ГОСУДАРСТВЕННОЕ БЮДЖЕТНОЕ ПРОФЕССИОНАЛЬНОЕ
ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ АРХАНГЕЛЬСКОЙ ОБЛАСТИ
«МИРНИНСКИЙ ПРОМЫШЛЕННО-ЭКОНОМИЧЕСКИЙ ТЕХНИКУМ»

МЕТОДИЧЕСКИЕ РЕКОМЕНДАЦИИ ПО ОП.09 ОСНОВЫ АЛГОРИТМИЗАЦИИ И ПРОГРАММИРОВАНИЯ

для специальности: 09.02.01 Компьютерные системы и комплексы

2022 г.

Методические рекомендации для ОП.09 Основы алгоритмизации и программирования разработаны для выполнения практических работ в среде программирования C# и составлены в соответствии с рабочей программой учебной дисциплины и учебным планом по специальности 09.02.01 «Компьютерные системы и комплексы».

Организация-разработчик: государственное бюджетное профессиональное образовательное учреждение Архангельской области «Мирнинский промышленно-экономический техникум»

Разработчики:
Кузнецова С.П., заведующий дневным отделением;

ОДОБРЕНЫ цикловой комиссией дисциплин специальностей 09.02.01 и 13.02.11	Составлены в соответствии с требованиями ФГОС по специальности среднего профессионального образования 09.02.01 «Компьютерные системы и комплексы» и учебным планом
Председатель цикловой комиссии В.И.Письменник	Заместитель директора техникума по учебной работе М.Н.Венедиктова

Глава 1. ПОНЯТИЕ АЛГОРИТМА И СПОСОБЫ ЕГО ОПИСАНИЯ

1.1 Этапы решения задачи

Можно выделить следующие основные этапы решения задачи:

- постановка (формулировка) задачи;
- построение модели, выбор метода решения задачи;
- разработка алгоритма;
- проверка правильности алгоритма;
- реализация алгоритма;
- анализ алгоритма и его сложности;
- отладка программы, обнаружение, локализация и устранение возможных ошибок;
- получение результата;
- составление документации.

Понятие алгоритма занимает центральное место в вычислитель- ной математике и программировании. Справедливо следующее опре- деление. *Алгоритм* – строгая и четкая система правил, определяющая последовательность действий над некоторыми объектами и после ко- нечного числа шагов приводящая к достижению поставленной цели.

Прежде чем понять задачу, необходимо ее четко сформулировать. Это условие не является достаточным для понимания задачи, но оно абсолютно необходимо. Следующий важный шаг в решении задачи – формулировка для нее математической модели. Выбор модели и реа- лизация алгоритма в значительной степени может повлиять на эффек- тивность алгоритма решения задачи. Все перечисленные выше этапы нельзя рассматривать независимо друг от друга. В особенности пер- вые три сильно влияют на последующие.

Наиболее распространенная процедура доказательства правиль- ности программы (следующий этап) – это прогон ее на ранних тестах. Однако это не исключает все сомнения. Необходимо доказательство конечности алгоритма, при котором будут проверены все подходящие входные данные и получены все подходящие выходные данные.

Реализация алгоритма – процесс корректного преобразования алгоритма в машинную программу. Требует также построения целой системы структур данных для представления модели.

Задача анализа алгоритма и его сложности – получение оценок или границ для объема памяти или времени работы алгоритма. Пол- ный анализ способен выявить узкие места в программах. Критерии оценок алгоритмов будут рассмотрены далее.

Эксплуатации программы предшествует отладка – исправление синтаксических и логических ошибок. Процесс проверки программы включает экспериментальное подтверждение того факта, что про- грамма делает именно то, что должна делать. Обычно множество вхо- дов огромно, и полная проверка невозможна. Необходимо выбрать множество вводов, которые проверяют каждый участок программы. Программы следует тестировать также для того, чтобы определить их вычислительные ограничения.

Этап документации не является последним шагом в процессе по- строения алгоритма. Он должен переплетаться со всем процессом по- строения алгоритма для того, чтобы была возможность понять про- граммы, написанные другими.

Последние этапы обеспечивают обратную связь, которая может заставить пересмотреть некоторые из предшествующих этапов.

1.1. Свойства алгоритма

Характерными свойствами алгоритма являются определенность, массовость и результативность.

Определенность (детерминированность) алгоритма предполагает такое составление предписания, которое не оставляет места для раз- личных толкований или искажений результата, т.е. последователь- ность действий алгоритма строго и точно определена.

Массовость определяет возможность использования любых ис- ходных данных из некоторого допустимого множества. Правило, сформулированное только для данного случая, не является алгорит- мом (например, таблица умножения не является алгоритмом, а прави- ло умножения «столбиком» есть алгоритм).

Результативность (конечность) алгоритма означает, что при лю- бом допустимом исходном наборе данных алгоритм закончит свою работу за конечное число шагов.

1.2. Классификация алгоритмов

По типу используемого вычислительного процесса различают ли- нейные (прямые), разветвляющиеся и циклические алгоритмы.

Линейные алгоритмы описывают линейный вычислительный про- цесс, этапы которого выполняются однократно и последовательно один за другим.

Разветвляющийся алгоритм описывает вычислительный процесс, реализация которого происходит по одному из нескольких заранее

предусмотренных направлений. Направления, по которым может следовать вычислительный процесс, называются ветвями. Выбор конкретной ветви вычисления зависит от результатов проверки вы- полнения некоторого логического условия. Результатами проверки являются: «истина» (да), если условие выполняется, и «ложь» (нет), при невыполнении условия.

Циклический алгоритм описывает вычислительный процесс, эта- пы которого повторяются многократно. Различают простые циклы, не содержащие внутри себя других циклов, и сложные (вложенные), со- держащие несколько циклов. В зависимости от местоположения усло- вия выполнения цикла различают циклы с предусловием и циклы с постусловием. В соответствии с видом условия выполнения циклы делятся на циклы с параметром и итерационные циклы.

Многие из реально существующих алгоритмов имеют смешан- ный характер, т.е. могут содержать линейные участки, разветвления, циклы с известным количеством повторений и итерационные циклы. В связи с этим составление алгоритмов сразу в законченной форме затруднено. Поэтому для составления сложных алгоритмов рекомен- дуется использовать нисходящее проектирование программ (метод пошаговой детализации, метод последовательных уточнений). Его суть: первоначально продумывается общая структура алгоритма, без детальной проработки его отдельных частей. Далее прорабатываются отдельные блоки, не детализированные на предыдущем шаге. Таким образом, на каждом шаге разработки уточняется реализация фраг- мента алгоритма, т.е. решается более простая задача.

1.3. Способы описания алгоритмов

Существуют следующие способы описания алгоритмов:

1) запись на естественном языке (словесное описание);

2) изображение в виде схемы (графическое описание);

3) запись на алгоритмическом языке (составление программы). Способы словесного описания алгоритмов отличаются применяе-
мыми метаязыками (языки, предназначенные для описания языка про-
граммирования).

Например, словесное описание алгоритма решения квадратного уравнения $a \cdot x^2 + b \cdot x + c = 0$ будет выглядеть следующим образом:

1) $D := b^2 - 4 \cdot a \cdot c$;

2) если $D < 0$, идти к 4;

3) $x_1 := (-b + \sqrt{D}) / (2 \cdot a)$;
$x_2 := (-b - \sqrt{D}) / (2 \cdot a)$;

4) Останов.

Основной недостаток словесного описания – плохая наглядность.

Графическое описание алгоритма – это представление алгоритма в виде схемы (двухмерного рисунка), состоящей из последовательно- сти блоков (геометрических фигур), каждый из которых отображает содержание очередного шага алгоритма (управляющей структуры). Внутри фигур кратко записывают выполняемое действие. Такую схему называют блок-схемой алгоритма.

На рис. 2.1. приведены основные условные обозначения, исполь- зуемые при графической записи алгоритма.

