display();
}
```

## Контрольные вопросы

1. Что такое перегруженный оператор?

**Лабораторная работа №7. Модульное тестирование ПО. Разработка в стиле TDD**

**Цель работы:** познакомиться с понятиями "процедура" и "функция" в языке программирования Pascal, рассмотреть их сходства и различия, закрепить практические навыки работы с системой TURBO Pascal на примере реализации алгоритмов при помощи процедур и функций, научиться применять метод последовательной детализации в практическом программировании; применять процедуры и функции при решении задач.

**Общие сведения**

Новый проект консольного приложения . В нём нет ни одной строчки кода. Надо придумать, как вообще будет работать приложение. Внимание, начало! Создаем тест, который описывает 4-ое требование:

```
public class ReporterTests
{
[Fact]
  public void ReturnNumberOfSentReports()
{
var reporter = new Reporter();
    var reportCount = reporter.SendReports();
    Assert.Equal(2, reportCount);
}
}
```

* This source code was highlighted with Source Code Highlighter.

Класс Assert проверяет равно ли количество отосланных отчетов 2. Тест запускается консольной утилитой xUnit, либо каким-нибудь плагином к Visual Studio.

Только что мы спроектировали API нашего приложения. Мы будем использовать объект Reporter с функцией SendReports. Функция SendReports возвращает количество отправленных отчетов, это показывает тест с помощью утверждения Assert.Equal. Если переменная reportCount не будет равена 2, то тест не пройдет.

На этом первый этап проектирования закончился, переходим к кодированию. Напишем минимум кода, чтобы этот тест сработал.

В проекте на данный момент есть только один класс ReporterTests. Пора создать тестируемый класс Reporter. Добавляем в проект объект Reporter и создаем у него пустую функцию SendReports. Для того, чтобы тест прошёл, функцияSendReports должна вернуть цифру 2. Пока не понятно, как задать начальные условия в объекте Reporter, чтобы функцияSendReports вернула цифру 2.

Возвращаемся к проектированию. Я думаю, что у меня будет отдельный класс для создания отчётов, и класс для отправки отчётов. Сам

объект Reporter будет управлять логикой взаимодействия этих классов. Назовем первый объектIReportBuilder, а второй – IReportSender. Попроектировали, пора написать код:

```
[Fact]
public void ReturnNumberOfSentReports()
{
IReportBuilder reportBuilder;
IReportSender reportSender;
Var reporter = new Reporter(reportBuilder, reportSender);
Var reportCount = reporter.SendReports();
Assert.Equal(2, reportCount);
}
```

* This source code was highlighted with Source Code Highlighter.

Как будут работать классы, реализующие эти интерфейсы, сейчас не имеет значения. Главное, что мы можем сформировать IReportBuilder'ом все отчёты и отправить их с помощью IReportSender'a.

Вопрос: почему стоит использовать интерфейсы IReportBuilder и IReportSender, а не создать конкретные классы?

Реализовать объект для создания отчётов и объект для отправки отчётов можно по-разному. Сейчас удобнее скрыть будущие реализации этих классов за интерфейсами.

Вопрос: Как задать поведение объектов, с которыми взаимодействует наш тестируемый класс?

Вместо реальных объектов, с которыми взаимодействует наш тестируемый класс удобнее всего использовать заглушки или mock-объекты. В текущем приложении мы будем создавать mock-объекты с помощью библиотеки Moq.

```
[Fact]
public void ReturnNumberOfSentReports()
{
var reportBuilder = new Mock<IReportBuilder>();
var reportSender = new Mock<IReportSender>();

// задаем поведение для интерфейса IReportBuilder
// Здесь говорится: "При вызове функции CreateReports вернуть List<Report> состоящий из 2х объектов"
  reportBuilder.Setup(m => m.CreateRegularReports())
    .Returns(new List<Report> {new Report(), new Report()});


var reporter = new Reporter(reportBuilder.Object, reportSender.Object);
```

```
var reportCount = reporter.SendReports();
Assert.Equal(2, reportCount);
}
```

* This source code was highlighted with Source Code Highlighter.

Запускаем тест – он не проходит, потому что мы не реализовали функцию SendReports. Программируем самую простую из возможных реализаций:

```
public class Reporter
{
private readonly IReportBuilder reportBuilder;
private readonly IReportSender reportSender;

  public Reporter(IReportBuilder reportBuilder, IReportSender reportSender)
{
this.reportBuilder = reportBuilder;
this.reportSender = reportSender;
}

  public int SendReports()
{
return reportBuilder.CreateRegularReports().Count;
}
}
```

* This source code was highlighted with Source Code Highlighter.

Запускаем тест и он проходит. Мы реализовали 4-ое требование. При этом записали его в виде теста. Таким образом, мы составляем документацию нашей системы. Как показала практика – эта документация самая актуальная в любой момент времени и никогда не устаревает. Идем дальше.

Вопрос: Есть ли стандартный шаблон для написания теста?

Да. Он называется Arrange-Act-Assert (AAA). Т.е. тест состоит из трех частей. Arrange (Устанавливаем) – производим настройку входных данных для теста. Act (Действуем) – выполняем действие, результаты которого тестируем. Assert (Проверяем) – проверяем результаты выполнения. Я подпишу соответствующие этапы в следующем тесте.

Теперь займёмся первым требованием – отправлением отчётов. Тест будет проверять, что все созданные отчёты отправлены:

