[Лабораторная работа №0. Работа в среде Microsoft Visual Studio. Основы языка программирования C++. 2](#_Toc411406334)

[Обзор IDE Microsoft Visual Studio 2008 2](#_Toc411406335)

[Проекты 2](#_Toc411406336)

[Сборка проекта 2](#_Toc411406337)

[Решение 2](#_Toc411406338)

[Основное окно среды разработки 3](#_Toc411406339)

[Создание нового проекта 4](#_Toc411406340)

[Редактирование свойств проекта 5](#_Toc411406341)

[Установление зависимостей между проектами 5](#_Toc411406342)

[Редактирование исходного кода 5](#_Toc411406343)

[Компиляция и компоновка проекта 6](#_Toc411406344)

[Конфигурации проекта 6](#_Toc411406345)

[Платформа проекта 7](#_Toc411406346)

[Запуск приложения 7](#_Toc411406347)

[Отладка приложения 8](#_Toc411406348)

[Точки останова 8](#_Toc411406349)

[Просмотр и изменение значений переменных в процессе отладки 9](#_Toc411406350)

[Просмотр и изменение содержимого ячеек памяти 10](#_Toc411406351)

[Пошаговое выполнение (трассировка) программы 10](#_Toc411406352)

[Внесение изменений в код в процессе отладки программы 10](#_Toc411406353)

[Прекращение выполнения приложения 10](#_Toc411406354)

[Разработка приложений в Microsoft Visual Studio 11](#_Toc411406355)

[Прекомпилированные заголовки (precomiled headers) 11](#_Toc411406356)

[Практические задания 13](#_Toc411406357)

[Обязательные задания 13](#_Toc411406358)

[Задание 1 – 10 баллов 13](#_Toc411406359)

[Задание 2. – 10 баллов 14](#_Toc411406360)

[Бонус – 5 баллов за возможность указания верхней границы выводимого диапазона 16](#_Toc411406361)

[Задание 3. – 30 баллов 16](#_Toc411406362)

[Дополнительные задания 16](#_Toc411406363)

[Задание 4. – 60 баллов 16](#_Toc411406364)

[Задание 5. – 40 баллов 21](#_Toc411406365)

# Лабораторная работа №0. Работа в среде Microsoft Visual Studio. Основы языка программирования C++.

В данной лабораторной работе Вы познакомитесь с IDE[[1]](#footnote-1) Microsoft Visual Studio и научитесь использовать ее для написания и отладки программ на языке C++.

## Обзор IDE Microsoft Visual Studio 2008

Среда разработки Microsoft Visual Studio (в дальнейшем Visual Studio) позволяет разрабатывать и отлаживать программы, написанные на языках программирования C++, C#, Visual Basic .NET, F#, J# и других.

### Проекты

Разрабатываемая программа представлена в Visual Studio в виде **проекта**, содержащего файлы исходных кодов на используемом языке программирования, файлы ресурсов, например, изображений и различных вспомогательных файлов. Перечень файлов, входящих в проект, а также свойства проекта хранятся в **файле проекта**. Расширение файла проекта зависит от используемого языка программирования. Для проектов, написанных на языках программирования C и C++, это файл с расширением **.vcproj**.

### Сборка проекта

Сборка проекта – процесс преобразования исходных файлов проекта в некоторое конечное представление, например, исполняемый файл (.EXE), динамическую (.DLL) или статическую (.LIB) библиотеку.

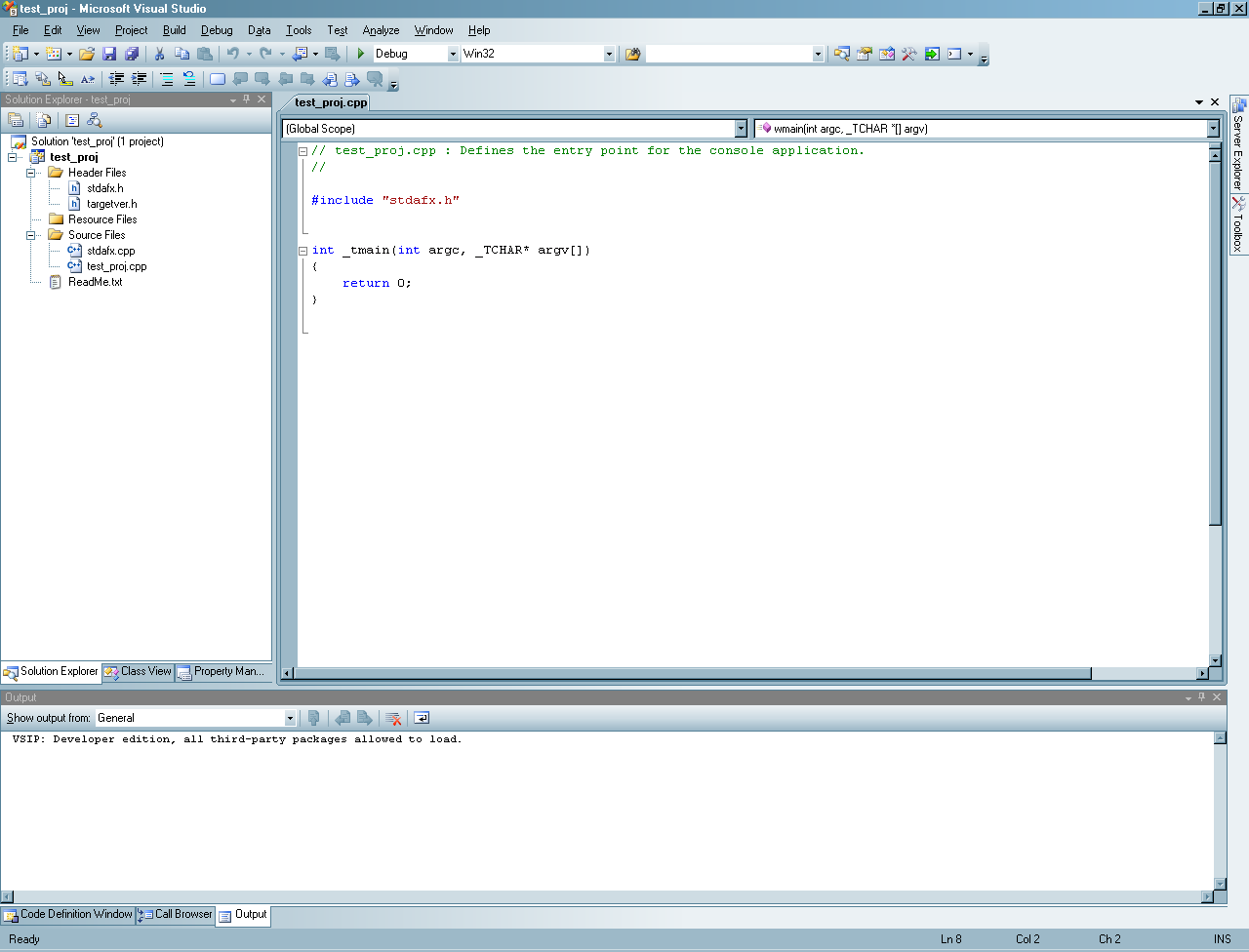
В процессе сборки, как правило, выполняется компиляция файлов исходных кодов для получения объектных файлов[[2]](#footnote-2) и их последующая компоновка[[3]](#footnote-3) в приложение или библиотечный модуль.