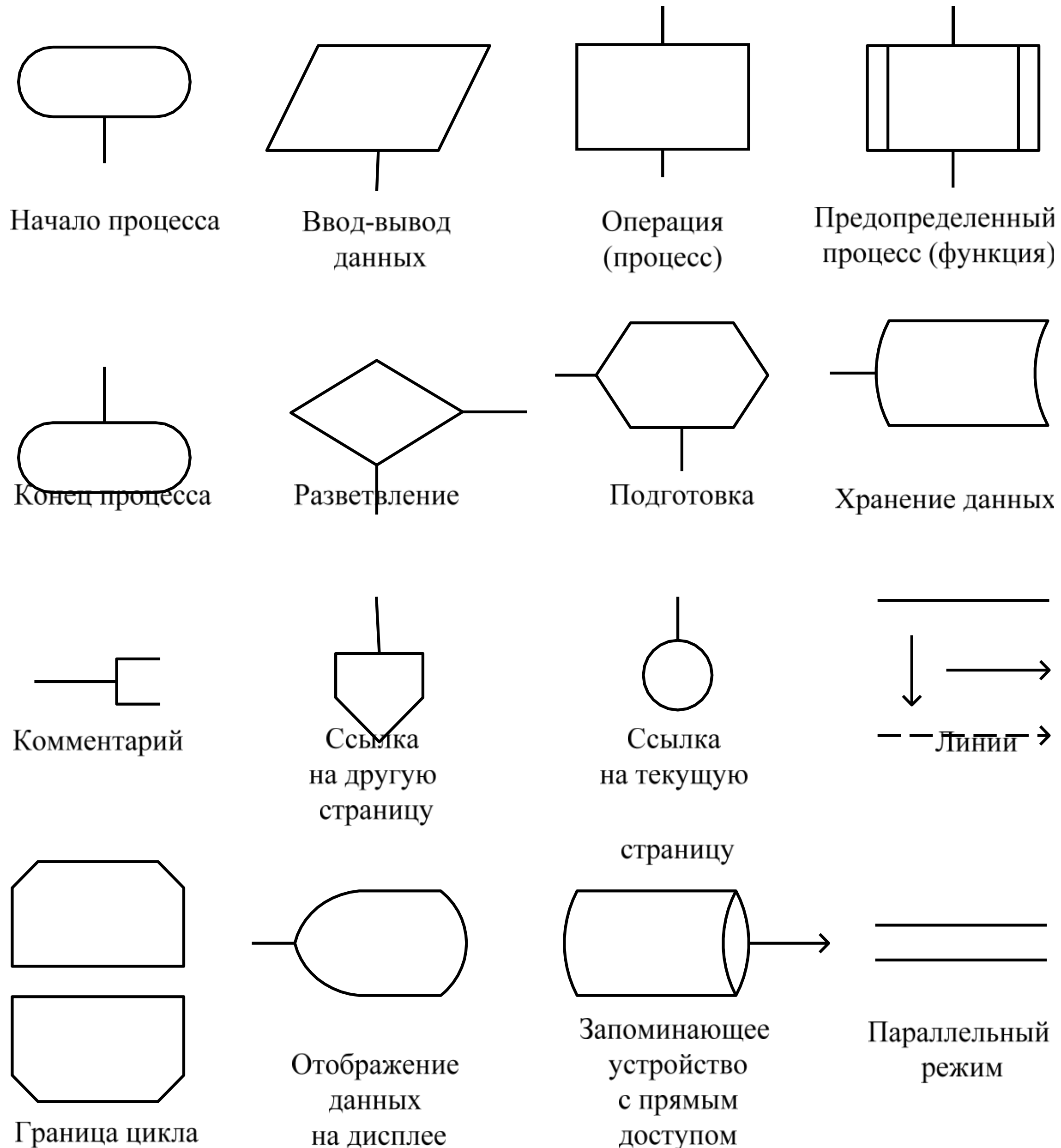

Рис. 2.1 Условные обозначения, используемые в блок-схемах

Для стандартизации и унификации языка схем алгоритмов в 1992 г. был принят ГОСТ 19.701–90 «Единая система программной документации. Схемы алгоритмов, программ, данных и систем. Условные обозначения и правила выполнения» [1]. В настоящее время данный стандарт продолжает действовать в Республике Беларусь.

На рис. 2.2 приведена блок-схема алгоритма нахождения макси- мального из трех заданных чисел, которая использует разветвляю- щийся алгоритм.

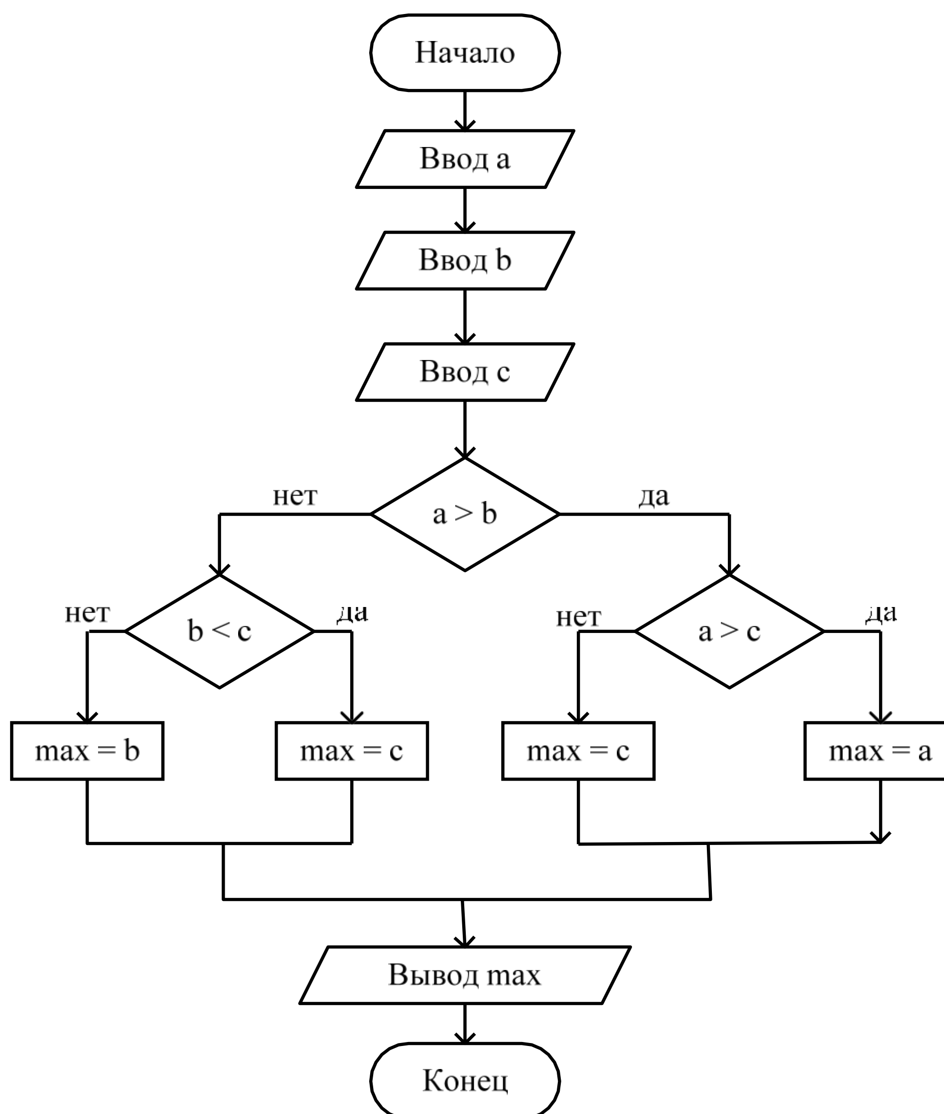

Рис. 2.2. Блок-схема алгоритма нахождения максимального числа

На рис. 2.3 приведена блок-схема алгоритма вычисления графика функции в заданном интервале, в которой используется циклический вычислительный процесс с известным числом повторений.

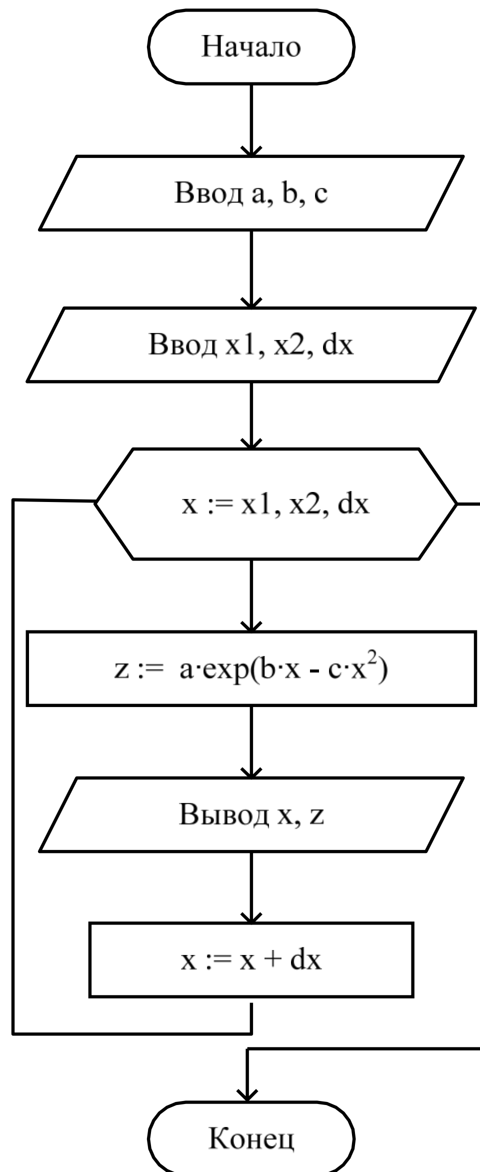

Рис. 2.3. Блок-схема алгоритма вычисления графика функции в заданном интервале

Рассмотрим еще один пример, в котором используется цикл с не- известным числом повторений. Необходимо вычислить значение функции $Y = \sin x$ через разложение функции в бесконечный ряд:

$$Y = \sin x = x - x^3 / 3! + x^5 / 5! - x^7 / 7! + \dots .$$

Вычисления прекращаются, когда разность между модулями двух соседних слагаемых станет меньше величины $e = 0{,}0001$. Схема алгоритма решения данной задачи имеет вид, представленный на рис. 2.4. Как видно, здесь проверка условий окончания циклических вычислений осуществляется в конце цикла.