```
[Fact]
public void SendAllReports()
```

```
{
// arrange
var reportBuilder = new Mock<IReportBuilder>();
var reportSender = new Mock<IReportSender>();

reportBuilder.Setup(m => m.CreateRegularReports())
.Returns(new List<Report> {new Report(), new Report()});

var reporter = new Reporter(reportBuilder.Object, reportSender.Object);

// act
reporter.SendReports();

// assert
reportSender.Verify(m => m.Send(It.IsAny<Report>()), Times.Exactly(2));
}
```

* This source code was highlighted with Source Code Highlighter.

Вопрос: Надо ли писать тесты для всех объектов приложения в одном тестовом классе?

Очень нежелательно. В этом случае тестовый класс разрастется до огромных размеров. Лучше всего на каждый тестируемый класс создать отдельный файл с тестами.

Запускаем тест, он не проходит, потому что мы не реализовали отправку отчётов в функции SendReports. На этом, как обычно мы проектировать заканчиваем и переходим к кодированию:

```
public int SendReports()
{
IList<Report> reports = reportBuilder.CreateRegularReports();

foreach (Report report in reports)
{
reportSender.Send(report);
}
return reports.Count;
}
```

* This source code was highlighted with Source Code Highlighter.

Запускаем тесты – оба проходят. Мы реализовали ещё одно бизнес-требование. К тому же, запустив оба теста мы убедились, что не сломали функциональность, которую делали 5 минут назад.

Вопрос: Как часто надо запускать все тесты?

Чем чаще, тем лучше. Любое изменение в коде может неожиданно для вас отразится на других частях системы. Особенно, если этот код писали не Вы. В идеале все тесты должны запускаться автоматически системой интеграции (Continuous Integration) при каждой сборке проекта.

Вопрос: Как протестировать приватные методы?

Если вы дочитали до этого момента, то уже понимаете, что раз сначала пишутся тесты, а уже потом код, значит весь код внутри класса будет по-умолчанию протестирован.

Пора подумать о том, как реализовывать третье требование. С чего начнем? Нарисуем UML-диаграммы или просто помедитируем сидя в кресле? Начнём с теста! Запишем 3-е бизнес-требование в коде:

```
[Fact]
public void SendSpecialReportToAdministratorIfNoReportsCreated()
{
var reportBuilder = new Mock<IReportBuilder>();
var reportSender = new Mock<IReportSender>();

reportBuilder.Setup(m => m.CreateRegularReports()).Returns(new List<Report>());
reportBuilder.Setup(m => m.CreateSpecialReport()).Returns(new SpecialReport());
var reporter = new Reporter(reportBuilder.Object, reportSender.Object);
reporter.SendReports();
reportSender.Verify(m => m.Send(It.IsAny<Report>()), Times.Never());
reportSender.Verify(m => m.Send(It.IsAny<SpecialReport>()), Times.Once());
}
```

* This source code was highlighted with Source Code Highlighter.

Запускаем и убеждаемся, что тест не проходит. Теперь наши усилия направлены на починку этого теста. Здесь проектирование как обычно заканчивается и мы возвращаемся к программированию:

```
public int SendReports()
{
IList<Report> reports = reportBuilder.CreateRegularReports();
if (reports.Count == 0)
{
reportSender.Send(reportBuilder.CreateSpecialReport());
}
```

```
foreach (Report report in reports)
{
reportSender.Send(report);
}
return reports.Count;
}
```

* This source code was highlighted with Source Code Highlighter.

Запускаем тесты – все 3 теста проходят. Мы реализовали новую функции и не сломали старые. Это не может не радовать!

Вопрос: Как узнать какой код уже протестирован?

Покрытие кода тестами можно проверить с помощью различных утилит. Для начала могу посоветовать PartCover.

Вопрос: Надо ли стремиться покрыть код тестами на 100%?

Нет. Это потребует слишком больших усилий на создание таких тестов и ещё больше на их поддержку. Нормальное покрытие колеблется от 50 до 90%. Т.е. должна быть покрыта вся бизнес-логика без обращений к базе данных, внешним сервисам и файловой системе.

Второе требование я предлагаю реализовать вам самим и поделиться в комментариях финальной частью функции SendReports и вашего теста. Вы ведь сначала напишете тест, так?

Вопрос: Как же мне протестировать взаимодействие с базой данных, работу с SMTP-сервером или файловой системой?

Действительно, тестировать это нужно. Но это делается не модульными тестами. Потому что модульные тесты должны проходить быстро и не зависеть от внешних источников. Иначе вы будете запускать их раз в неделю. Более подробно об этом написано в статье «Эффективный модульный тест».

Вопрос: Когда я могу применять TDD?

TDD можно применять для создания любого приложения. Очень удобно его применять, если вы изучаете возможности новой библиотеки или языка программирования. Особых границ в применении нет. Возможны неудобства с тестированием многопоточных и других специфических приложений.

**Лабораторная работа №8. Композиция, наследование, полиморфизм.**

**Цель работы:** изучить реализацию на языке C++ отношений между классами: агрегации, наследования, зависимости

**Общие сведения**

Ознакомиться с отношениями между классами: агрегацией, наследованием, зависимостью, а также с механизмом позднего связыва- ния. 2. Путем модификации программ, разработанных в лабораторных работах № 1,

Разработать программу такую, чтобы в ней были опре- делены несколько классов, реализующих понятие геометрической фи- гуры в графической системе:

абстрактный класс «Фигура», содержащий чисто виртуальные функции;

класс «Закрашенный», позволяющий задать кисть, ее параметры и, возможно, осуществить закраску;

класс «Фигура-контур» – потомок класса «Фигура»;

класс «Закрашенная фигура» – потомок класса «Фигура-контур»,

класс «Закрашенный» при этом использовать либо как второго родителя (множественное наследование), либо как часть класса «Закрашенная фигура» (агрегация);

класс «Комбинированная фигура», реализующий две вложенные фигуры с закраской между ними.

Реализацию классов поместить в отдельный файл.

Разработать функцию, демонстрирующую поведение разработанных классов, включая демонстрацию механизма позднего связывания.

Подготовить текстовые файлы с разработанной программой, оттранслировать их, собрать и выполнить программу с учетом требова- ний операционных систем и программных оболочек, в которых эта программа выполняется. При необходимости исправить ошибки и вновь повторить технологический процесс решения задачи.

Оформить отчет по лабораторной работе. Отчет должен содер- жать постановку задачи, описание разработанных классов, алгоритм, текст разработанной программы и результаты тестирования.

Защитить лабораторную работу, ответив на вопросы препода- вателя.

Рассмотрим использование механизма наследования для реализа- ции геометрических фигур в графической системе.

Сначала опишем наиболее общий класс «Фигура», в котором будут только общие свойства всех фигур.

