Часть 2. Разработка программ на C++ в среде Visual Studio

Лабораторная работа.5. MS Visual Studio, создание приложения с диалоговым меню и пользовательскими структурами данных

Цель выполнения лабораторной работы

Освоение работы в среде MS Visual Studio 2005, создание консольного приложения с диалоговым меню и пользовательскими структурами данных. Освоение возможностей MS Visual Studio по отладке и контролю утечек памяти.

Порядок выполнения работы

Обучаемые разрабатывают представленные в описании демонстрационные программы в среде MS Visual Studio 2005.

Получают индивидуальное задание для построения своего проекта.

Обучаемые разрабатывают программы в соответствии с представленным описанием по выданным им индивидуальным заданиям.

Среда разработки Visual Studio 2005. Hello new word!

Первые версии Microsoft Visual Studio появились еще в 90-х года прошлого века. Среды разработки 5.0 и 6.0 уже и мели привычный оконный интерфейс. Новая платформа разработки Microsoft Visual Studio .Net, пришедшая на смену предыдущим версиям в начале 2000-х, концептуально отличается наличием специальной платформы исполнения .Net Framework управляемого кода, но также поддерживает возможность написания классических приложений на C++ (для которых не требуется наличие специальной среды исполнения).

В Текущем курсе лабораторных работ будет использована v.8 Microsoft Visual Studio .Net выпущенная в 2005 году.

После ее запуска на экране появляется многооконное приложение Windows, фрагмент которого приведен на Рис. 5-1. Его главное меню содержит наименования – File, Edit, View, Tools, Test, Windows, Community, Help. Основной набор команд который на понадобиться в нашем курсе очень узок – это основные команды для разработки консольных приложений Windows.