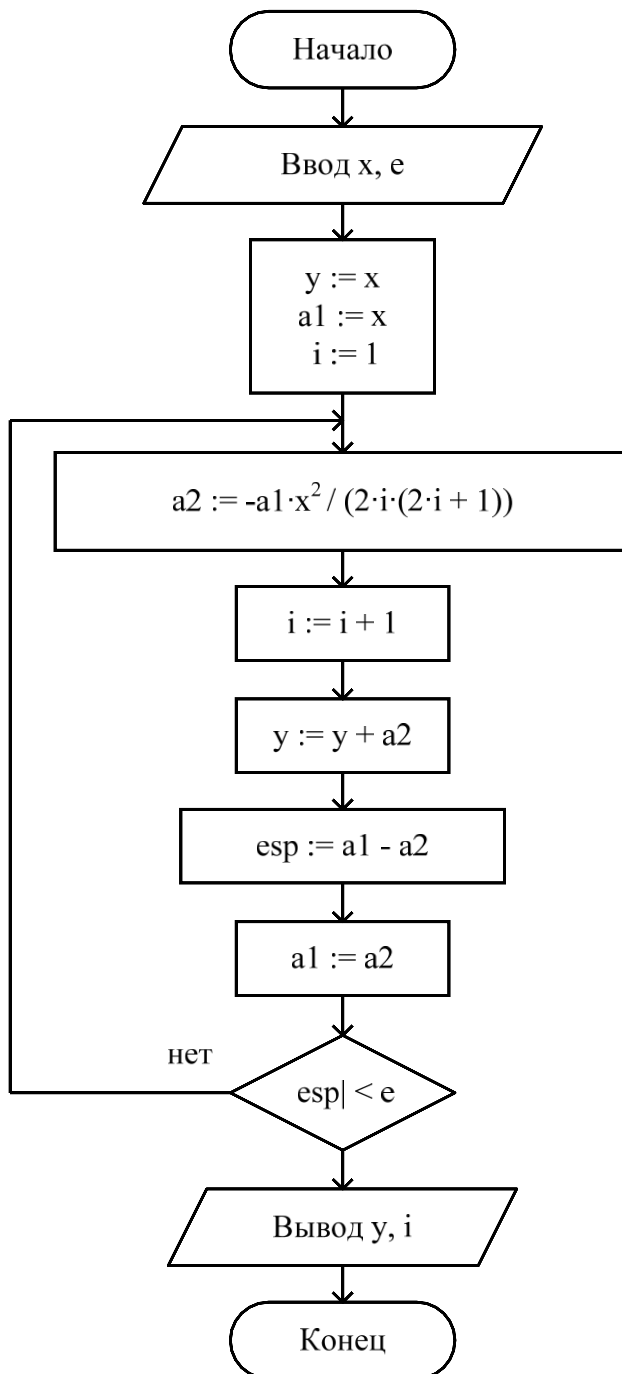

Рис. 2.4. Блок-схема алгоритма вычисления значения функции с переменным количеством шагов

Понятия алгоритма и программы не имеют четкого разграни- чения. Так, программа, записанная на алгоритмическом языке – это окончательный вариант алгоритма решения задачи, ориентирован- ный на конкретного исполнителя (компьютер или язык програм- мирования).

1.4. Понятие структурного программирования

В 70-е годы прошлого века возник новый подход к разработке ал- горитмов и программ, который получил название структурного проек- тирования программ. К его достоинствам можно отнести: более высо- кую производительность; читаемость программ; простоту тестирова- ния и эффективность программ.

Одна из концепций структурного программирования – нисходя- щее проектирование. Это средство разбиения большой задачи на меньшие подзадачи так, чтобы каждую подзадачу можно было рас- сматривать независимо.

Все операции в программе, построенной на основе структурного программирования, должны представлять собой либо исполняемые в линейном порядке выражения, либо одну из следующих управляющих конструкций: вызовы подпрограмм; вложенные на произвольную глу- бину операторы условия; циклические операторы.

1.5. Практические задания

1. Составить алгоритм и блок-схему вычисления вещественной функции $ae^{(bx - cx)}$ в заданном интервале с заданным шагом.
2. Изобразить алгоритм и блок-схему нахождения минимального (максимального) числа из 4 заданных.
3. Составить алгоритм и блок-схему вычисления факториала.
4. Представить алгоритм и блок-схему ввода в массив статиче- ской последовательности неизвестной длины.
5. Составить алгоритм и блок-схему нахождения суммы чисел от 1 до заданного числа *n*.

Глава 2. ИНТЕГРИРОВАННАЯ СРЕДА РАЗРАБОТКИ MICROSOFT VISUAL C++

3.1. Введение

Минимальная адресованная ячейка памяти состоит из 8 двоичных позиций. В каждую позицию может быть записан либо 0, либо 1. В одной ячейке из 8 двоичных разрядов помещается объем информа- ции в 1 байт, поэтому объем памяти принято оценивать количеством байт (1 024 байт = 1 Кб, 1 048 576 байт = 1 Мб).

При размещении данных производится их запись с помощью нулей и единиц – кодирование, при котором каждый символ заменяется по- следовательностью из 8 двоичных разрядов в соответствии со стан- дартной кодовой таблицей ASCII (прил. 1). Например, D (код 68) → 01000100; F (код 70) → 00100110; 4 (код 52) → 00110100.

Программа – это последовательность команд (инструкций), которые помещаются в памяти и выполняются процессором в указанном порядке. Программа записывается на языке высокого уровня, наиболее удобном для реализации алгоритма решения определенного класса задач. Для ав- томатизации процесса разработки программ используются интегриро- ванные среды разработки (IDE – Integrated Development Environment). Обычно они содержат: входной язык системы; транслятор с входного языка на машинный язык; редактор связей; библиотеки программ; сред- ства отладки; обслуживающие (сервисные) программы и документацию.

Исходный текст программы, введенный с помощью клавиатуры в память компьютера называется *исходный модуль* (*Source module*) (в C++ файл имеет расширение **.cpp*).

Транслятор – программа, осуществляющая перевод текстов с од- ного языка на другой, т.е. с входного языка системы программирова- ния на машинный язык. Одной из разновидностей транслятора является *компилятор* (*Compiler*), в котором трансляция отделена от выполне- ния программ. Другим видом транслятора является интерпретатор. *Интерпретатор* (*Interpreter*) выполняет созданную программу путем одновременного анализа и реализации предписанных действий, при использовании отсутствует разделение на две стадии – перевод и вы- полнение. Большинство трансляторов языка С/С++ – компиляторы.

Результат обработки исходного модуля компилятором – *объект- ный модуль* (*Object module*) (расширение **.obj*), это незавершенный ва- риант машинной программы (например, к нему должны быть присоеди- нены модули стандартных библиотек). Он содержит текст программы

на машинном языке и дополнительную информацию, обеспечивающую объединение этого модуля с другими независимо транслируемыми мо- дулями. Объектный модуль готов к редактированию связей.

Исполняемый (абсолютный, загрузочный) модуль создает вторая специальная программа – *компоновщик*. Ее еще называют *редактором связей* (*Linker*). Он и создает модуль, пригодный для выполнения на основе одного или нескольких объектных модулей. Компоновщик со- бирает, устанавливает связи между модулями и создает еще один вид программного модуля – загрузочный модуль.

Загрузочный модуль (*Load module*) (расширение *.*exe*) – это про- граммный модуль, представленный в форме, пригодной для выполнения. На рис. 3.1 показана схема решения задачи с помощью компью- тера, в которой учтена необходимость использования языка программирования.