```
struct Point;
class Shape {
protected: Point Center; // центр фигуры . . .
public: virtual void draw ( ) = 0; // чисто виртуальная функция . . .
};
```

Описание класса содержит защищенную (protected) часть, элементы (атрибуты и служебные функции) в этой части видимы не только самому классу и его друзьям (как в закрытой части ), но и его наследникам.

Описание virtual означает, что функция является виртуальной – замещается в классе, производном от данного. Функция, интерфейс вы- зова которой может быть определен, а реализация – нет, объявляется чисто виртуальной, для чего используется синтаксис «= 0».

Для определения класса фигуры-контура «Окружность» мы долж- ны сказать, что она является фигурой (Shape), и указать особые свой- ства (в том числе определить чисто виртуальные функции).

```
struct Color;
class Circle: public Shape {
protected: int Radius; // радиус
Color col; // цвет контура . . .
public:
Circle (Point cntr, int rds, Color cl); // конструктор
void draw ( ); // определение чисто виртуальной функции . . .
};
```

Класс «Закрашенная окружность» определим как наследника классов «Окружность» и «Закрашенный».

```
class Filled;
class FilledCircle: public Circle, public Filled {
public:
void draw ( );// переопределение чисто виртуальной функции . . .
};
```

Класс «Комбинированная фигура», реализующий две вложенные фигуры с закраской между ними, определим как наследника класса «Закрашенная окружность».

```
class CombiCircle: public FilledCircle {
protected:
FilledCircle fc; // внутренняя фигура . . .
public:
void draw ( );// переопределение чисто виртуальной функции . . .
 };
```

При определении классов их суперклассы были объявлены нами как открытие (public). В результате открытые и защищенные члены суперкласса становятся открытыми и защищенными членами подкласса. Таким образом, подкласс считается также и подтипом, т. е. обязуется выполнять все обязательства суперкласса. В частности, он обеспечивает совместимое с суперклассом подмножество интерфейса и обладает неразличимым, с точки зрения клиентов суперкласса, поведением.

С наследованием связан особый тип полиморфизма – включение (чистый полиморфизм). Данный тип полиморфизма реализуется при вызове виртуальных функций для указателей (ссылок) на объекты. При открытом наследовании указатель родительского класса может указывать на объекты

всех подклассов. Если виртуальная функция имеет различные реализации в подклассах, то выбор, какую ее реализацию вызывать, определяется с учетом выяснения подтипа на этапе выполнения. Таким образом, виртуальная функция вызывается в зависимо- сти не от типа указателя, а от реального типа объекта, на который он указывает. Данная ситуация называется механизмом позднего связывания.

Чистый полиморфизм позволяет взаимодействовать с объектом, не зная, к какому конкретному классу он относится. Это происходит за счет общего интерфейса классов в открытой иерархии наследования. Продемонстрируем механизм позднего связывания в построенной нами иерархии классов.