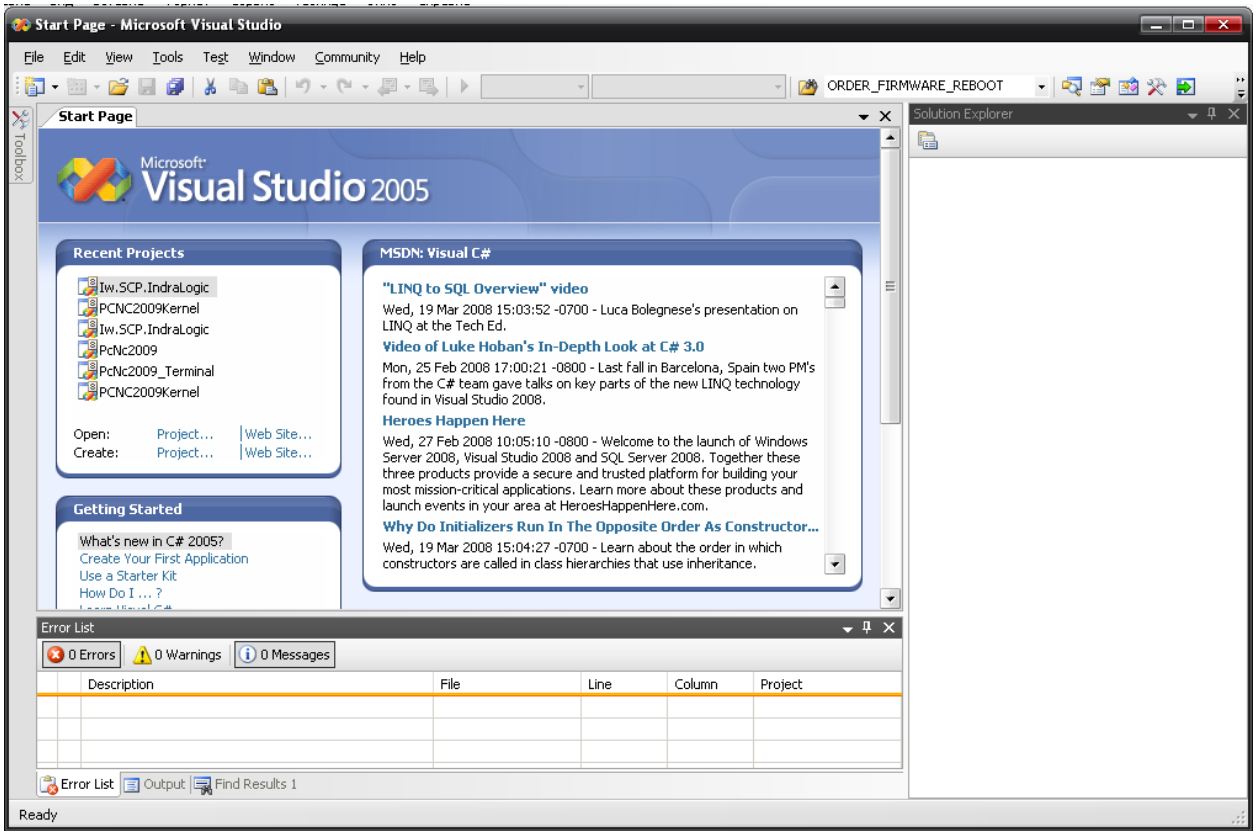

Рис. 5-1 Стартовое окно среды разработки MS Visual Studio 2005

Консольное приложение Windows внешне очень напоминает приложение в стиле MS-DOS. Монитор работает в режиме, похожем на текстовый режим DOS-приложений, окно консольного приложения может быть распахнуто на весь экран нажатием комбинации Alt+Enter. Однако сняты все прежние ограничения на ресурсы по оперативной памяти – задача может использовать максимальный объем, предоставляемый операционной системой Windows, массивы могут иметь достаточно большие размеры, обусловленные 32-разрядной адресацией памяти. Кроме того, для данных типа int выделяется по 4 байта, что расширяет диапазон представления таких значений по модулю до 2^{31}-1

В отличие от консольных приложений стандартные приложения Windows могут быть организованы как однооконные или многооконные приложения, использующие типовой интерфейс в виде различных кнопок, обычных и всплывающих меню, различного рода диалоговых окон, списков, линеек прокрутки и других компонент, упрощающих реализацию типовых процедур, устоявшихся в системах программирования.

Для создания заготовки проекта консольного приложения в среде Ms VisualStudio необходимо выполнить команду File->New->Project в главном меню и в появившемся диалоговом окне () сделать следующее:

1. выбрать Visual C++->Win32 Console Application,
2. Ввести название проекта и выбрать его размещение
3. и нажать кнопку OK.