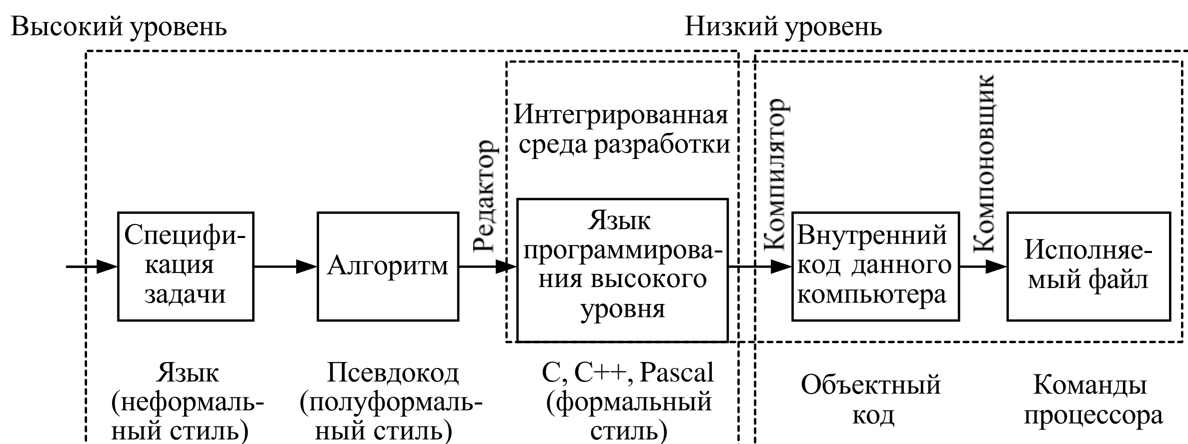

Рис. 3.1. Схема решения задачи

Библиотека программ представляет собой программы, предназна- ченные для решения распространенных задач. Программы, включенные в библиотеку, оформляются специальным образом, облегчающим их вы- зов, использование, передачу входных данных и результатов.

3.2. Интегрированная среда разработки Visual Studio 2008 и Visual C++

Среда разработки Microsoft Visual Studio (VS) 2008 отличается от предыдущих версий повышением производительности разработ- чиков, поддержкой новейших технологий и управлением всем цик- лом создания приложений. Существуют следующие варианты по- ставки: VS Team System 2008, VS Team System 2008 Team Suite,

VS Team System 2008 Architecture Edition, VS Team System 2008 Development Edition, VS Team System 2008 Test Edition, VS Team System 2008 Database Edition, VS Team System 2008 Team Foundation Server, VS Team System 2008 Test Load Agent, Visual C++ 2008 Express Edition и др.

Visual C++ состоит из следующих компонентов.

Средства компилятора Visual C++ 2008. Компилятор поддер- живает как традиционную разработку с использованием машинного кода, так и разработку с использованием платформ виртуальных ма- шин, таких как среда CLR (Common Language Runtime). Компилятор продолжает напрямую поддерживать архитектуру x86 и оптимизирует производительность кода для обеих платформ.

Библиотеки Visual C++ содержат общепризнанную библиотеку шаблонных классов ATL (Active Template Library), библиотеки MFC (Microsoft Foundation Class) и стандартные библиотеки, такие как биб- лиотеки стандартных шаблонов STL (Standard Template Library) и биб- лиотеки времени выполнения языка C CRT (Common Runtime Library). Библиотека CRT включает альтернативные функции с улучшенной безопасностью для функций с известными проблемами безопасности. Библиотека STL/CLR позволяет разработчикам, использующим управ- ляемый код, использовать также и возможности библиотеки STL.

Среда разработки Visual C++ предоставляет всестороннюю под- держку при управлении проектами и их настройке (включая улучшен- ную поддержку больших проектов), редактировании исходного кода, просмотре исходного кода, а также мощные средства отладки.

Кроме традиционных приложений с пользовательским интерфей- сом Visual C++ позволяет разрабатывать веб-приложения, приложения интеллектуальных клиентов для Windows, решения для мобильных устройств, использующих «тонкие клиенты» и «интеллектуальные клиенты». Язык C++, являющийся самым популярным в мире языком уровня системы, и Visual C++ вместе предоставляют разработчику высококлассное средство мирового уровня для построения программ- ного обеспечения.

3.3. Создание проекта

Программы (чаще именуемые приложениями), создаваемые в среде разработки Microsoft Visual Studio 2008 (MS VS), представляются в виде двух типов контейнеров, вложенных один в другой. Один (главный контейнер) называется *решения* (*Solutions*), а другой – *про- ект* (*Project*). Проект определен как конфигурация, объединяющая

группу файлов. Окно диспетчера решения может содержать множест- во проектов, а проект содержит множество элементов. Такой подход к оформлению приложения позволяет работать с группой проектов как с одним целым, что ускоряет процесс разработки приложений.

При написании программы на Visual C++ с помощью Visual Studio первым этапом является выбор типа проекта. Для каждого типа про- екта Visual Studio устанавливает параметры компилятора и генерирует стартовый код.

Консольное приложение

При изучении C/C++ будем пользоваться специальным видом приложений – консольными приложениями, которые формируются на основе заранее заготовленных в среде проектирования шаблонов. Консольные (опорные, базовые) приложения – это приложения без графического интерфейса, которые взаимодействуют с пользователем через специальную командную строку или запускаются специальной командой из главного меню среды. Шаблон консольного приложения добавляет в создаваемое приложение необходимые элементы, после чего разработчик вставляет в шаблон свои операторы C/C++. Обще- ние с пользователем происходит через специальное окно, открываю- щееся средой после запуска приложения (в него выводятся сообщения программы, вводятся данные и выводится результат).

Создание нового проекта

Для создания нового проекта нужно выбрать команду меню **File/New/Project…**. Система предложит создать новый проект, и поя- вится диалог, представленный на рис. 3.2.

В закладке *Visual C++* в списке различных типов выполняемых файлов необходимо выбрать *Win32*, а в окне *Templates – Win 32 Console Application*. Затем следует набрать имя проекта (например, *My_first*) в поле *Name* и имя каталога, в котором будут храниться все файлы, относящиеся к данному проекту, в поле *Location*. После этого нажать кнопку *OK*.

На следующем шаге мастера необходимо нажать кнопку *Finish*, и результатом будет заготовка консольного приложения (рис. 3.3).

Как и другие окна операционной системы (ОС) Windows, окно среды разработки содержит строку заголовка, меню и панели инстру- ментов. В рабочей области среды разработки содержится окно редак- тора для ввода программного кода, окно *Обозреватель решений и про- ектов* (*Solution Explorer*).

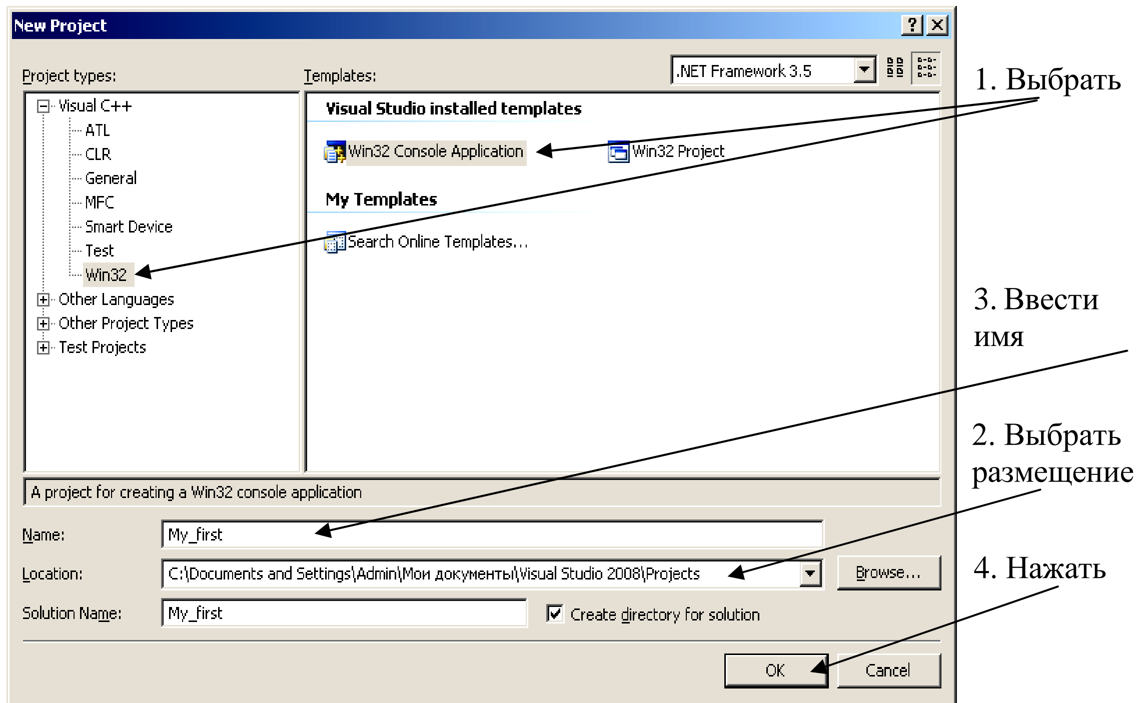

Рис. 3.2. Диалог создания нового проекта

Заготовка состоит из заголовка функции `int _tmain(int argc, _TCHAR* argv[])` и тела, ограниченного фигурными скобками (рис. 3.3). Теперь можно набирать текст как в обычном текстовом редакторе, работать необходимыми для ввода и редактирования клавишами.