```
int main ( ){
Shape *pS;
Circle C;
FilledCircle FC;
CombiCircle CC;
. . .
 pS = &C; pS -> draw( ); // pS указывает на объект типа Circle,
// рисуется фигура-контур
pS = &FC; pS -> draw ( ); pS указывает на объект типа FilledCircle,
 // рисуется закрашенная фигура
pS = &CC; pS -> draw ( ); // pS указывает на объект типа CombiCircle,
// рисуется комбинированная фигура
 }
```

**Контрольные вопросы**

1. Если класс содержит хотя бы одну чисто виртуальную функцию, то класс называется _____.
2. Вызов функции, обрабатываемый во время компиляции, называется _____ связыванием.
3. Производный класс, полученный закрытым наследованием, не является подтипом базового класса?
4. Что такое виртуальная функция и каковы преимущества ее использования.
5. Каким образом в Java реализуется механизм абстрактных базовых классов и общего интерфейса. В чем разница между абстрактным классом и интерфейсом.
6. Сравните преимущества и недостатки композиции и наследования.
7. Объясните необходимость виртуальных деструкторов
8. Как, по вашему, должны работать виртуальные функции в конструкторе и деструкторе.

**Лабораторная работа №9. Модульное тестирование ПО. Разработка в стиле TDD.**

**Цель работы:** изучить реализацию на языке C++ отношений между классами: агрегации, наследования, зависимости

**Общие сведения**

Исключение — это условие ошибки, возможно вне элемента управления программы, которое не позволяет продолжать выполнение программы по обычному пути выполнения. Некоторые операции, включая создание объектов, ввод-вывод файлов и вызовы функций из других модулей, — это возможные источники исключений, даже если программа выполняется правильно. В надежном коде можно предвидеть и обработать исключения.

Для поиска логических ошибок в одной программе или модуле рекомендуется использовать утверждения вместо исключений (см. разделИспользование утверждений).

Visual C++ поддерживает три типа обработки исключений.

- Обработка исключений C++

  В большинстве программ на языке C++ необходимо использовать обработку исключений C++, поскольку она является типобезопасной и гарантирует вызов деструкторов объектов во время очистки стека.
- Структурированная обработка исключений

  Windows предоставляет собственный механизм исключений — SEH. Его не рекомендуется использовать при программировании C++ или MFC. Используйте SEH только в программах, написанных на языке C без использования MFC.
- Исключения MFC

Начиная с версии 3.0, MFC использует исключения C++, но по-прежнему поддерживает более старые макросы обработки исключений, которые по форме схожи с исключениями C++. Хотя эти макросы не рекомендуется использовать в новом программировании, они по-прежнему поддерживаются для обеспечения обратной совместимости. В программах, в которых уже используются макросы, можно также свободно использовать исключения C++. Во время предварительной обработки макросы вычисляются как ключевые слова обработки исключений, определенные в реализации Visual C++ языка C++, начиная с версии Visual C++ 2.0. Начиная работать с исключениями C++, разработчик может оставить существующие макросы исключений.

С помощью компилятора /EH можно указать тип обработки исключений для использования в проекте; по умолчанию используется обработка исключений C++. Не следует комбинировать механизмы обработки ошибок, например не следует использовать исключения C++ со структурированной обработкой исключений. Использование механизма обработки исключений языка C++ позволяет написать более переносимый

код и обрабатывать исключения любого типа. Дополнительные сведения о недостатках структурированной обработки исключений см. в разделе Структурированная обработка исключений. Дополнительные сведения о комбинировании макросов MFC и исключений C++ см. в разделе Исключения. Использование макросов MFC и исключений C++.

Дополнительные сведения об обработке исключений в приложениях CLR см. в разделе Обработка исключений (расширения компонентов C++).

Дополнительные сведения об обработке исключений в процессорах x64 см. в разделе Обработка исключений (x64).

**Операторы try, throw и catch (C++)**

Для реализации обработки исключений в C++ используйте выражения try, throw и catch.

Сначала используйте блок try для включения одного или нескольких операторов, которые могут создавать исключение.

Выражение throw означает, что исключительное условие, как правило - ошибка, произошло в блоке try.

В качестве операнда выражения throw можно использовать объект любого типа. Обычно этот объект используется для передачи информации об ошибке. В большинстве случаев рекомендуется использовать класс std::exception или один из производных классов, определенных в стандартной библиотеке. Если один из них не подходит, рекомендуется создать собственный производный класс исключений из std::exception.

Для обработки исключений, которые могут быть созданы, необходимо реализовать один или несколько блоков catch сразу после блока try.Каждый блок catch указывает тип исключения, которое он может обрабатывать.

В этом примере показан блок try и его обработчики. Предположим, GetNetworkResource() получает данные через сетевое подключение, а 2 типа исключений являются определенными пользователем классами, производными от std::exception. Обратите внимание, что исключения перехватываются ссылкой const в операторе catch. Рекомендуется создавать исключения по значению и захватывать их ссылкой константы.

Пример

```
MyData md;
try {
  // Code that could throw an exception
  md = GetNetworkResource();
}
catch (const networkIOException& e) {
  // Code that executes when an exception of type
  // networkIOException is thrown in the try block
  // ...
  // Log error message in the exception object
  cerr << e.what();
}
catch (const myDataFormatException& e) {
  // Code that handles another exception type
  // ...
  cerr << e.what();
}
```

```
// The following syntax shows a throw expression
MyData GetNetworkResource()
{
  // ...
  if (IOSuccess == false)
    throw networkIOException("Unable to connect");
  // ...
  if (readError)
    throw myDataFormatException("Format error");
  // ...
}
```

Заметки

Сразу за предложением try находится защищенный раздел кода. Выражение throwвызывает исключение, т.е. создает его. Блок кода после предложения catch является обработчиком исключения. Это обработчик, который перехватывает исключение, вызываемое, если типы в выражениях throw и catch совместимы. Список правил, которыми регулируется сопоставление типов в блоках catch, см. в разделе Проверка блоков Catch (C++). Если оператор catch задает многоточие (...) вместо типа, блок catch обрабатывает все типы исключений. При компиляции с параметром /EHa, в их число могут входить структурированные исключения C и создаваемые системой или приложением асинхронные исключения, например нарушения, связанные с защитой памяти, делением на ноль и числами с плавающей запятой. Поскольку блоки catch обрабатываются в порядке программы для поиска подходящего типа, обработчик с многоточием должен быть последним обработчиком для соответствующего блока try. Используйте catch(...) осторожно, не позволяйте программе продолжать выполнение, если блоку catch не известно, как обработать конкретное перехваченное исключение. Как правило, блок catch(...)используется для ведения журнала ошибок и выполнения специальной очистки перед остановкой выполнения программы.

Выражение throw без операндов повторно создает обрабатываемое в данный момент исключение. Рекомендуется использовать эту форму при повторном создании исключения, поскольку это позволяет сохранить исходные сведения полиморфного типа исключения.Такое выражение следует использовать только в обработчике catch или в функции, вызываемой из обработчика catch. Вновь созданный объект исключения представляет собой исходный объект исключения, а не его копию.