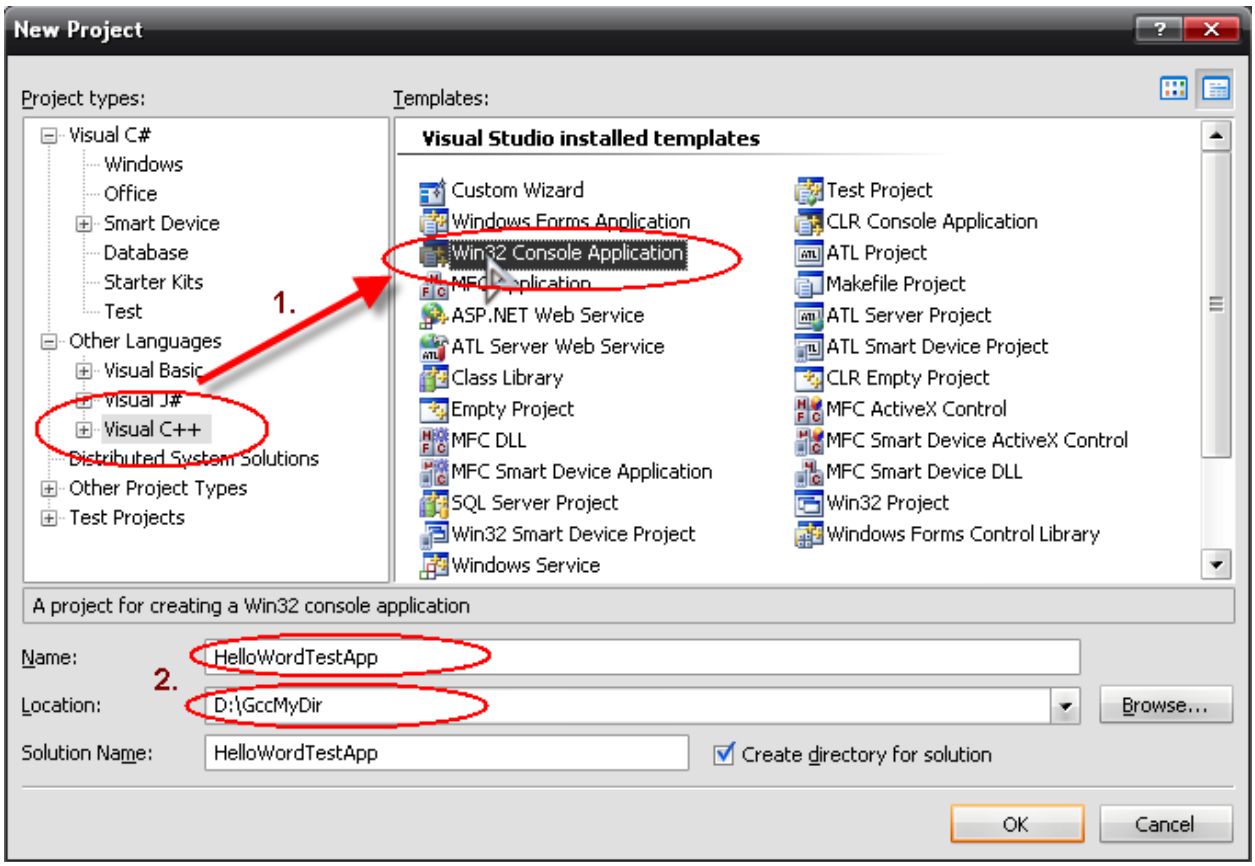

Рис. 5-2 Создание проекта для консольного приложения

После этого проявится диалоговое окно визарда (мастера создания) проекта в котором ничего не надо менять, можно лиш ознакомиться с предлагаемыми настройками проекта и нажать клавишу “Finish” (Рис. 5-3).

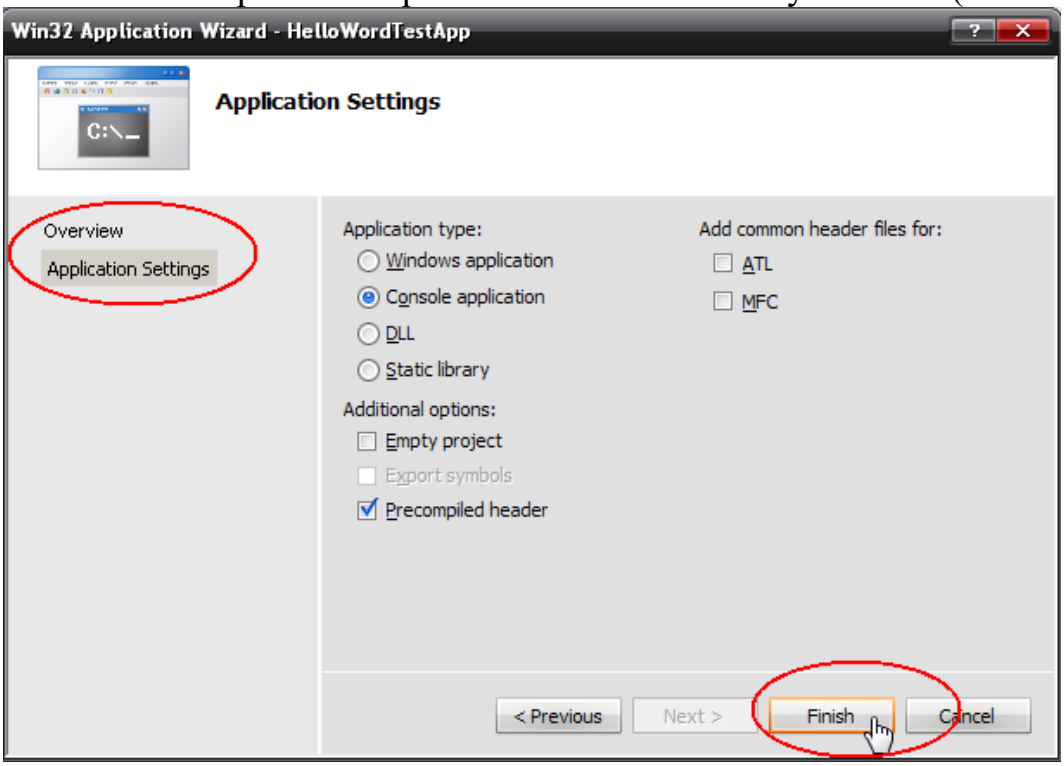

Рис. 5-3 Мастер создания проекта для консольного приложения

В результате выполнения в среде VisualStudio откроется созданный проект (Рис. 5-4):

1. В области 1. вы можете видеть главную функцию main вашего консольного приложения.