### Решение

Среда Visual Studio позволяет использовать несколько проектов совместно в составе **решения** (англ. Solution), что облегчает разработку целых программных комплексов, состоящих из нескольких программных модулей, использующихся совместно.

Между проектами, входящими в состав одного решения, можно установить зависимости, влияющие на последовательность сборки проектов решения, а также на использование одних проектов в составе других. Таким образом, становится возможным вынесение совместно используемого кода нескольких приложений в отдельный проект (например, статическую или динамическую библиотеку) и установка зависимостей проектов приложений от проектов используемой ими библиотеки. При сборке приложения сначала будут собраны проекты используемых данным приложением библиотек, а затем будет собрано само приложение.

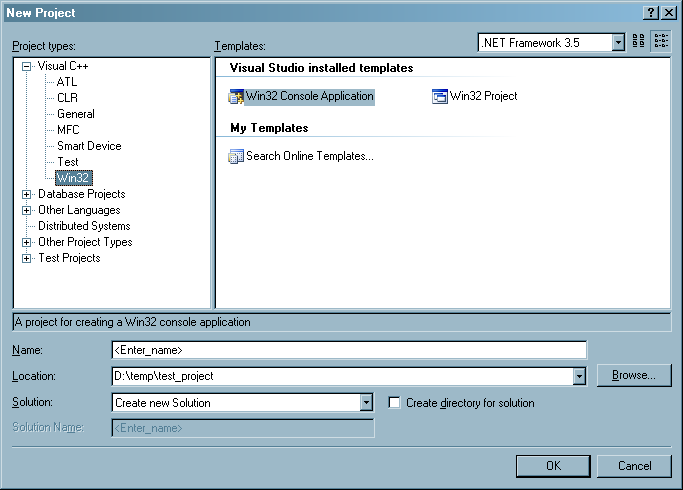
### Основное окно среды разработки



1. Окно редактирования исходного кода.
2. Solution Explorer (и другие панели).
3. Панели инструментов, меню.
4. Панели вывода информации.

### Создание нового проекта

Меню File→New Project открывает мастер создания нового проекта. Для каждого используемого языка программирования доступен свой набор проектов.

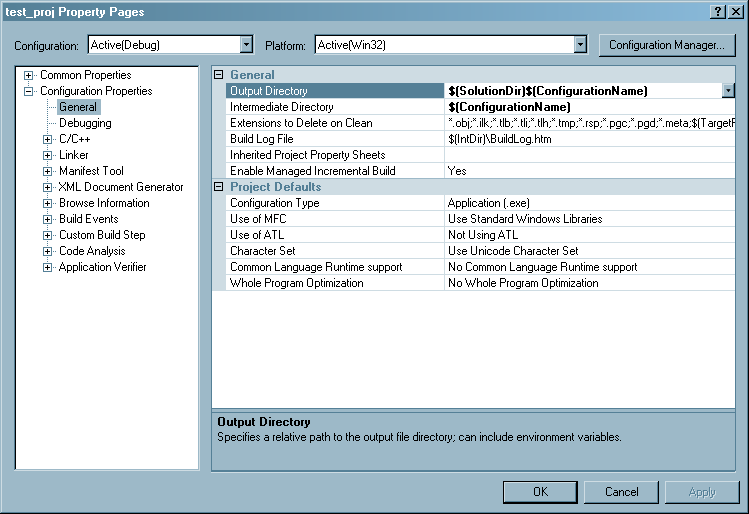
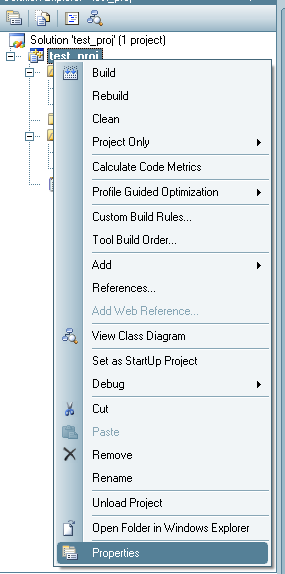


Для проектов на языке C++ необходимо отметить следующие типы проектов[[4]](#footnote-4):

* Windows application – оконное приложение для ОС Windows. В результате сборки данного типа проектов будет создано приложение с оконным интерфейсом.
* Console application – консольное приложение для ОС Windows. Создает приложение, работающее в текстовом режиме.
* Static library – статически компонуемая библиотека. Результатом сборки данного типа проекта будет .LIB файл, содержащий функции, которые могут быть скомпонованы в состав других модулей.
* Dynamic library – динамически компонуемая библиотека (DLL). В результате сборки данного проекта будет создан .DLL файл, содержащий функции, которые могут быть совместно использованы несколькими приложениями одновременно.
* MFC Application – приложение, использующее библиотеку базовых классов (MFC - Microsoft Foundation Classes), обеспечивающую построение оконных приложений для ОС Windows.
* ATL Project – COM[[5]](#footnote-5)-сервер, организованный в виде динамической библиотеки, исполняемого модуля или сервиса, использующий библиотеку ATL.

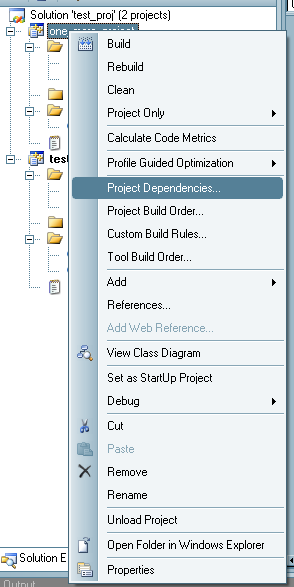
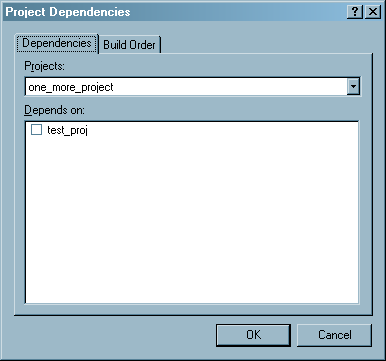
### Редактирование свойств проекта

Окно редактирования свойств проекта вызывается при помощи выбора пункта **Properties** контекстного меню выбранного проекта в окне **Solution Explorer**:



### Установление зависимостей между проектами

При помощи меню **Project Dependencies** возможно установить зависимости одного проекта от других. Данные зависимости будут учитываться при определении порядка сборки проектов в решении.

### Редактирование исходного кода

Встроенный редактор исходного кода Visual Studio предоставляет возможности по автоматическому завершению вводимых операторов, названий переменных и функций. Существуют также коммерческие плагины, расширяющие возможности стандартного редактора исходного кода. Один из лучших – **[Visual Assist](http://visualassist.com/)** компании Whole Tomato.

### Компиляция и компоновка проекта

Собрать проекты в составе одного решения можно при помощи клавиши[[6]](#footnote-6) **F7**. Откомпилировать открытый файл исходного кода можно при помощи комбинации клавиш **Ctrl+F7**.