```
try {
  throw CSomeOtherException();
}
catch(...) {
  // Catch all exceptions – dangerous!!!
  // Respond (perhaps only partially) to the exception, then
  // re-throw to pass the exception to some other handler
  // ...
  throw;
}
```

## Проверка блоков Catch (C++)

C++ позволяет создавать исключения любого типа, хотя обычно рекомендуется создавать типы, производные от std::exception. Исключение в C++ может быть перехвачено обработчиком catch, в котором определен тот же тип, что и у созданного исключения, или обработчиком, который способен перехватывать любой тип исключения.

Если созданное исключение имеет тип класса, у которого имеется один или несколько базовых классов, то его могут перехватывать обработчики, которые принимают базовые классы (и ссылки на базовые классы) этого типа исключения. Обратите внимание, что если исключение перехватывается по ссылке, то оно привязывается к самому объекту исключения; в противном случае обрабатывается его копия (как и в случае с аргументами функции).

Созданное исключение может перехватываться следующими типами обработчиков catch:

- Обработчик, который может принимать любой тип данных (синтаксис с многоточием).
- Обработчик, который принимает тот же тип, что и у объекта исключения; поскольку используется копия объекта, то модификаторы**const** и **volatile** игнорируются.
- Обработчик, который принимает ссылку на тот же тип, что и у объекта исключения.
- Обработчик, который принимает ссылку на форму **const** или **volatile** того же типа, что и у объекта исключения.
- Обработчик, который принимает базовый класс того же типа, что и у объекта исключения; поскольку используется копия объекта, то модификаторы **const** и **volatile** игнорируются. Обработчик **catch** для базового класса не должен предшествовать обработчику **catch**для производного класса.
- Обработчик, который принимает ссылку на базовый класс того же типа, что и у объекта исключения.
- Обработчик, который принимает ссылку на форму **const** или **volatile** базового класса того же типа, что и у объекта исключения.
- Обработчик, который принимает указатель, в который можно преобразовать созданный объект указателя при помощи стандартных правил преобразования указателей.

Порядок, в котором располагаются обработчики catch, имеет значение, поскольку обработчики для конкретного блока try проверяются в порядке их следования. Например, ошибкой будет поместить обработчик для базового класса перед обработчиком для производного класса. После того как программа обнаружит подходящий обработчик catch, все последующие обработчики не проверяются. Таким образом, обработчик catch с многоточием должен быть последним обработчиком для своего блока try. Например:

```
// ...
try
{
   // ...
}
catch( ... )
{
   // Handle exception here.
}
// Error: the next two handlers are never examined.
catch( const char * str )
```