2. В области 2. вы можете видеть файловую структуру созданного проекта.

3. В структуре проекта размещаются *.cpp и *.h файлы вашей программы. В частности стандартный генерируемый заголовочный файл stdafx.h.

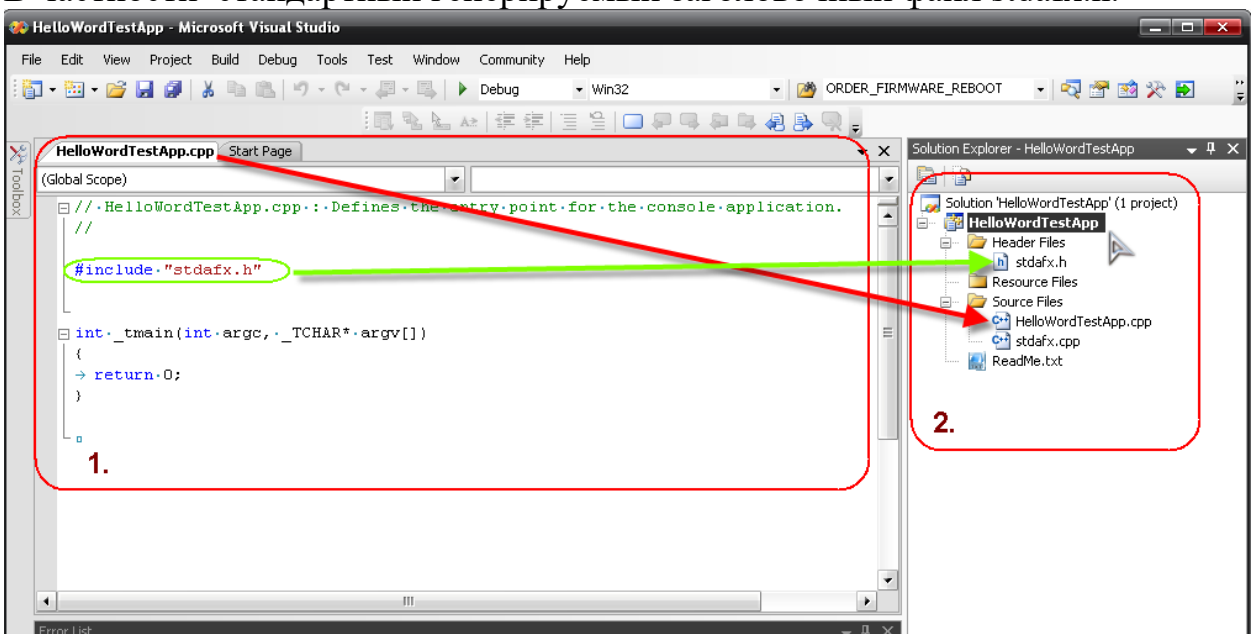

Рис. 5-4 Файл консольного приложения с функцией _tmain

Открыв заголовочный файл stdafx.h, ознакомимся с его содержанием (Рис. 5-5). Здесь используется инструкция препроцессора “pragma once” для подключения текущего файла только один раз и производится подключение заголовочных файлов библиотек работы с вводом/выводом и работы с расширенными символами (для использования типа wchar_t).