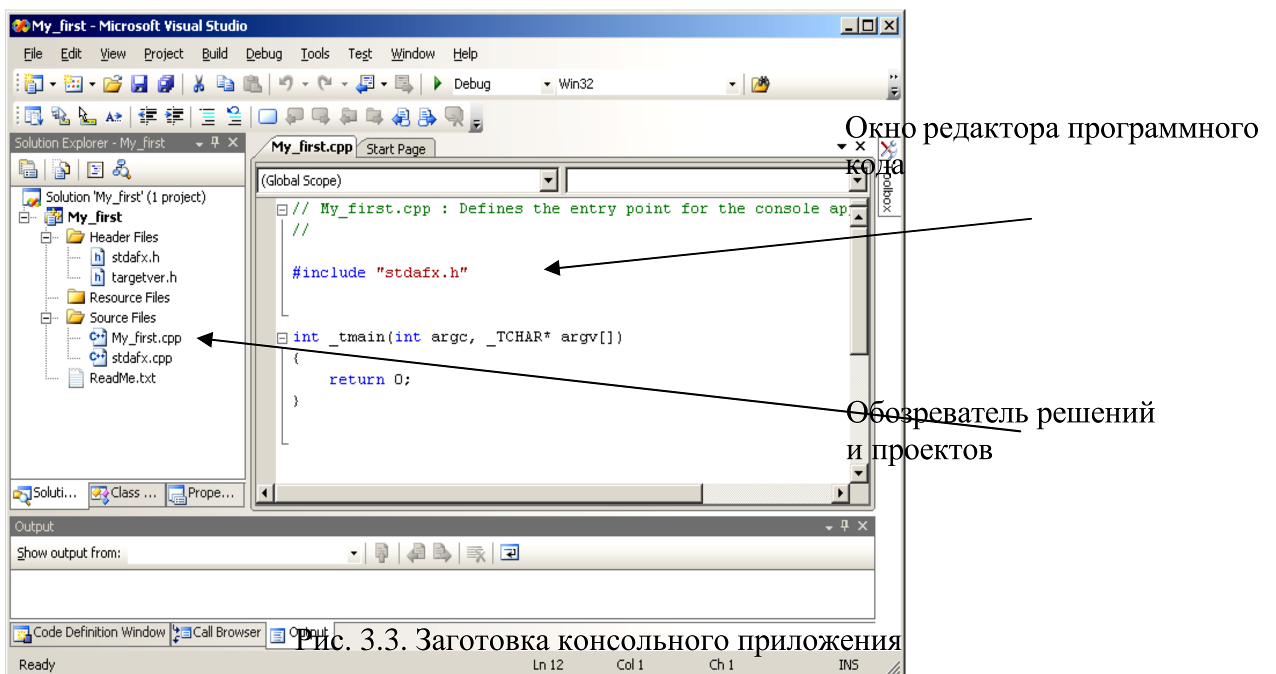

Рис. 3.3. Заготовка консольного приложения

Созданный по шаблону проект содержит: My_first.cpp – это главный исходный файл и точка входа в создаваемое приложение; stdafx.cpp – подключает специальный файл для компиляции прило- жения; stdafx.h – подключает специальные файлы для компиляции приложения, со следующим содержанием:

```
#pragma once

#include "targetver.h"

#include <stdio.h>
#include <tchar.h>
```

targetver.h – позволяет использовать специфические свойства Windows Vista. Содержимое файла:

```
#pragma once
#ifndef _WIN32_WINNT
#define _WIN32_WINNT 0x0600
#endif
```

ReadMe.txt – файл, описывающий некоторые из созданных шаблонов.

3.4. Загрузка существующего приложения

Для запуска интегрированной среды разработки и загрузки суще- ствующего приложения можно использовать: «ручной» запуск среды с последующим открытием решения или открытие приложения с ав- томатическим запуском среды разработки.

При первом варианте необходимо: запустить интегрированную среду – **Пуск/Программы/Microsoft Visual Studio 2008/Microsoft Visual Studio 2008**; открыть существующее приложение командой **File/Open/Project/Solution**; в появившемся диалоговом окне найти файл решения *.sln и щелкнуть по кнопке **Open**. Решение будет от- крыто, окно редактирования активизировано, и в это окно будет за- гружен исходный код программы.

При втором варианте загрузки приложения необходимо: найти файл решения *.sln с помощью стандартных средств ОС; открыть приложение двойным щелчком; в результате произойдет автоматиче- ский запуск интегрированной среды разработки.

3.5. Ввод и редактирование программного кода

Редактор обеспечивает все стандартные действия, которые дос- тупны для любого другого редактора (набор программного кода,

редактирование, копирование, вставка, поиск и т.д.), и кроме того, об- ладает большим набором дополнительных возможностей. Редактор программного кода поддерживает оперативную (в процессе ввода тек- ста) проверку программы.

Для получения справочной информации нужно установить тек- стовый курсор на элемент программы, для которого необходимо на- личие справки, и нажать клавишу **F1** (справка будет выдана на анг- лийском языке; получение справки возможно только в случае, если на компьютере установлена справочная служба MSDN Library). Полу- чаемая информация содержит примеры практического использования рассматриваемых элементов.

IntelliSense

Для быстрого и безошибочного набора программного кода в ре- дакторе среды MS VS имеется специальная служба **IntelliSense**, кото- рая обеспечивает: отображение списка методов и полей для классов, структур, пространства имен и других элементов кода (выбор нужного варианта может быть выполнен, например, при помощи двойного щелчка мыши на требуемой строке списка); отображение информации о параметрах для методов и функций (рис. 3.4.) (осуществляется авто- матически после ввода имени метода или функции); отображение краткого описания элементов кода программы (происходит при наве- дении указателя мыши на нужный элемент кода); завершение слов при наборе наименований команд и имен функций; автоматическое сопоставление правильности расстановки скобок (скобки },],) и `#endif` выделяются более темным цветом вместе с соответствующей открывающейся скобкой). Служба IntelliSense может быть отключена.

```
printf ("Введите число x\n");
float x;
scanf ("%f", &x);

float y = pow ((float) cos (exp (x)) + exp
printf ("P
                double pow(double _X, double _Y)
return 0;       double pow(double _X, int _Y)
                float pow(float _X, float _Y)
                float pow(float _X, int _Y)
                long double pow(long double _X, long double _Y)
                (+1 overloads)
```

Рис. 3.4. Служба IntelliSense, отображение функций

Цветовая индикация текста в коде программы

Слова исходного текста программы отображаются разными цве- тами в зависимости от характера текста. С помощью цветов выделя- ются комментарии, ключевые слова, имена классов, имена объектов, символьные строки-литералы. По умолчанию принят следующий ва- риант индикации:

– комментарии – зеленый цвет;
– ключевые слова – синий цвет;
– символьные строки-литералы – коричневый цвет;
– весь остальной текст, включая имена объектов – черный цвет.

Скрытие фрагментов текста

Текстовый процессор позволяет скрывать (не отображать) фраг- менты исходного кода, которые ограничены слева вертикальными ли- ниями с флажками в виде прямоугольников (рис. 3.5).

Установленный символ «+» в флажке означает, что фрагмент кода скрыт, а символ «–» означает, что скрытого текста нет. Выполняя щелчки на флажках, можно скрывать или раскрывать фрагменты ис- ходного кода.

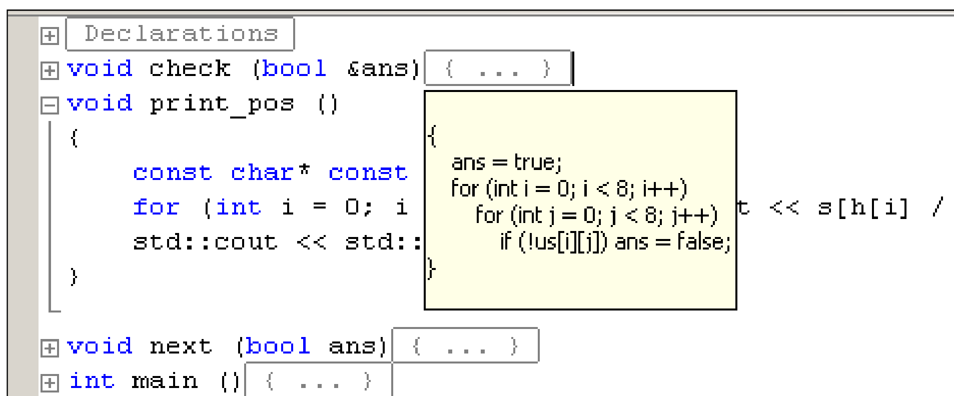

Рис. 3.5. Всплывающее окно просмотра скрытого текста

Имеется возможность просмотреть скрытый код через всплы- вающее окно (рис. 3.5). Это окно используется для оперативного про- смотра скрытого кода без его раскрытия. Оперативный просмотр вы- полняется удержанием курсора мыши в рамке с многоточием. Рамка отображается на месте скрытого кода.