### Конфигурации проекта

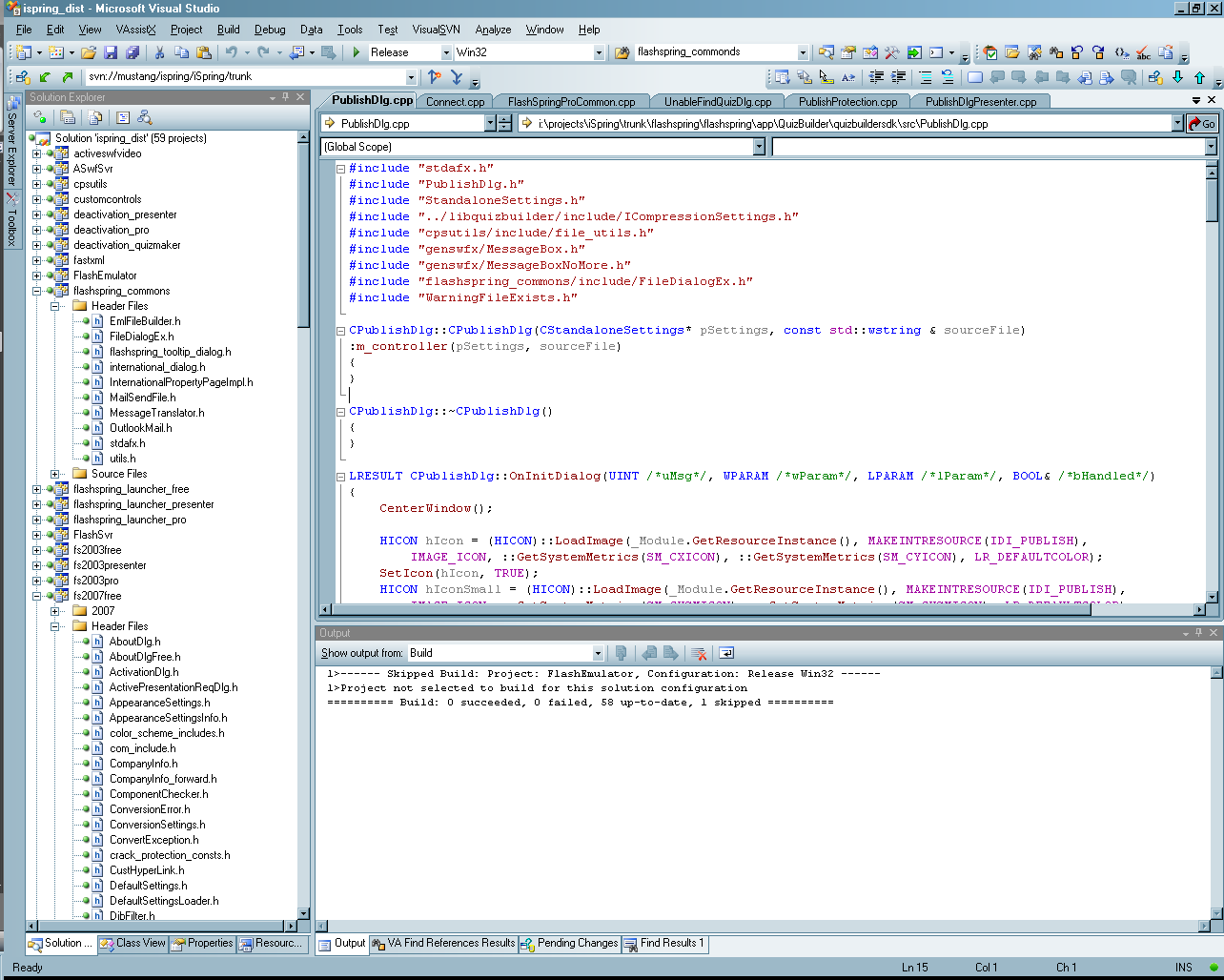
Конфигурация проекта – именованный набор настроек проекта. Изначально проект содержит 2 конфигурации:

* **Debug** – отладочная конфигурация. В процессе компиляции отключается оптимизация кода, в собираемый модуль помещается дополнительная информация для отладки кода, используются отладочные версии **библиотек времени выполнения**[[7]](#footnote-7). Собранный в отладочной конфигурации модуль, как правило, обладает большим размером и меньшим быстродействием, однако наиболее полно обеспечивает возможности отладки на уровне исходного кода.
* **Release** – финальная конфигурация. При компиляции в Release-конфигурации включена оптимизация кода по быстродействию или размеру, и, как правило, исключается отладочная информация из собираемого модуля. Собранный в финальной конфигурации модуль зачастую обладает меньшим размером и большим быстродействием, однако возможность отладки на уровне исходного кода значительно затруднена.

Отладочная конфигурация должна использоваться только в процессе отладки приложения. Для предоставления конечному пользователю, напротив, следует использовать Release-конфигурацию. Помимо низкого быстродействия и часто большего размера, приложение, собранное в отладочной конфигурации, может требовать для своей работы наличия отладочной версии библиотеки времени выполнения, которые не поставляются вместе с операционной системой.

Помимо стандартных конфигураций программист может задать дополнительные конфигурации для проектов, например, оптимизированные для уменьшения размера или уменьшения зависимостей от внешних runtime-библиотек.

Пользователь может выбрать одну из доступных конфигураций при помощи выпадающего списка на панели инструментов. Во время следующей сборки проекты будут скомпилированы и скомпонованы в указанной конфигурация.

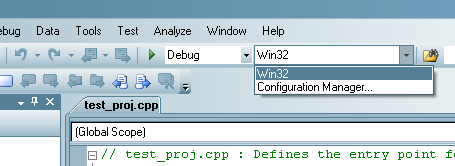


Для пакетной сборки проектов сразу в нескольких конфигурациях можно воспользоваться командой меню Build→Batch Build…

### Платформа проекта

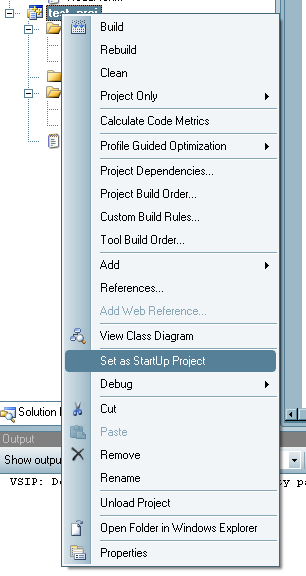
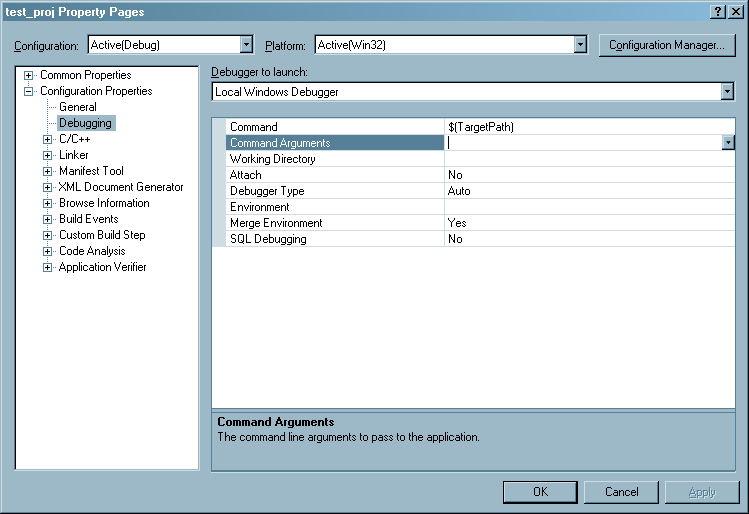
Операционные системы семейства Windows выпускаются в редакциях для 32-битных (x86) и 64-битных (x64) процессоров. Под управлением 64-битных версий операционных систем Windows могут выполняться 64-битные приложения, которым доступен больший объем оперативной памяти, способные выполнять операции над 64-битными целыми числами быстрее, чем 32-битные приложения.