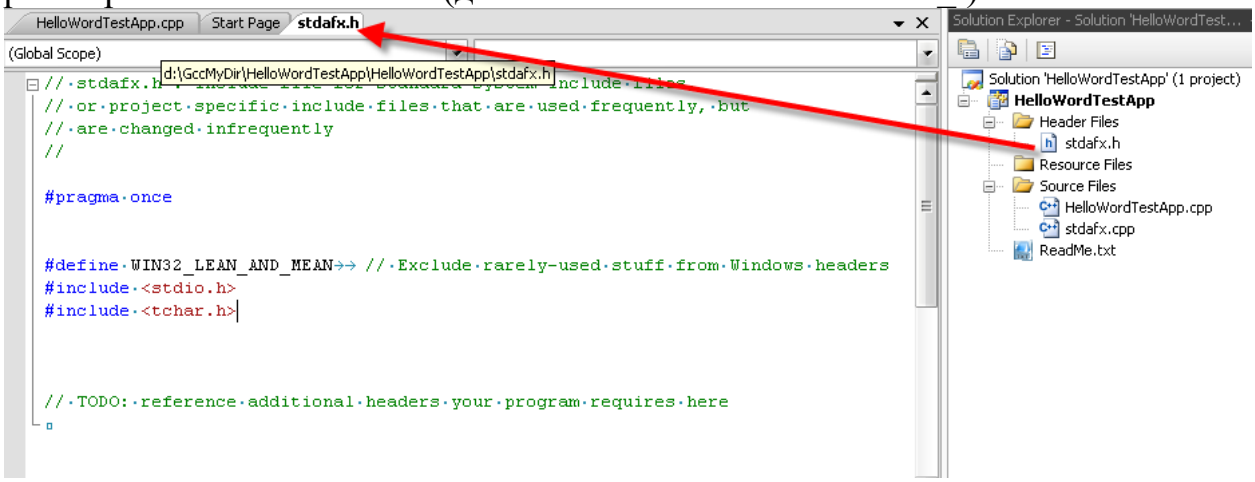

Рис. 5-5 Файл для подключения заголовочных файлов

Для получения полной структуры файлов и папок скомпилируйте шаблон приложения командой Build->Build Solution (F6).

Структура файлов и папок проекта полученного консольного приложения представлена на следующем рисунке (Рис. 5-6). Файл расширения *.sln (файл сборки) объединяет набор проектов организую рабочее пространство для создания много модульных приложений. В нашем случае рабочее пространство сборки включает только одно тестовое приложение HelloWordTestApp, папка которого размещается вместе с файлом сборки. Здесь же располагается папка debug в которой будут размещены файлы скомпилированного приложения.

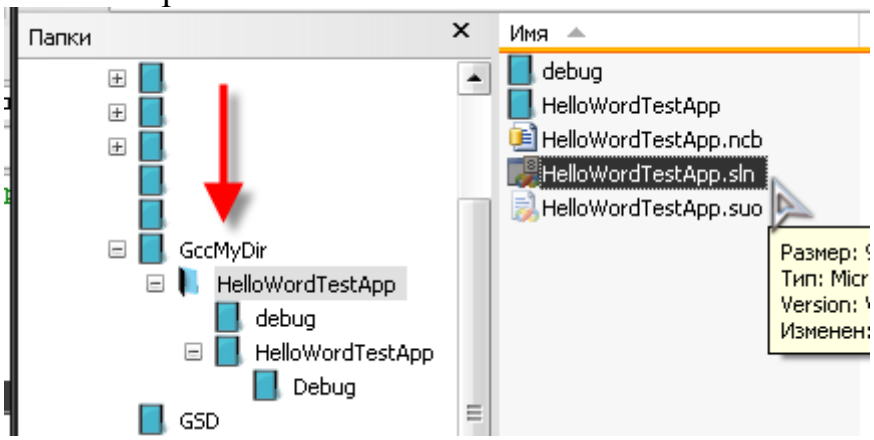

Рис. 5-6 Файл сборки проектов *.sln

Папка самого проекта (GccMyDir\HelloWordTestApp\HelloWordTestApp) содержит все файлы исходных кодов используемых в проекте плюс файл проекта (HelloWordTestApp.vcproj) и файл настройки текущих настроек пользователя Windows (HelloWordTestApp.vcproj.MICROSOF-730FBB.Администратор.user). Внутренняя папка Debug содержит временные файлы компиляции.