Скрытие кода позволяет сосредоточить внимание на том фраг- менте кода, который редактируется в данный момент, если убрать из поля зрения остальные фрагменты. Особенно удобно применять эту функцию при работе с файлами большого размера. В этом случае

скрытие является хорошей альтернативой листанию файла в окне ре- дактирования.

Редактор среды Visual C++ многооконный, можно открыть сколько угодно окон с текстами файлов, заголовков и программ. Для переключения между этими окнами используется комбина- ция **Ctrl+F6**.

3.6. Изменение структуры приложения

При разработке приложения может возникнуть необходимость внесения изменения не только в исходный код, но и в состав файлов проекта. Изменение структуры проекта, и как следствие, структуры приложения, можно выполнить с помощью проводника проектов и решений (рис. 3.3).

В окне обозревателя решений выделите проект (в примере папка `My_first`). В результате станут доступны средства изменения струк- туры проекта, в частности: добавление файла к проекту; исключение файла из проекта; исключение файла из проекта и его физическое уда- ление; переименование файлов проекта.

Для выполнения указанных действий надо открыть папку проекта. Щелчком левой кнопки мыши указывается нужный файл, затем щелч- ком правой кнопки активизируется всплывающее меню.

Команда **Exclude From Project** позволяет исключить файл из проекта, при этом файл физически сохраняется (в той папке, где раз- мещен). Команда **Delete** позволяет исключить файл из проекта, при этом файл физически уничтожается. Команда **Rename** позволяет на- значить файлу новое имя. Подключение к проекту существующего файла выполняется командой **Add/Existing Item**….

В открывшемся диалоговом окне с использованием стандартных средств ОС Windows можно добавить требуемый файл. Этот файл бу- дет включен в проект. Физическое копирование файла в папку проек- та не требуется. Аналогично можно добавлять к текущему решению проекты и удалять их.

3.7. Выполнение приложения в режиме отладки Средства

отладки программ

Процесс отладки состоит из многократных попыток выполнения
программы на компьютере и анализа получившихся результатов. Ошибки, допускаемые при написании программ, разделяются на син- таксические и логические.

Синтаксические ошибки – нарушение формальных правил написа- ния программы на конкретном языке, обнаруживаются на этапе транс- ляции и могут быть легко исправлены. Трансляторы выдают информа- цию о синтаксических ошибках, указывают места ошибок и их характер. *Логические ошибки* – ошибки алгоритма и семантические, которые могут быть исправлены только разработчиком программы. Причина ошибки алгоритма – несоответствие построенного алгоритма ходу полу- чения конечного результата сформулированной задачи. Причина семан- тической ошибки – неправильное понимание смысла операторов языка. На этапе редактирования связей обнаруживаются ошибки, свя- занные с неправильным оформлением функций, ошибки в командах вызова функций и программ из библиотеки. На этапе выполнения об- наруживаются логические ошибки программы (например, деление на ноль, бесконечный цикл и т.п.). Среда отладки позволяет в пошаговом режиме обнаружить и локализовать ошибку.

Если программа содержит синтаксические ошибки, при выполне- нии построения они автоматически отображаются в окне *Outprut* (Вы- вод), по умолчанию расположенном в нижней части окна интегриро- ванной оболочки (рис. 3.6). Каждое сообщение начинается с имени ис- ходного файла и номера строки (в скобках), где обнаружена ошибка или предупреждение. Следом за номером строки идет номер и краткое опи- сание предупреждения или ошибки. По номеру можно найти описание ошибки в документации. Лучше всего добиться, чтобы в окончательном варианте не было ни того, ни другого, хотя с предупреждениями испол- няемый файл создается и может быть запущен. Если дважды щелкнуть на строке с сообщением об ошибке, то среда автоматически переклю- чится в окно редактирования и сама укажет на ошибочный фрагмент программы (рис. 3.6). Следует заметить, что компилятор может «не очень точно» локализовать место возникновения ошибки, особенно ес- ли это пропущенная скобка или точка с запятой. В этом случае надо внимательно проверить текст над указанной строкой. Исправив все ошибки, необходимо повторить построение исполняемого файла.

После того, как программа становится работоспособной, проводится ее тестирование – проверка правильности ее функционирования на раз- личных наборах исходных данных из диапазона допустимых значений.

Даже если синтаксических ошибок нет, приложение может рабо- тать не так, как ожидалось. Наиболее эффективные средства интегри- рованного отладчика: выполнение программы по шагам, просмотр значений любых переменных в любой точке программы, задание то- чек останова и др.

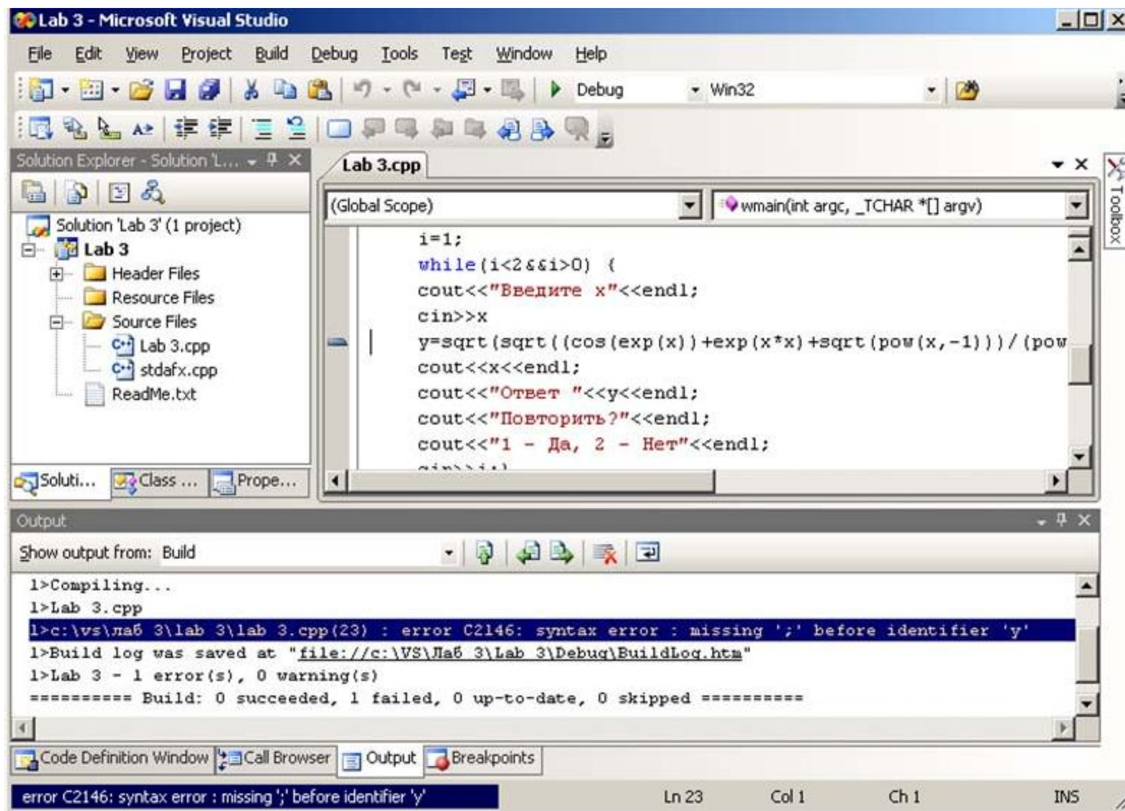

Рис. 3.6. Построение проекта с ошибками

При анализе сообщений, выдаваемых транслятором, необходимо учитывать следующие факторы:

– транслятор при запуске анализирует весь исходный код и пыта- ется выявить все ошибки (*Errors*). Поэтому не исключено, что одна ошибка может повлечь за собой множество сообщений. Например, ошибка в объявлении переменной приведет к появлению сообщений об ошибках в тех строках исходного кода, где эта переменная исполь- зуется. Признаком такого рода ошибки являются многочисленные со- общения однотипного характера. Не пытайтесь исправить абсолютно все ошибки за один раз. После внесения нескольких исправлений в исходный код можно запустить приложение. Если были внесены исправления в строку с первопричинной ошибкой, многие (а возмож- но все) сообщения исчезнут;

– транслятор контролирует соблюдение формальных правил запи- си операторов. В некоторых случаях ошибка, допущенная в операто- ре, не нарушает синтаксических правил в нем, но приводит к наруше- нию правил в других операторах. В этих случаях транслятор фиксиру- ет следствие ошибки, а не ее первопричину. Поэтому не следует слепо доверять указаниям транслятора на характер и место ошибки.