Среда Visual Studio позволяет создавать как 32-битные, так и 64-битные приложения (при установке соответствующих версий компилятора). При сборке приложения можно выбрать платформу (32-битную или 64-битную, если были установлены соответствующие компоненты), для которой будет осуществляться сборка указанной конфигурации проекта.



### Запуск приложения

Комбинация клавиш **Ctrl+F5** позволяет запустить собранный проект (в выбранной конфигурации для указанной платформы) прямо из среды разработки. В случае, когда решение содержит несколько проектов, один из проектов может быть помечен как запускаемый при помощи контекстного меню окна Solutions Explorer.

В окне настройки проекта пользователь может указать аргументы командной строки, передаваемые приложению при его запуске, директорию по умолчанию, а также дополнительные динамически подключаемые библиотеки, загружаемые в адресное пространство приложения при его запуске.

### Отладка приложения

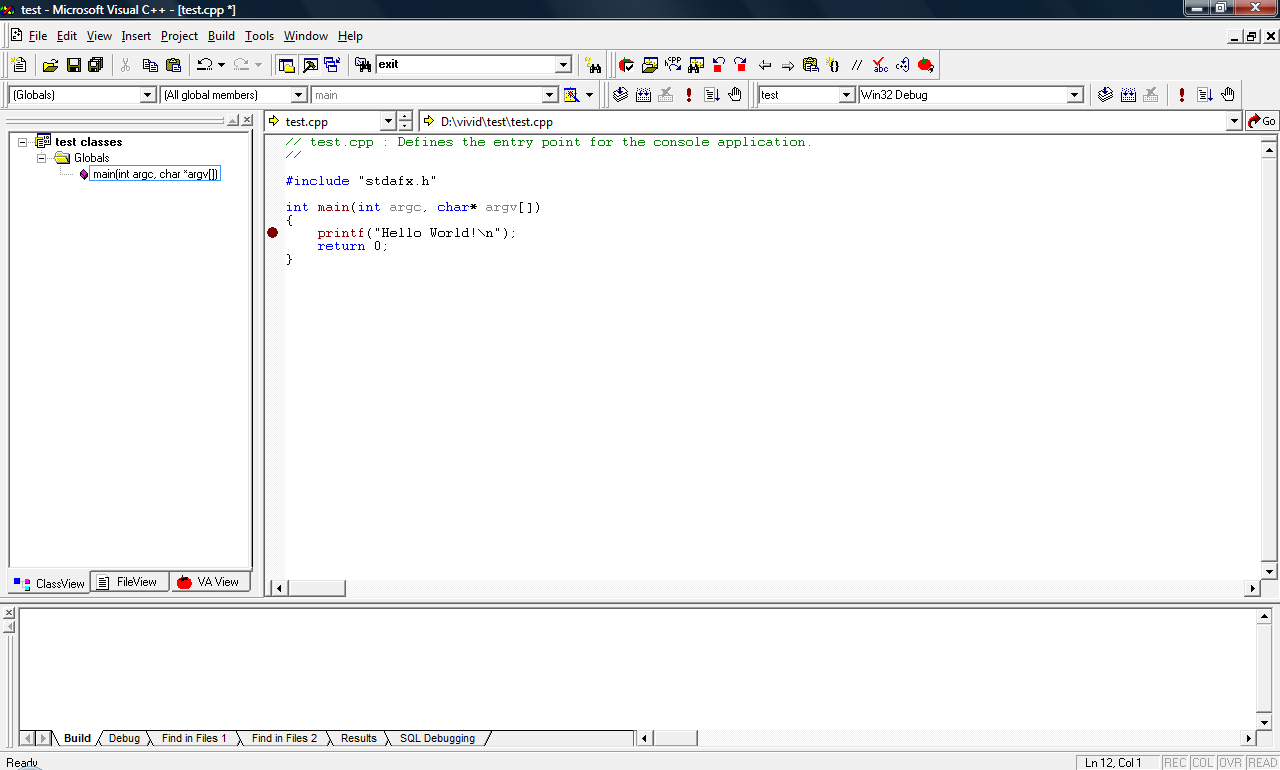
Отладка приложения (англ. - **debugging**[[8]](#footnote-8)) – процесс поиска и исправления ошибок в разрабатываемой программе.

**Клавиша** **F5** осуществляет запуск активного проекта под управлением встроенного отладчика среды Visual Studio. В режиме отладки программисту доступны следующие возможности отладки приложений:

* Остановка выполнения программы в указанных ***точках останова***[[9]](#footnote-9)
* Трассировка программы
* Просмотр и изменение значений переменных и содержимого ячеек памяти

#### Точки останова

**Клавиша F9** позволяет установить или снять ранее установленную точку останова со строки, на которую установлен курсор в редакторе исходного кода. Строка программы, на которой установлена точка останова, помечается кружочком:



Программист может установить произвольное количество точек останова в программе.

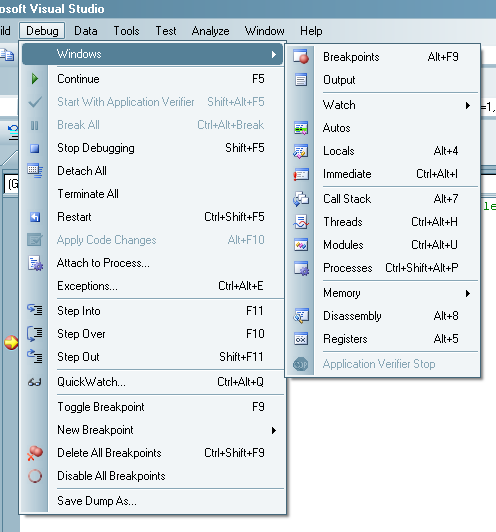
Внимание: в момент запуска приложения в режиме отладки положение точек останова может измениться, например, если точка останова была установлена на строках, не содержащих операторов, или внутри комментариев.

Выполнение программы приостанавливается, как только выполнение доходит строки, помеченной точкой останова.

#### Просмотр и изменение значений переменных в процессе отладки

В процессе отладки приостановленного приложения программисту доступны окна просмотра локальных переменных (Local variables) и окна просмотра выражений (Watch).

Окна отладчика показываются при помощи команд меню Debug->Windows.



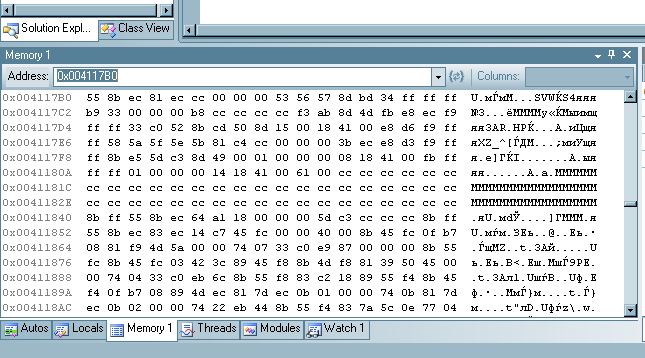
Для просмотра переменных доступны следующие окна:

* Autos – отображение переменных, использующихся текущим и предыдущим операторами;
* Locals – отображение локальных переменных текущей функции или метода класса.

В окне **Watch** выражений программист может просматривать значения переменных и выражений, а также изменять значения переменных (но не значения выражений).

#### Просмотр и изменение содержимого ячеек памяти

Помимо изменения значения отдельных переменных программисту также доступно содержимое адресного пространства отлаживаемого процесса. При помощи окна **Memory** программист может просматривать и изменять содержимое ячеек памяти. Значения ячеек памяти могут отображаться в виде 8-, 16-, и 32-битных значений (в шестнадцатеричной системе счисления).