```
{
   cout << "Caught exception: " << str << endl;
}
catch( CExcptClass E )
{
   // Handle CExcptClass exception here.
}
```

## Контрольные вопросы

**Лабораторная работа №10. Обобщенное программирование, шаблоны.**

**Цель работы:** изучить реализацию на языке C++ отношений между классами: агрегации, наследования, зависимости

## Общие сведения

STL обеспечивает общецелевые, стандартные классы и функции, которые реализуют наиболее популярные и широко используемые алгоритмы и структуры данных.

STL строится на основе шаблонов классов, и поэтому входящие в нее алгоритмы и структуры применимы почти ко всем типам данных.

Ядро библиотеки образуют три элемента: контейнеры, алгоритмы и итераторы.

Контейнеры (containers) - это объекты, предназначенные для хранения других элементов. Например, вектор, линейный список, множество.

Ассоциативные контейнеры (associative containers) позволяют с помощью ключей получить быстрый доступ к хранящимся в них значениям.

В каждом классе-контейнере определен набор функций для работы с ними. Например, список содержит функции для вставки, удаления и слияния элементов.

Алгоритмы (algorithms) выполняют операции над содержимым контейнера. Существуют алгоритмы для инициализации, сортировки, поиска, замены содержимого контейнеров. Многие алгоритмы предназначены для работы с последовательностью (sequence), которая представляет собой линейный список элементов внутри контейнера.

Итераторы (iterators) - это объекты, которые по отношению к контейнеру играют роль указателей. Они позволяют получить доступ к содержимому контейнера примерно так же, как указатели используются для доступа к элементам массива.

С итераторами можно работать так же, как с указателями. К ним можно применить операции *, инкремента, декремента. Типом итератора объявляется тип iterator, который определен в различных контейнерах. Существует пять типов итераторов:

Итераторы ввода (input iterator) поддерживают операции равенства, разыменования и инкремента.

==, !=, *i, ++i, i++, *i++

Специальным случаем итератора ввода является istream_iterator.

Итераторы вывода (output iterator) поддерживают операции разыменования, допустимые только с левой стороны присваивания, и инкремента.

++i, i++, *i = t, *i++ = t

Специальным случаем итератора вывода является ostream_iterator.

Однонаправленные итераторы (forward iterator) поддерживают все операции итераторов ввода/вывода и, кроме того, позволяют без ограничения применять присваивание.

==, !=, =, *i, ++i, i++, *i

Двунаправленные итераторы (bidirectional iterator) обладают всеми свойствами forward-итераторов, а также имеют дополнительную операцию декремента (--i, i--, *i--), что позволяет им проходить контейнер в обоих направлениях.

Итераторы произвольного доступа (random access iterator) обладают всеми свойствами bidirectional-итераторов, а также поддерживают операции сравнения и адресной арифметики, то есть непосредственный доступ по индексу.

i += n, i + n, i -= n, i - n, i1 - i2, i[n], i1 < i2, i1 <= i2, i1 > i2, i1 >= i2

В STL также поддерживаются обратные итераторы (reverse iterators). Обратными итераторами могут быть либо двунаправленные итераторы, либо итераторы произвольного доступа, но проходящие последовательность в обратном направлении.

Вдобавок к контейнерам, алгоритмам и итераторам в STL поддерживается еще несколько стандартных компонентов. Главными среди них являются распределители памяти, предикаты и функции сравнения.

У каждого контейнера имеется определенный для него распределитель памяти (allocator), который управляет процессом выделения памяти для контейнера.

По умолчанию распределителем памяти является объект класса allocator. Можно определить собственный распределитель.

В некоторых алгоритмах и контейнерах используется функция особого типа, называемая предикатом. Предикат может быть унарным и бинарным. Возвращаемое значение: истина либо ложь. Точные условия получения того или иного значения определяются программистом. Тип унарных предикатовUnPred, бинарных - BinPred . Тип аргументов соответствует типу хранящихся в контейнере объектов.

Определен специальный тип бинарного предиката для сравнения двух элементов. Он называется функцией сравнения (comparison function). Функция возвращает истину, если первый элемент меньше второго. Типом функции является тип Comp .

Особую роль в STL играют объекты-функции.

Объекты-функции - это экземпляры класса, в котором определена операция "круглые скобки" (). В ряде случаев удобно заменить функцию на объект-функцию. Когда объект-функция используется в качестве функции, то для ее вызова используется operator().

```
class less
{
public:
 bool operator()(int x, int y)
 {
  return x < y;
  }
};
```

В STL определены два типа контейнеров: последовательности и ассоциативные.

Ключевая идея для стандартных контейнеров заключается в том, что когда это представляется разумным, они должны быть логически взаимозаменяемыми. Пользователь может выбирать между ними, основываясь на соображениях эффективности и потребности в специализированных операциях. Например, если часто требуется поиск по ключу, можно воспользоваться map (ассоциативным массивом). С другой стороны, если преобладают операции, характерные для списков, можно воспользоваться контейнером list. Если добавление и удаление элементов часто производится в концы контейнера, следует подумать об использовании очереди queue, очереди с двумя концами deque, стека stack. По умолчанию пользователь должен использовать vector ; он реализован, чтобы хорошо работать для самого широкого диапазона задач.

Идея обращения с различными видами контейнеров и, в общем случае, со всеми видами источников информации - унифицированным способом ведет к понятию обобщенного программирования. Для поддержки этой идеи STL содержит множество обобщенных алгоритмов. Такие алгоритмы избавляют программиста от необходимости знать подробности отдельных контейнеров.

В STL определены следующие классы-контейнеры (в угловых скобках указаны заголовочные файлы, где определены эти классы):

bitset - множество битов <bitset.h>

vector - динамический массив <vector.h>

list - линейный список <list.h>

deque - двусторонняя очередь <deque.h>

stack - стек <stack.h>

queue - очередь <queue.h>

priority_queue - очередь с приоритетом <queue.h>

map - ассоциативный список для хранения пар ключ/значение, где с каждым ключом связано одно значение <map.h>

multimap - с каждым ключом связано два или более значений <map.h>

set - множество <set.h>

multiset - множество, в котором каждый элемент не обязательно уникален <set.h>

Типы:

value_type - тип элемента

allocator_type - тип распределителя памяти

size_type - тип индексов, счетчика элементов и т.д.

iterator - ведет себя как value_type *

reverse_iterator просматривает контейнер в обратном порядке

reference - ведет себя как value_type &

key_type - тип ключа (только для ассоциативных контейнеров)

key_compare - тип критерия сравнения (только для ассоциативных контейнеров)

mapped_type - тип отображенного значения

Итераторы:

begin() - указывает на первый элемент

end() - указывает на элемент, следующий за последним

rbegin() - указывает на первый элемент в обратной последовательности

rend() - указывает на элемент, следующий за последним в обратной последовательности

Доступ к элементам:

front() - ссылка на первый элемент

back() - ссылка на последний элемент

operator [](i) - доступ по индексу без проверки

at(i) - доступ по индексу с проверкой

Включение элементов:

insert(p, x) - добавление x перед элементом, на который указывает p

insert(p, n, x) - добавление n копий x перед p

insert(p, first, last) - добавление элементов из [first:last] перед p

push_back(x) - добавление x в конец

push_front(x) - добавление нового первого элемента (только для списков и очередей с двумя концами)

Удаление элементов:

pop_back() - удаление последнего элемента

pop_front() - удаление первого элемента (только для списков и очередей с двумя концами)

erase(p) - удаление элемента в позиции p

erase(first, last) - удаление элементов из [first:last]

clear() - удаление всех элементов

Другие операции:

size() - число элементов

empty() - контейнер пуст

capacity() - память, выделенная под вектор (только для векторов)

reserve(n) - выделяет память для контейнера под n элементов

resize(n) - изменяет размер контейнера (только для векторов, списков и очередей с двумя концами)

swap(x) - обмен местами двух контейнеров

==, !=, < операции сравнения

Операции присваивания:

operator =(x) - контейнеру присваиваются элементы контейнера x

assign(n, x) - присваивание контейнеру n копий элементов x (не для ассоциативных контейнеров)

assign(first, last) - присваивание элементов из диапазона [first:last]

Ассоциативные операции:

operator [](k) - доступ к элементу с ключом k

find(k) - находит элемент с ключом k

lower_bound(k) - находит первый элемент с ключом k

upper_bound(k) - находит первый элемент с ключом, большим k

equal_range(k) - находит lower_bound (нижнюю границу) и upper_bound (верхнюю границу) элементов с ключом k

Вектор vector в STL определен как динамический массив с доступом к его элементам по индексу.