Помимо сообщений об ошибках, транслятор может выдавать пре- дупреждения (*Warnings*). Предупреждения выдаются при обнаружении «подозрительных» с точки зрения логики операторов, хотя синтакси- ческие правила их записи не нарушены. По умолчанию предупрежде- ния не препятствуют построению решения и его выполнению. Тем не менее, стоит внимательно проанализировать предупреждения. Часто предупреждения являются косвенным признаком наличия в исходном коде логических ошибок.

По характеру использования средства отладки можно разделить на две группы: средства интерактивной отладки; средства планируе- мой отладки. Средства первой группы позволяют выполнить про- грамму по шагам. Каждый шаг соответствует выполнению кода, ука- занного в одной строке программы. На каждом шаге планируется один следующий шаг отладки, это позволяет реализовать гибкий сце- нарий отладки. Средства второй группы позволяют спланировать оп- ределенный сценарий процесса отладки на множестве шагов. Они хо- рошо приспособлены к многократным отладочным прогонам, но име- ют меньшую гибкость отладочных действий.

Трассировка выполняется с точностью до одной строки исходной программы, а контроль значений – с точностью до одного объекта. Поэтому рекомендуется не располагать в одной строке программы не- сколько операторов и не использовать слишком сложных выражений.

Средства интерактивной и планируемой отладки могут приме- няться в любой последовательности и в любом сочетании исходя из целей, которые преследуются в конкретном отладочном прогоне.

Управление средствами отладки выполняется «горячими клави- шами» или через главное меню среды разработки.

Выполнение приложения с использованием средств интерактивной отладки

Используют два способа пошагового выполнения приложения:

– без трассировки вызываемых методов (клавиша **F10** или икон- ка на панели инструментов *Debug* (рис. 3.7), или команда меню **Debug/Step Over**);

– с трассировкой вызываемых методов (клавиша **F11** или иконка на панели инструментов *Debug* (рис. 3.7), или команда меню **Debug/Step Into**).

Рис. 3.7. Панель инструментов *Debug*

При обоих способах производится останов перед выполнением текущей строки исходного кода. Различия между командами **Step Into** и **Step Over** проявляются только тогда, когда в программе встречается вызов функции. Если выбрать команду Step Into, то отладчик войдет в функцию и начнет выполнять шаг за шагом все ее операторы. При выборе команды **Step Over** отладчик выполнит функцию как единое целое и перейдет к строке, следующей за вызовом функции. Эту ко- манду удобно применять в тех случаях, когда в программе делается обращение к стандартной функции или созданной вами подпрограмме, которая уже была протестирована.

Для контроля значений полей и свойств объектов используются всплывающие окна. Необходимо подвести курсор мыши к имени ин- тересующего объекта и удержать его некоторое время. Появится всплывающее окно, в котором будет указано имя объекта (поля) и его текущее значение (рис. 3.8). Сдвиг курсора мыши приводит к исчез- новению этого окна.

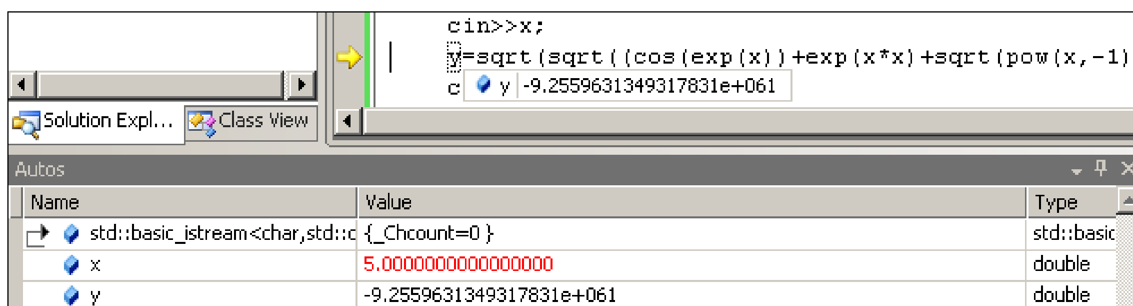

Рис. 3.8. Окно просмотра текущего значения переменной

После достижения целей, которые были запланированы на сеанс отладки, необходимо вывести приложение из отладочного режима командой **Shift+F5** или **Debug/Stop Debugging** (или одноименная кнопка на панели инструментов *Debug*).

Выполнение приложения с использованием средств планируемой отладки

При интерактивной или планируемой отладке используются точки останова (точки, в которых отладчик автоматически прерывает выполнение). Место точки основа можно выбрать по своему усмотре- нию (обычно место, в котором может содержаться ошибка). Точка оста- нова назначается щелчком левой кнопки мыши в специальном поле сле- ва от строки текста программы. Назначенная точка останова отмечается маркером в виде красного круга слева от строки (рис. 3.9). Повторный

щелчок левой кнопки мыши на маркере точки останова приводит к ее отмене. Точку останова можно назначить и отменить клавишей **F9** или командой меню **Debug/Toggle Breakpoint**. В этом случае она устанавли- вается на той строке кода программы, где помещен курсор.

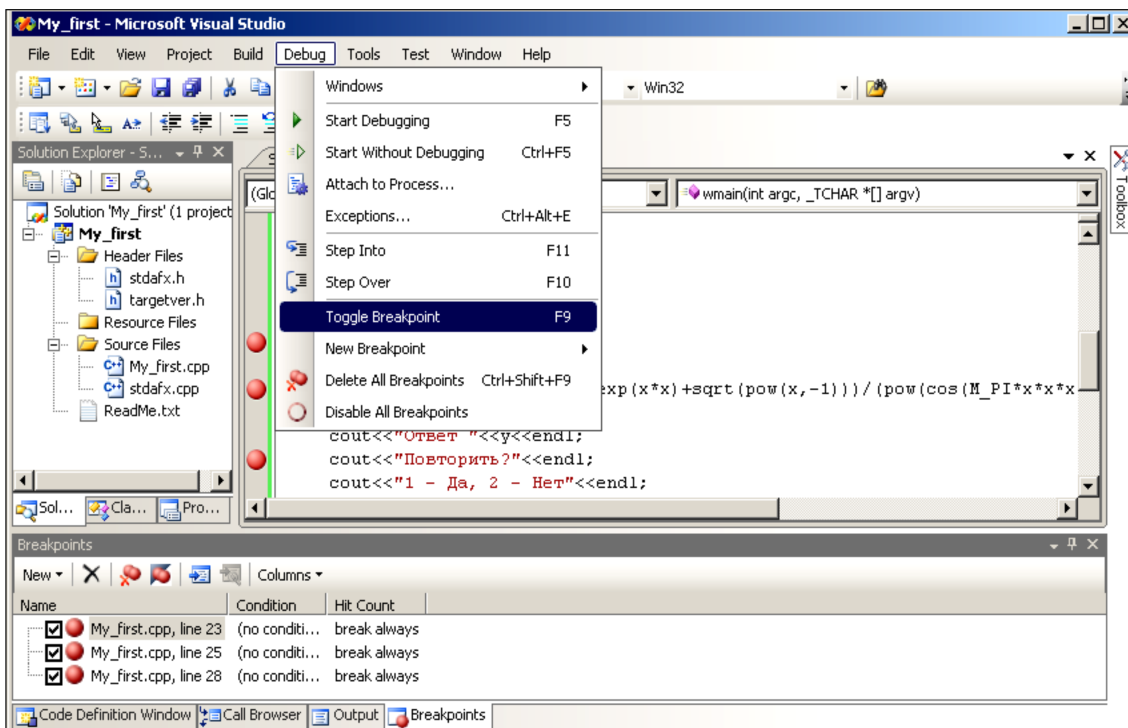

Рис. 3.9. Способы установки точки останова

После назначения точки останова приложение запускается в ре- жиме отладки функциональной клавишей **F5** или командой меню **Debug/Start**. Приложение будет выполнено до точки останова. Дос- тигнутая точка останова отмечается маркером в виде стрелки внутри маркера точки. Выполнить приложение до очередной точки останова можно, повторив указанные команды.

Для контроля значений рекомендуется использовать средства непрерывного контроля состояния объектов. Имена интересующих объектов после запуска программы в режиме отладки заносят в спе- циальное окно просмотра *Watch* (рис. 3.10). Имя заносится в столбец *Name*. Последующее нажатие *ENTER* или щелчок клавишей мыши на строке с именем приводит к появлению в поле (в столбце) *Value* зна- чения переменной (объекта).