Окно Memory вызывается при помощи команды меню **Debug→Memory**.

#### Пошаговое выполнение (трассировка) программы

Трассировка позволяет программисту проследить выполнение программы «по шагам», на каждом шаге отслеживая значения переменных и/или содержимое ячеек памяти, а также порядок выполнения операторов.

Следующие комбинации клавиш используются для пошаговой трассировки программы:

* **F10** – выполнить текущий оператор. Если текущим оператором является оператор вызова функции или метода класса, то он выполняется как единое целое.
* **F11** – выполнить текущий оператор. Если текущим оператором является оператор вызова функции, то происходит вход внутрь тела функции.
* **Shift + F11** – продолжить выполнение программы до выхода из текущей функции или метода класса.
* **Ctrl + F10** – продолжить выполнение программы до достижения позиции курсора.
* **Alt + Num\*** – установить курсор в позицию выполнения следующего оператора.
* **F5** – возобновить выполнение программы.

#### Внесение изменений в код в процессе отладки программы

Среда Visual Studio предоставляет механизм Edit and Continue, позволяющий внести в программу необходимые изменения в процессе отладки и в ряде случаев продолжить выполнение измененной программы без ее перезапуска. После внесения изменений достаточно нажать комбинацию клавиш **Alt + F10**, либо выполнить одну из команд трассировки программы.

#### Прекращение выполнения приложения

Остановить работу приложения и выйти из режима отладки можно осуществить при помощи комбинации клавиш **Shift + F5**.

## Разработка приложений в Microsoft Visual Studio

### Прекомпилированные заголовки (precomiled headers)

Для связи исходных файлов друг с другом во время компиляции используется директива препроцессора **#include**, выполняющая подключение заголовочного файла с заданным именем вместо директивы #include. Подключаемые заголовочные файлы, в свою очередь, могут включать другие заголовочные файлы и так далее

Объем заголовочных файлов операционной системы Windows, использующихся обычным приложением, может занимать несколько сотен килобайт или даже несколько мегабайт. Обработка заголовочных файлов осуществляется для каждой **единицы компиляции**[[10]](#footnote-10) проекта, и в указанном случае может занимать длительное время.

Допустим, объем внешних заголовочных файлов, использующихся в проекте среднего размера, составляет 10 тысяч строк, а количество единиц компиляции, содержащихся в проекте, равно 200 общим объемом 50 тысяч строк.

Для полной перекомпиляции проекта потребуется обработать 10000\*200 + 50000 = 2 050 000 строк исходного кода, причем 2 миллиона строк из них составят заголовочные файлы. Пусть обработка 1 миллиона строк кода происходит за 1 минуту.

этом случае полная сборка проекта потребует около 2 минут. Для ускорения процесса обработки заголовочных файлов в ряде компиляторов (и в Visual Studio) используются файлы предварительно скомпилированных заголовков.

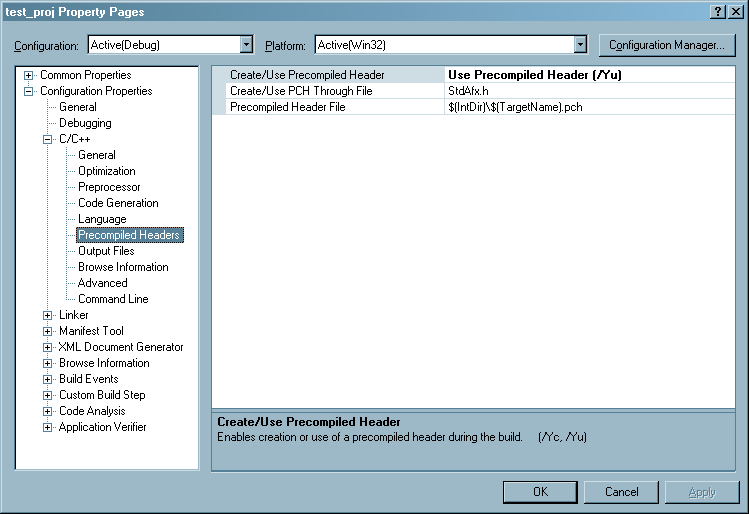
Идея проста. В одном из заголовочных файлов проекта, по умолчанию в файле **stdafx.h**, при помощи директивы #include подключаются **часто используемые и редко изменяемые** заголовочные файлы. На основе данного заголовочного файла компилятор создает специальный предварительно скомпилированный заголовочный файл (в Visual Studio это файл с расширением .pch). В предварительно скомпилированном заголовочном файле располагается содержимое заголовочного файла в предварительно обработанном виде. В остальных единицах компиляции самой первой инструкцией является директива **#include**, выполняющая подключение заголовочного файла **stdafx.h**.

Следует помнить об обязательном подключении заголовочного файла stdafx.h в начале каждой единицы компиляции (не следует включать их заголовочных файлах), если в проекте используются прекомпилированные заголовки. В противном случае вы получите сообщение об ошибке:

fatal error C1010: unexpected end of file while looking for precompiled header directive

Кроме того, любой код, расположенный до подключения прекомпилированного заголовка, будет проигнорирован компилятором. Поэтому директива #include его подключения должна быть первой инструкцией в единице компиляции.

Использование заголовочных файлов в проекте задается в окне свойств проекта (раздел Configuration Properties→С/C++→Precompiled Headers).



Построение предварительно компилируемого заголовочного файла происходит при каждом его изменении или изменении заголовочных файлов, которые из него подключаются, и занимает длительное время. При этом также происходит перекомпилирование остальных файлов проекта. Поэтому в файле предварительно компилируемых заголовков следует подключать только редко изменяющиеся заголовочные файлы, иначе выгоды от его использования не будет никакой.

Примерами редко изменяющихся файлов могут быть заголовочные файлы операционной системы, стандартных библиотек языков C и C++ и прочих сторонних библиотек. Примерами часто изменяющихся файлов могут быть файлы, входящие в состав проектов решения.

## Практические задания

На оценку «**удовлетворительно**» необходимо выполнить все обязательные задания и набрать не **менее 40 баллов.**

На оценку «**хорошо**» необходимо выполнить все обязательные задания и часть дополнительных и набрать **не менее 80 баллов**.

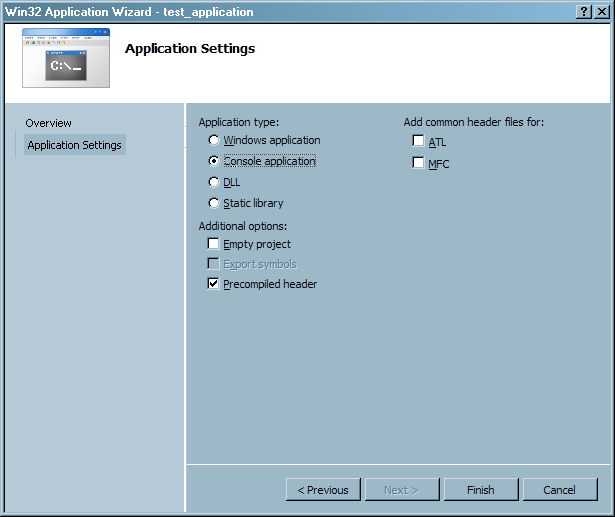
На оценку «**отлично**» необходимо выполнить все обязательные и все дополнительные, набрав **не менее 140 баллов.**

Внимание, дополнительные задания принимаются только после успешной сдачи и защиты обязательных заданий.