```
template <class T, class Allocator = allocator<T> > class std::vector
{
// ...
};
```

где T - тип предназначенных для хранения данных. Allocator задает распределитель памяти, который по умолчанию является стандартным.

В классе vector определены следующие конструкторы:

```
explicit vector(const Allocator &a = Allocator());
explicit vector(size_type число, const T &значение = T(),
        const Allocator &a = Allocator());

vector(const vector<T, Allocator> &объект);

template<class InIter> vector(InIter начало, InIter конец,

const Allocator &a = Allocator());
```

Первая форма представляет собой конструктор пустого вектора. Во второй форме конструктора вектора число элементов - это число, а каждый элемент равен значению значение. Параметр значение может быть значением по умолчанию. Третья форма конструктора вектор - это конструктор копирования. Четвертая форма - это конструктор вектора, содержащего диапазон элементов, заданный итераторами начало и конец.

```
vector<int> a;
vector<double> x(5);
vector<char> c(5, '*');
vector<int> b(a); // b = a
```

Для любого объекта, который будет храниться в векторе, должен быть определен конструктор по умолчанию. Кроме того, для объекта должны быть определены операторы < и == .

Для класса вектор определены следующие операторы сравнения:

==, <, <=, !=, >, >=.

Кроме этого, для класса vector определяется оператор индекса [].

Новые элементы могут включаться с помощью функций insert(), push_back(), resize(), assign().

Существующие элементы могут удаляться с помощью функций erase(), pop_back(), resize(), clear()

Доступ к отдельным элементам осуществляется с помощью итераторов begin(), end(), rbegin(), rend().

Манипулирование контейнером, сортировка, поиск в нем и тому подобное возможно с помощью глобальных функций файла - заголовка<algorithm.h>.

```
#include <iostream.h>
#include <vector.h>

using namespace std;

int main(void)
{
 vector<int> v;
  for(int i = 0; i < 10; i++)
  {
   v.push_back(i);
  }
  cout << "size = " << v.size() << "\n";
  for (int i = 0; i < 10; i++)
  {
   cout << v[i] << " ";
  }
  cout << endl;
  for (int i = 0; i < 10; i++)
  {
   v[i] = v[i] + v[i];
  }
  for (int i = 0; i < v.size(); i++)
  {
   cout << v[i] << " ";
  }
  cout << endl;
  return 0;
}
 Доступ к вектору через итератор
#include <iostream.h>
#include <vector.h>

using namespace std;

int main(void)
{
  vector<int> v;
  for (int i = 0; i < 10; i++)
  {
   v.push_back(i);
  }
```

```
  cout << "size = " << v.size() << "\n";
  vector<int>::iterator p = v.begin();
  while (p != v.end())
  {
    cout << *p << " ";
    p++;
  }
  return 0;
}
```

Вставка и удаление элементов

```
#include <iostream.h>
#include <vector.h>

using namespace std;

int main(void)
{
  vector<int> v(5, 1);
  // вывод
  for (int i = 0; i < 5; i++)
  {
    cout << v[i] << " ";
  }
  cout << endl;
  vector<int>::iterator p = v.begin();
  p += 2;
  // вставить 10 элементов со значением 9
  v.insert(p, 10, 9);
  //вывод
  p = v.begin();
  while (p != v.end())
  {
    cout << *p << " ";
    p++;
  }
  // удалить вставленные элементы
  p = v.begin();
  p += 2;
  v.erase (p, p + 10);
  // вывод
  p = v.begin();
  while (p != v.end())
  {
    cout << *p << " ";
    p++;
  }
  return 0;
}
```

Вектор содержит объекты пользовательского класса