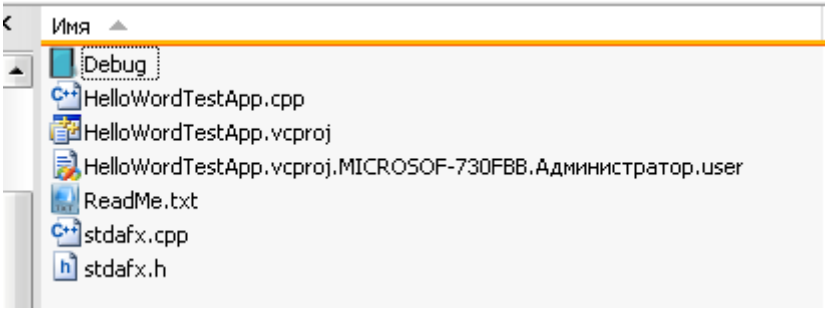

Рис. 5-7 Содержимое папки проекта

Рассмотрим содержимое папки HelloWordTestApp\debug. Здесь выделим 2 основных файла, относящихся к нашему приложению:

HelloWordTestApp.exe – исполняемый файл приложения;

HelloWordTestApp.pdb – файл отладочной информации, который необходим при пошаговой отладке приложения.

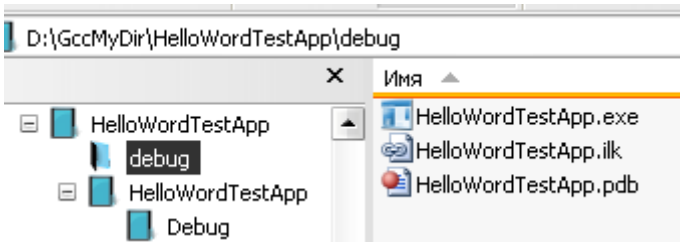

Рис. 5-8 Папка debug

Вернемся в среду VisualStudio и изменим код программы для традиционного Hello Word приложения.

```
  #include "stdafx.h"
  #include "conio.h"

  int _tmain(int argc, _TCHAR* argv[])
  {
    int i = 2005;
    printf("Hello new word of Visual Studio %d! \n", i);
    int j, k = 100, someThing; // объявление переменных не в начале
программы!
    printf("Define variable in any place of programm. k = %d. \n", k);

    getch();
    return 0;
  }
```

Снова скомпилируем (Build->Build Solution (F6)) код приложения и запустим его на исполнение командой Debug->Start Debugging (F5). Для выполнения консольного приложения будет открыто окно консоли. Результат выполнения программы очевиден.

Вывод результатов компиляции и диагностики приложения

Результаты процесса компиляции и диагностические сообщения процессы выполнения отображаются в окне инструментария Output (открытие командой View->Output (Ctrl+W, O))(Рис. 5-9)