### Обязательные задания

#### Задание 1 – 10 баллов

Создайте в среде Visual Studio новый проект на языке C++ **Win32 Console Application**. В окне мастера создания проекта выберите тип приложения «Console Application».



Ознакомьтесь с исходным кодом программы, выводящей через запятую числа от 1 до 10:

#include "stdafx.h"

int main(int argc, char\* argv[])

{

int i;

for (i = 1; i <= 10; ++i)

{

printf("%d", i);

if (i != 10)

{

printf(", ");

}

}

printf("\n");

return 0;

}

Данная программа иллюстрирует использование оператора **for** для организации циклического выполнения с известным начальным (i = 0) и конечным (i <= 10) условием, а также для изменения переменной цикла (++i – операция приращения переменной i).

В теле цикла, заключенном между фигурными скобками после оператора **for**, осуществляется вывод значения целочисленной переменной цикла при помощи функции форматированного вывода **printf**. Модификатора форматирования ”%d” используется для осуществления вывода значения целочисленного выражения (в нашем случае – значения целочисленной переменной i) в десятичной системе счисления.

Для вывода шестнадцатеричного значения целочисленного выражения используется модификатор ”%x”.

Оператор ветвления **if** управляет выводом запятой – запятая должна выводиться после каждого числа, кроме самого последнего.

**Задание:** Выполните трассировку данной программы, обращая внимание на содержимое переменных на каждом шаге выполнения программы. Чему равно значение переменной **i** после выхода из цикла **for**?

Доработайте данную программу, чтобы она выводила через запятую все **четные** числа диапазона от 1 до 100, отсортированные **в порядке убывания**, т.е. 100, 98, 96, …, 6, 4, 2

#### Задание 2. – 10 баллов

Разработайте программу, выводящую в порядке возрастания все целые числа диапазона от 1 до 1000, делящиеся без остатка на сумму своих цифр.

Для решения задачи можно воспользоваться следующим алгоритмом:

* Организуем цикл от 1 до 1000, в котором вычисляем сумму цифр переменной цикла
* Выводим только те значения переменной цикла, остаток деления которой на сумму ее цифр равен нулю

Рассмотрим решение задачи по вычислению суммы цифр некоторого числа.

Рассмотрим некоторое целое число, записываемое в десятичной системе счисления в виде AN…A2A1A0, где Ai – некоторая цифра от 0 до 9. Необходимо найти значения всех Ai, зная значение числа.

Т.к. десятичная система является позиционной системой счисления[[11]](#footnote-11), то любое число ***x*** в ней может быть представлено в виде конечной линейной комбинации степеней числа 10:



Например, число 563 может быть представлено в виде:



Необходимо найти такие коэффициенты ak, являющиеся решением указанной линейной комбинации.

Функция, вычисляющая сумму цифр числа, представлена ниже:

int SumDigits(int i)

{

int sum = 0;

*// цикл продолжается до тех пор, пока число не станет равным 0*

while (i != 0)

{

sum += i % 10;

i /= 10;

}

return sum;

}

Оператор % вычисляет остаток от деления левого операнда на правый операнд.

Выполните пошаговую трассировку данной функции, чтобы понять принцип ее работы

Напишем небольшую программу для проверки правильности работы функции.

#include "stdafx.h"

#include <assert.h>

// calculates sum of the digits of i

int SumDigits(int i)

{

…

}

int main(int argc, char\* argv[])

{

for (int i = 0; i <= 9; ++i)

{

for (int j = 0; j <= 9; ++j)

{

for (int k = 0; k <= 9; ++k)

{

int x = i \* 100 + j \* 10 + k;

int sum = SumDigits(x);

assert((i + j + k) == sum);

}

}

}

return 0;

}

В данной программе мы познакомились с макросом **[assert](http://ru.wikipedia.org/wiki/Assert)**, определенным в файле **assert.h**, который в **отладочной конфигурации** может использоваться для проверки сделанных программой вычислений. В том случае, если выражение в макросе **assert** ложно (его значении равно нулю), то программа аварийно завершает свою работу, выводя в стандартный поток вывода ошибок (stderr) следующую диагностическую информацию:

* Текст выражения, значение которого равно 0
* Имя файла с исходным кодом (предопределенный макрос \_\_FILE\_\_)
* Строка файла с исходным кодом (предопределенный макрос \_\_LINE\_\_)

В финальной конфигурации данный макрос никак не влияет на работу программы.

**Внимание**, макрос assert следует использовать только в тех случаях, когда вы на 100% уверены в том, что значение некоторого выражения истинно, и что единственная причина, по которой оно может быть ложным – логическая ошибка в Вашей программе. В частности, не следует использовать макрос assert для проверки успешности файловых операций, т.к. они могут завершаться неудачей по причинам не связанным с работой программы.

Итак, используя функцию SumDigits, разработайте программу, выводящую в порядке возрастания целые числа диапазона от 1 до 1000, делящиеся без остатка на сумму своих цифр.

##### Бонус – 5 баллов за возможность указания верхней границы выводимого диапазона

Дополнительные баллы можно получить за реализацию следующего дополнительного функционала:

* Возможность указания верхней границы выводимого диапазона при помощи аргумента командной строки. Используйте рассмотренную нами функцию StringToInt для перевода аргумента из строки в число. Внимание, программа должна осуществлять проверку того, что значение верхней границы диапазона больше или равно 1.

#### Задание 3. – 30 баллов

Разработайте программу, выводящую [числа Фибоначчи](http://ru.wikipedia.org/wiki/%D0%A7%D0%B8%D1%81%D0%BB%D0%B0_%D0%A4%D0%B8%D0%B1%D0%BE%D0%BD%D0%B0%D1%87%D1%87%D0%B8)[[12]](#footnote-12) диапазона от 0 до N. Целочисленный параметр N (верхняя граница) должен быть указан пользователем в виде аргумента командной строки. **Числа последовательности должны выводиться через запятую по 5 чисел в каждой строке**.

**Внимание**, программа должна проверять значения входных параметров. В частности, параметр N, должен быть целым числом большим 0. В виду особенностей целочисленной арифметики языков Си и Си++, должна корректно обрабатываться ситуация с переполнением результатов целочисленных выражений. Вы можете столкнуться с данным эффектом, например, при N = 2 миллиардам.

Внимание, время работы программы должно расти линейно в зависимости от

### Дополнительные задания

#### Задание 4. – 60 баллов

Рассмотрим принципы работы программы, вычисляющей сумму чисел, передаваемых ей в виде аргументов командной строки.

#include "stdafx.h"

#include "stdlib.h"

int main(int argc, char\* argv[])

{

if (argc <= 1)

{

printf("Program calculates sum of its command line arguments.\n");

return 0;

}

int sum = 0;

for (int i = 1; i < argc; ++i)

{

char \* pLastChar = NULL;

const char \* pArg = argv[i];

int param = strtol(pArg, &pLastChar, 10);

if ((\*pArg == '\0') || (\*pLastChar != '\0'))

{

printf("Argument #%d is not a number\n.", i);

return 1;

}

printf("%d", param);

sum += param;

if (i != argc-1)

{

printf(" + ");

}

}

printf(" = %d", sum);

return 0;

}

Рассмотрим работу данной программы более подробно.