Если рядом с именем переменной стоит знак плюс, то для этой переменной может быть отображена дополнительная информация (массивы, указатели или объекты класса). Если нажать <F10> два раза

и щелкнуть на «+» возле имени переменной, то отладчик отобразит значение, хранимое в памяти по адресу, содержащемуся в указателе.

Окно *Watch* также позволяет изменять значения переменных, за которыми ведется наблюдение. В том случае, когда ясно, что отобра- жаемое значение не верно, можно установить корректное значение и продолжить поиск ошибок. Это средство можно использовать для пропуска первых шагов в цикле с большим количеством итераций. Чтобы изменить значение, выполните двойной щелчок на отображае- мом значении переменной и введите новое.

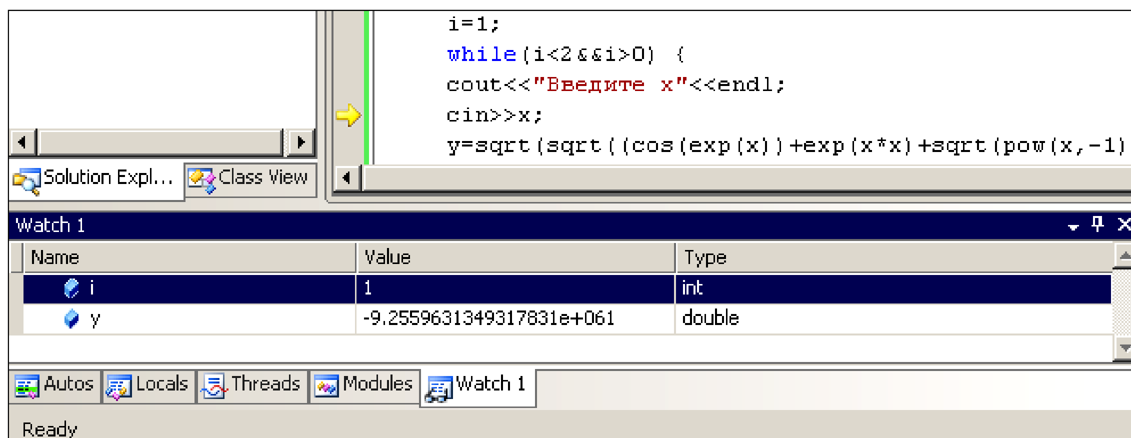

Рис. 3.10. Окно просмотра *Watch*

Может быть назначено до четырех окон просмотра с помощью команд меню: **Debug/Windows/Watch/Watch1** (*Watch2*, *Watch3* или *Watch4*). Переход от одного окна просмотра к другому выполняется щелчком мыши на закладке окна. Состояние объектов отображается в окнах просмотра, причем последнее изменение состояния отмечается красным цветом. Если надобность в контроле состояния объекта от- падает, имя объекта можно удалить из окна просмотра, выделив его курсором мыши и нажав клавишу **DEL**.

Кроме окна *Watch* имеется еще 4 вкладки в нижней части экрана (рис. 3.10). Вкладка *Autos* (Автоматические) показывает автоматиче- ские переменные, используемые в текущем операторе. Вкладка *Locals* (Локальные) показывает значения переменных, локальных по отно- шению к текущей функции. Вкладка *Threads* (Потоки) позволяет про- сматривать и управлять потомками в многопоточных приложениях. Вкладка *Modules* (Модули) перечисляет детали модулей кода, выпол- няемых в данный момент.

Завершение сеанса отладки выполняется командой **Shift+F5** или **Debug/Stop Debugging**.

3.8. Создание исполняемого файла без отладочной информации

Для запуска приложения в данном режиме надо выполнить ко- манду **Debug/Start Without Debugging**. При запуске приложения по- следовательно реализуются два процесса:

– *построение решения*. Исходный код из файла транслируется в промежуточный код. Промежуточный код приложения сохраняется в файле `*.exe`;

– *выполнение решения*. Автоматически запускается среда испол- нения, в которой выполняется код, содержащийся в файле `*.exe`.

Если в исходном коде есть ошибки, решение не будет построено. Синтаксические ошибки будут выявлены в процессе трансляции ис- ходного файла `*.cpp` в промежуточный код. Сообщение о наличии ошибок выводится в окне, вид которого приведен на рис. 3.11.

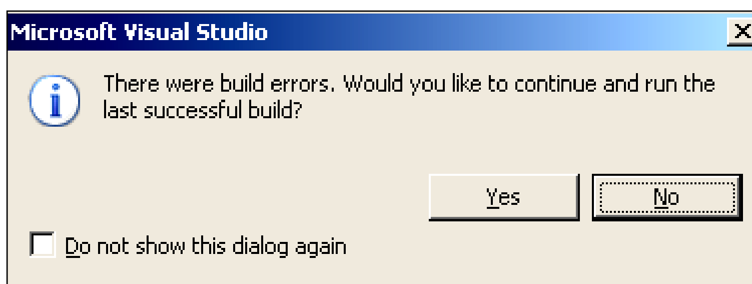

Рис. 3.11. Окно сообщений об ошибках

Надо отказаться от построения решения, щелкнув по кнопке **No**,
и исправить ошибки.

До этого построение приложения осуществлялось в отладочной конфигурации (*Win32 Debug*) с включением в файл всей необходимой отладочной информации, что позволило в дальнейшем использовать возможности отладки (рис. 3.12). Эта информация сохраняется в фай- ле `*.pdb`. После отладки и исправления всех ошибок осуществляется построение приложения в рабочей конфигурации (*Release*). Рабочая конфигурация не содержит какой-либо отладочной информации и ис- пользует заданную оптимизацию кода. Процесс оптимизации может изменять последовательность выполнения кода, чтобы сделать его бо- лее эффективным, или даже вовсе исключать излишний код. Как ре- зультат, размер исполняемого файла существенно уменьшается. По- скольку это разрушает отображение «один к одному» между исход- ным кодом и соответствующими блоками исполняемого машинного

кода, оптимизация может затруднить или запутать пошаговое выпол- нение программы. Для переключения в окончательную конфигурацию необходимо выбрать команду **Build/Configuration Manager** из меню. На экран будет выведено диалоговое окно установки активной конфи- гурации проекта. Выберается опция *Win32 Release*. Повторяется по- строение выполнением команды **Build/Rebuild All**. Каждая конфигу- рация проекта определяет также папки, куда будут помещены файлы с промежуточными и окончательными результатами компиляции и компоновки. По умолчанию это папки *Debug* и *Release*, которые рас- полагаются в папке проекта.

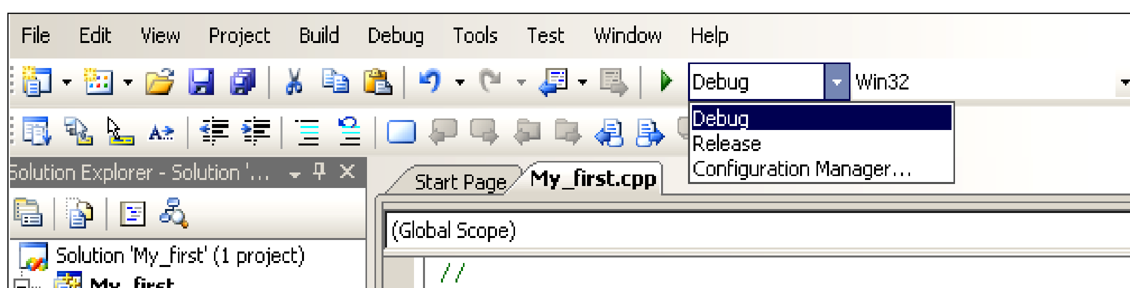

Рис. 3.12. Выбор отладочной конфигурации

Коротко рассмотрим назначение папок и файлов приложения. Папка решения содержит: файл текущего примера решения `*.sln`, файл с информацией о проектах решения и опциями решения `*.suo`, данные `Intellisense` для решения и вложенную паку с названием решения. В файле решения зафиксирован перечень проектов, входя- щих в решение. Во вложенной папке находятся файлы и папки про- екта: файл `My_first.vcproj` в формате XML содержит перечень файлов, включенных в проект; файл `My_first.cpp` содержит исход- ный код программы на языке С++; `*.obj` – объектные файлы, со- держащие машинный код исходных файлов проекта; `*.pch` – пред- варительно скомпилированный файл заголовков; `*.pdb` – файл с от- ладочной информацией, используемой при выполнении программы в режиме отладки; `*.idb` – файл с информацией, необходимой для пе- рестройки всего решения и др. Папка *Debug* используется для хране- ния временных файлов. В этой папке размещаются файлы с про- граммным кодом на промежуточном языке. В частности, файл `My_first.exe` содержит программный код, который реализует функ- циональность приложения.

В прил. 2 содержится описание команд меню интегрированной
среды разработки VS 2008