```
#include <iostream.h>
#include <vector.h>
#include "student.h"
```

```
using namespace std;
int main(void)
{
  vector<STUDENT> v(3);
  v[0] = STUDENT("Иванов", 45.9);
  v[1] = STUDENT("Петров", 30.4);
  v[0] = STUDENT("Сидоров", 55.6);
  // вывод
  for (int i = 0; i < 3; i++)
  {
    cout << v[i] << " ";
  }
  cout << endl;
  return 0;
}
```

Ассоциативный массив содержит пары значений. Зная одно значение, называемое ключом (key), мы можем получить доступ к другому, называемому отображенным значением (mapped value).

Ассоциативный массив можно представить как массив, для которого индекс не обязательно должен иметь целочисленный тип:

V &operator [](const K &) возвращает ссылку на V, соответствующий K .

Ассоциативные контейнеры - это обобщение понятия ассоциативного массива.

Ассоциативный контейнер map - это последовательность пар (ключ, значение), которая обеспечивает быстрое получение значения по ключу. Контейнер map предоставляет двунаправленные итераторы.

Ассоциативный контейнер map требует, чтобы для типов ключа существовала операция < . Он хранит свои элементы отсортированными по ключу так, что перебор происходит по порядку.

Спецификация шаблона для класса map:

```
template<class Key, class T, class Comp = less<Key>,
        class Allocator = allocator<pair> >
class std::map;
```

В классе map определены следующие конструкторы:

```
explicit map(const Comp &c = Comp(), const Allocator &a = Allocator());
map(const map<Key, T, Comp, Allocator> &ob);
template<class InIter> map(InIter first, InIter last, const Comp &c = Comp(),

const Allocator &a = Allocator());
```

Первая форма представляет собой конструктор пустого ассоциативного контейнера, вторая - конструктор копии, третья - конструктор ассоциативного контейнера, содержащего диапазон элементов.

Определена операция присваивания:

```
map &operator =(const map &);
```

Определены следующие операции: ==, <, <=, !=, >, >= .

В map хранятся пары ключ/значение в виде объектов типа pair .

Создавать пары ключ/значение можно не только с помощью конструкторов класса pair, но и с помощью функции make_pair, которая создает объекты типа pair , используя типы данных в качестве параметров.

Типичная операция для ассоциативного контейнера - это ассоциативный поиск при помощи операции индексации ([]).

```
mapped_type &operator [](const key_type &K);
```

Множества set можно рассматривать как ассоциативные массивы, в которых значения не играют роли, так что мы отслеживаем только ключи.

template<classT, class Cmp = less<T>, class Allocator = allocator<T> >

class std::set
{
  //...
};

Множество, как и ассоциативный массив, требует, чтобы для типа T существовала операция "меньше" (< ). Оно хранит свои элементы отсортированными, так что перебор происходит по порядку.

Каждый алгоритм выражается шаблоном функции или набором шаблонов функций. Таким образом, алгоритм может работать с очень разными контейнерами, содержащими значения разнообразных типов. Алгоритмы, которые возвращают итератор, как правило, для сообщения о неудаче используют конец входной последовательности. Алгоритмы не выполняют проверки диапазона на их входе и выходе. Когда алгоритм возвращает итератор, это будет итератор того же типа, что и был на входе. Алгоритмы в STL реализуют большинство распространенных универсальных операций с контейнерами, такие как просмотр, сортировка, поиск, вставка и удаление элементов.

Алгоритмы определены в заголовочном файле <algorithm.h> .

Ниже приведены имена некоторых наиболее часто используемых функций-алгоритмов STL.

Немодифицирующие операции.

for_earch() - выполняет операции для каждого элемента последовательности

find() - находит первое вхождение значения в последовательность

find_if() - находит первое соответствие предикату в последовательности

count() - подсчитывает количество вхождений значения в последовательность

count_if() - подсчитывает количество выполнений предиката в последовательности

search() - находит первое вхождение последовательности как подпоследовательности

search_n() - находит n-е вхождение значения в последовательность

Модифицирующие операции.

copy() - копирует последовательность, начиная с первого элемента

swap() - меняет местами два элемента

replace() - заменяет элементы с указанным значением

replace_if() - заменяет элементы при выполнении предиката

replace_copy() - копирует последовательность, заменяя элементы с указанным значением

replace_copy_if() - копирует последовательность, заменяя элементы при выполнении предиката

fill() - заменяет все элементы данным значением

remove() - удаляет элементы с данным значением

remove_if() - удаляет элементы при выполнении предиката

remove_copy() - копирует последовательность, удаляя элементы с указанным значением

remove_copy_if() - копирует последовательность, удаляя элементы при выполнении предиката

reverse() - меняет порядок следования элементов на обратный

random_shuffle() - перемещает элементы согласно случайному равномерному распределению ("тасует" последовательность)

transform() - выполняет заданную операцию над каждым элементом последовательности

unique() - удаляет равные соседние элементы

unique_copy() - копирует последовательность, удаляя равные соседние элементы

Сортировка.

sort() - сортирует последовательность с хорошей средней эффективностью

partial_sort() - сортирует часть последовательности

stable_sort() - сортирует последовательность, сохраняя порядок следования равных элементов

lower_bound() - находит первое вхождение значения в отсортированной последовательности

upper_bound() - находит первый элемент, больший чем заданное значение

binary_search() - определяет, есть ли данный элемент в отсортированной последовательности

merge() - сливает две отсортированные последовательности

Работа с множествами.

includes() - проверка на вхождение

set_union() - объединение множеств

set_intersection() - пересечение множеств

set_difference() - разность множеств

Минимумы и максимумы.

min() - меньшее из двух

max() - большее из двух

min_element() - наименьшее значение в последовательности

max_element() - наибольшее значение в последовательности

Перестановки.

next_permutation() - следующая перестановка в лексикографическом порядке

prev_permutation() - предыдущая перестановка в лексикографическом порядк