В функцию **main** передаются параметры **argc** и **argv**, несущие информацию об аргументах командной строки, с которыми было запущено приложение. Параметр **argc** хранит количество переданных аргументов, а параметр **argv** является массивом строк (точнее, массивом указателей на первые символы строк), хранящих аргументы командной строки.

Внимание, **индексация элементов массива в языке C/C++ начинается с 0**, т.е. индекс самого первого элемента массива равен 0, а самого последнего равен **N-1**, где N – количество элементов массива.

В массиве **argv** элемент с индексом 0 хранит путь к приложению (в случае, если приложение запущено из пакетного .bat файла, то только имя). Поэтому переменная i изменяется в цикле от **1** до **argc-1**:

for (int i = 1; i < argc; ++i)

{

…

}

Функция **strtol** (sting to long) выполняет преобразование массива символов, указатель на первый символ которого передается в первом аргументе функции, в длинное целое число, записанное в системе счисления, передаваемой через 3 аргумент функции. 2 аргумент функции – адрес переменной типа **char\*,** в которую будет записан адрес первого нецифрового символа (а точнее, символа, отсутствующего в указанной системе счисления), на котором остановилась обработка переданной строки.

Для преобразования строки в число с плавающей запятой используется функция strtod:

double strtod(const char \* str, char \*\* pLastChar);

В случае, когда преобразование строки в число прошло без ошибок, в переменную, адрес которой передан через второй аргумент функции, будет записан адрес завершающего символа переданной строки с нулевым кодом. Использование данного факта позволяет нам проверить правильность переданного аргумента.

if ((\*pArg == '\0') || (\*pLastChar != '\0'))

В данном операторе ветвления проверяются 2 условия:

1. Является ли самый первый символ переданной строки нулевым символом, иными словами, является ли переданная строка пустой.
2. Является ли символ, на котором была прекращено преобразование строки в число, ненулевым. Иными словами, содержатся ли в строке символы кроме цифровых.

Первое условие является необходимым, т.к. в случае пустой строки результат второго условия не позволит нам определить факт ошибки. Итак, мы считаем строку преобразованной в число без ошибок, если она не пустая и была обработана до конца.

Для передачи пустой строки, либо строки с пробелами программе в виде аргумента командной строки можно, заключив строку в двойные кавычки:

myprog.exe ”” ”argument with spaces”

Для передачи строки, содержащей символ двойных кавычек программе в виде аргумента командной строки можно, экранируя кавычки при помощи обратной косой черты:

myprog.exe \”

Функция main возвращает целочисленный код ошибки, причем принято в случае успешного завершения работы программы возвращать нулевое значение, а в случае ошибки – ненулевое. В случае обнаружения ошибки (нечисловой аргумент) осуществляем принудительный выход из программы с ненулевым кодом возврата.

Несмотря на свою работоспособность, программа получилась несколько сложной для понимания. Главным образом, из-за кода преобразования строки в число.

Улучшим эту ситуацию, вынеся код преобразования строки в число в отдельную функцию:

int StringToInt(const char \* str, bool & err)

{

char \* pLastChar = NULL;

int param = strtol(str, &pLastChar, 10);

err = ((\*str == '\0') || (\*pLastChar != '\0'));

return param;

}

Функция **StringToInt** принимает параметр **err** по ссылке, в который записывает значение **true** в случае обнаружения ошибки при переводе строки в число. В дальшейшем, мы познакомимся с механизмом исключений, продвинутым способом сигнализирования об ошибке.

Заметьте также, что функция **StringToInt** лишь сигнализирует об ошибке, но не выводит никаких сообщений в output – это обязанность кода высшего уровня (функция main). Делегирование данной обязанности функции **main** позволяет программе сообщить пользователю об ошибке более конкретно: «***i****-ый аргумент командной строки не является числом*», вместо абстрактного: «*ошибка преобразования строки fsv9vk в число*». В первом случае пользователь получает исчерпывающую информацию об ошибке и может сразу принять меры по её устранению, а во втором случае пользователю сообщается о том, что где-то в программе произошла ошибка преобразования строки в число.

Сообщение пользователю об ошибке (тем более, вызванной действиями самого пользователя) должно нести **высокоуровневой характер** и содержать ключевую информацию о причине ее возникновения и о мерах по ее устранению.

Никогда не следует показывать пользователю неинформативные сообщения, вроде: «Incorrect argument», «Error», т.к. после такого «плевка в лицо» пользователь, скорее всего, удалит Вашу программу и будет использовать вместо нее программу Вашего конкурента.

После внесенных исправлений функция **main** будет иметь следующий вид:

int main(int argc, char\* argv[])

{

if (argc <= 1)

{

printf("Program calculates sum of its command line arguments.\n");

return 0;

}

int sum = 0;

for (int i = 1; i < argc; ++i)

{

bool err;

int param = StringToInt(argv[i], err);

if (err)

{

printf("Argument #%d is not a number\n.", i);

return 1;

}

printf("%d", param);

sum += param;

if (i != argc-1)

{

printf(" + ");

}

}

printf(" = %d", sum);

return 0;

}

Написанная нами программа вполне может содержать ошибки. Хорошей практикой является критическое отношение к вновь написанной программе: **программа считается содержащей ошибочной, пока достоверно не будет доказано обратное**.

Разумеется, впадать в крайности и математически доказывать корректность написанной программы мы не будем. Вместо этого мы убедимся в том, что результаты работы программы (выводимая программой информация) соответствуют ожидаемым на заданном наборе входных данных (различные наборы аргументов командной строки).

Проверять работу программы следует на всем протяжении цикла ее разработки, а не после того, как программа целиком написана. **Чем раньше будет обнаружена ошибка, тем легче ее будет исправить, не сломав что-то еще.**

Проверка программы вручную обладает рядом недостатков:

* Занимает длительное время
* Сложнее воспроизводится
* Быстро надоедает

Попробуем автоматизировать процесс тестирования при помощи интерпретатора языка сценариев командной строки cmd.exe, встроенного в ОС Windows.

Создадим .bat файл при помощи любого текстового редактора (подойдет даже «блокнот»), в котором будем проверять:

* код возврата нашего приложения (мы ведь не зря их используем в нашем приложении), запущенного с различными параметрами
* соответствие выводимых нашей программой данных ожидаемому результату. Очевидно, что проверки кода возврата приложения не достаточно для проверки ее работы. Необходимо сравнить выводимый программой текст с тем, который мы ожидаем от нее получить (ошибка в программе – несоответствие результата работы программы (или ее части) ожидаемым результатам, поэтому для начала сам программист должен знать, каких результатов ждать от разрабатываемой программы). Выводимые программой в стандартный поток вывода можно перенаправить в файл на диске, а затем при помощи утилиты **fc.exe**[[13]](#footnote-13) сравнить содержимое данного файла с заранее подготовленным файлом, содержащим ожидаемый результат

Пакетный файл, решающий данную задачу, представлен ниже:

REM запуск программы без параметров

test.exe

IF ERRORLEVEL 1 GOTO err

REM запуск программы с одним нечисловым параметром

test.exe notANumber

IF NOT ERRORLEVEL 1 GOTO err

REM запуск программы с пустой строкой в качестве одного из параметров

test.exe ""

IF NOT ERRORLEVEL 1 GOTO err

REM запуск программы с одним числовым параметром (и проверяем результат)

test.exe -123 > output.txt

IF ERRORLEVEL 1 GOTO err

FC /B output.txt -123.txt

IF ERRORLEVEL 1 GOTO err

REM запуск программы с двумя числовыми параметрами

test.exe 15 2 > output.txt

IF ERRORLEVEL 1 GOTO err

FC /B output.txt 15+2=17.txt

IF ERRORLEVEL 1 GOTO err

REM запуск программы с тремя числовыми параметрами

test.exe 15 2 -8 > output.txt

IF ERRORLEVEL 1 GOTO err

FC /B output.txt 15+2+-8=9.txt

IF ERRORLEVEL 1 GOTO err

ECHO Program testing succeeded :-)

EXIT

:err

ECHO Program testing failed :-(

EXIT

Кроме того, нам понадобятся вспомогательные файлы, содержащие ожидаемый результат.

**-123.txt**

-123 = -123

**15+2=17.txt**

15 + 2 = 17

**15+2+-8=9.txt**

15 + 2 + -8 = 9

Для тестирования нашей программы вполне достаточно было проверить 6 вариантов ее запуска. Для тестирования более сложных программ потребуется проверить большее количество вариантов. Чтобы от тестов была ощутимая польза, необходимо покрыть ими, по меньшей мере, все ветки выполнения программы. И помните:

Тестирование способно обнаружить лишь факт наличия ошибок, а не их отсутствия.

**Задание:** На основе данной программы разработайте программу **eval.exe**, вычисляющую результат вычисления арифметического выражения, операции и операнды которого передаются через аргументы командной строки. **Вычисление выражения должно осуществляться без учета приоритета операторов, результат выводится с точностью до 3 знаков после запятой**. Например, результатом вычисления:

eval.exe 13 + 7 / 8 \* 3.8

должно быть число 9.500.

Должны поддерживаться следующие арифметические операции: сложение, вычитание, умножение и деление.

Программа должна корректно обрабатывать ошибки в передаваемых аргументах (недопустимые значения параметров и выражения в целом, например, деление на 0).

**В комплекте с программой должен поставляться набор файлов, позволяющий проверить корректность ее работы в автоматическом режиме**.

#### Задание 5. – 40 баллов

Разработайте программу, вычисляющую расстояние, которое пролетит тело вдоль горизонтальной оси, брошенное **в вакууме** под углом **α** к горизонту с некоторой начальной скоростью **Vo**. Вычисленное расстояние программа должна вывести в стандартный поток вывода.

Угол и начальная скорость вводятся пользователем из стандартного потока ввода. Угол задается в градусах, начальная скорость – в метрах в секунду. Ускорение свободного падения равно 9,8 м/с2.

Для вычисления значения тригонометрических функций синуса и косинуса следует воспользоваться встроенными функциями sin() и cos() стандартной библиотеки языка Си, подключаемых при помощи заголовочного файла **math.h**. Данные функции принимают значение угла, заданное в [радианах](http://ru.wikipedia.org/wiki/%D0%A0%D0%B0%D0%B4%D0%B8%D0%B0%D0%BD)[[14]](#footnote-14).

Для перевода градусов в радианы необходимо угол в градусах умножить на Пи и разделить на 180. **Значение числа Пи взять с точностью не менее 8 знаков после запятой**.

Программа должна вычислять расстояние, непрерывно запрашивая величину угла и начальную скорость до тех пор, пока пользователь не введет слово **exit**:

**Enter v0 (or type ‘exit’)**>*10*

**Enter a0 (or type ‘exit’)**>*30*

**Distance is: ...**

**Enter v0 (or type ‘exit’)**>*exit*

**Goodbye**

Выделить код вычисления расстояния полета тела в отдельную функцию **CalculateDistance()**.

1. **IDE** – Integrated Development Environment, интегрированная среда разработки, среда разработки, содержащая редактор исходных текстов программ, систему отладки, объединенная с компилятором или интерпретатором языка (или нескольких языков) программирования. Позволяет ускорить процесс разработки и отладки программ. [↑](#footnote-ref-1)
2. **Объектый файл** (англ. - object file) – файл с объектным кодом, получаемый в результате трансляции компилятором или ассемблером исходного текста программы. **Объектный код** (англ. object code) – смесь машинного кода процессора (или кода некоторой виртуальной машины) с некоторой служебной информацией (например, информацией о внешних ссылках), позволяющей в процессе компоновки собрать загрузочный модуль программы [↑](#footnote-ref-2)
3. **Компоновка** (англ. linking) – процесс подготовки загрузочного модуля (например, приложения или библиотеки), при котором производится связывание объектных модулей программы между собой (**разрешение внешних ссылок**) и с библиотеками времени исполнения (Run-time libraries). [↑](#footnote-ref-3)
4. Устанавливаемые дополнительно модули расширения (плагины) среды Visual Studio могут добавить возможность создания других типов проектов. [↑](#footnote-ref-4)
5. **COM** (англ. Component Object Model, модель компонентных объектов Microsoft) – стандартный механизм, включающий интерфейсы, с помощью которых одни объекты предоставляют свои сервисы другим. Является основой многих объектных технологий на платформе Windows, в том числе OLE и ActiveX. [↑](#footnote-ref-5)
6. Visual Studio позволяет пользователю выбрать один из нескольких профилей настроек клавиатуры, а также настроить клавиши по своему усмотрению. Указанные автором комбинации клавиш соответствуют профилю “Visual C++ 6”. [↑](#footnote-ref-6)
7. **Библиотека времени выполнения** (англ. - run time library или RTL) – файл, содержащий подпрограммы, прикомпоновываемые к программе во время ее исполнения (динамически компонуемая библиотека – dynamic linking library, DLL) или на этапе компоновки (статически компонуемая библиотека – static linking library). RTL обеспечивает среду выполнения программы, являясь прослойкой между приложением и операционной системой. [↑](#footnote-ref-7)
8. Когда в начале 1945 г. ЭВМ Mark I был остановлен для ремонта, Грейс Хопер (Grace Hopper) заметила в одном из реле мотылька, попадание которого, возможно и вызвало сбой. После этого операция восстановления работоспособности стала именоваться ею debugging. Это вполне вероятно, так как выделяемое компьютером тепло привлекает насекомых. [↑](#footnote-ref-8)
9. **Точка останова** (англ. breakpoint) – место в программе, заданное по адресу команды или какому-то условию, где ее нормальное выполнение прерывается для целей отладки и может быть возобновлено с помощью специальной команды отладчика. [↑](#footnote-ref-9)
10. **Единица компиляции** – файл, участвующий в процессе компиляции. Для приложений на языках C и C++ это файлы с одним из расширений: .с, .cpp, .cxx. Заголовочные файлы (.h, .hxx, .hpp) единицами компиляции не являются, однако могут подключаться в единицах компиляции и других заголовочных файлах [↑](#footnote-ref-10)
11. **Позиционная система счисления** – система счисления, в которой значение каждого числового знака (цифры) зависит от его позиции (разряда). [↑](#footnote-ref-11)
12. **Числа Фибоначчи** – элементы числовой последовательности 1, 1, 2, 3, 5, 8, …, в которой каждое последующее число равно сумме двух предыдущих чисел. [↑](#footnote-ref-12)
13. **fc.exe** – утилита, входящая в состав ОС Windows, выполяющая сравнение содержимого двух файлов и выводящая различия между ними. Сравнение может выполняться как в текстовом, так и в двоичном режиме. Нулевой код возврата сигнализирует об отсутствии различий между сравниваемыми файлами. [↑](#footnote-ref-13)
14. **Радиан** – основная единица измерения плоских углов в современной математике. Радиан определяется как угловая величина дуги единичной длины на окружности единичного радиуса. Таким образом, величина полного угла равна 2 радиан [↑](#footnote-ref-14)