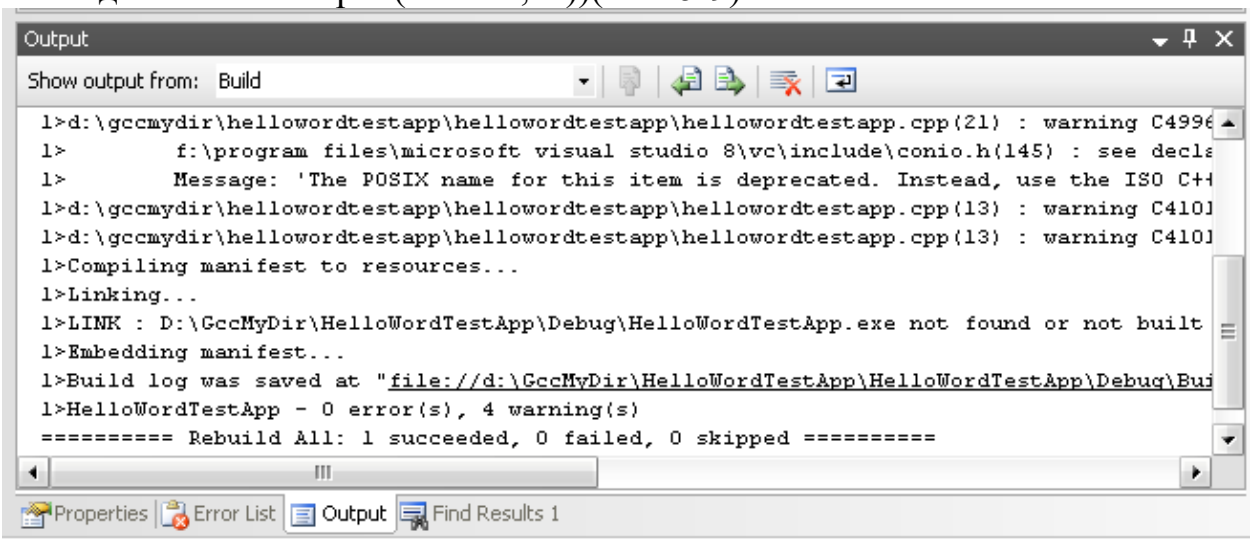

Рис. 5-9 Окно с результатами построения проекта

Так же ошибки и предупреждения процесса компиляции выводятся в более структурированном виде в окне Error List (открытие командой Ctrl+W, E) (). Двойным нажатием левой кнопки мышки на строчке проблемы курсор переносится в редактор кода в позицию ошибки.

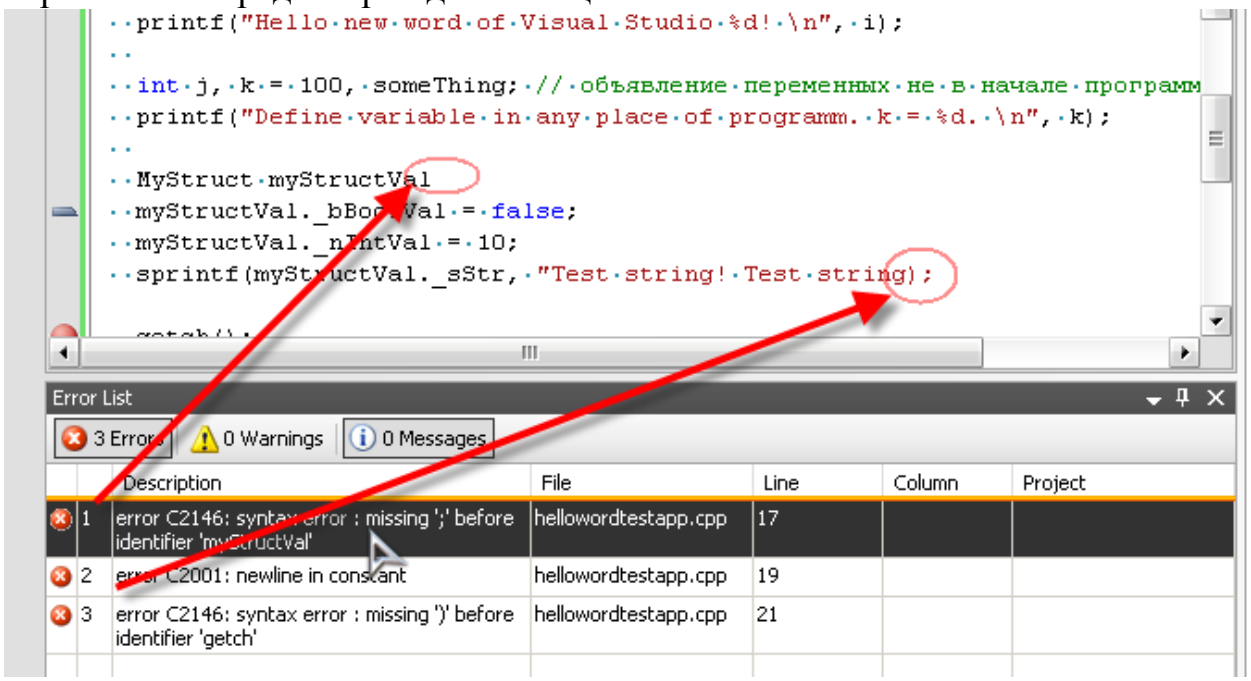

Рис. 5-10 Окно сообщений об ошибках

Например в представленном на рисунке выше примере в строке 16 пропущен символ “;” и закрывающие двойные кавычки в аргументе функции sprintf().

Инструментарий отладки Visual Studio

Теперь воспользуемся самым необходимым инструментом VisualStudio – режимом отладки.

В принципе команда Debug->Start Debugging (F5) уже запускает приложение под контролем системы отладки. Для примера после запуска приложения под отладчиком вернитесь в окно среды разработки и нажмите на панели инструментов отладки на паузу (Рис. 5-11).

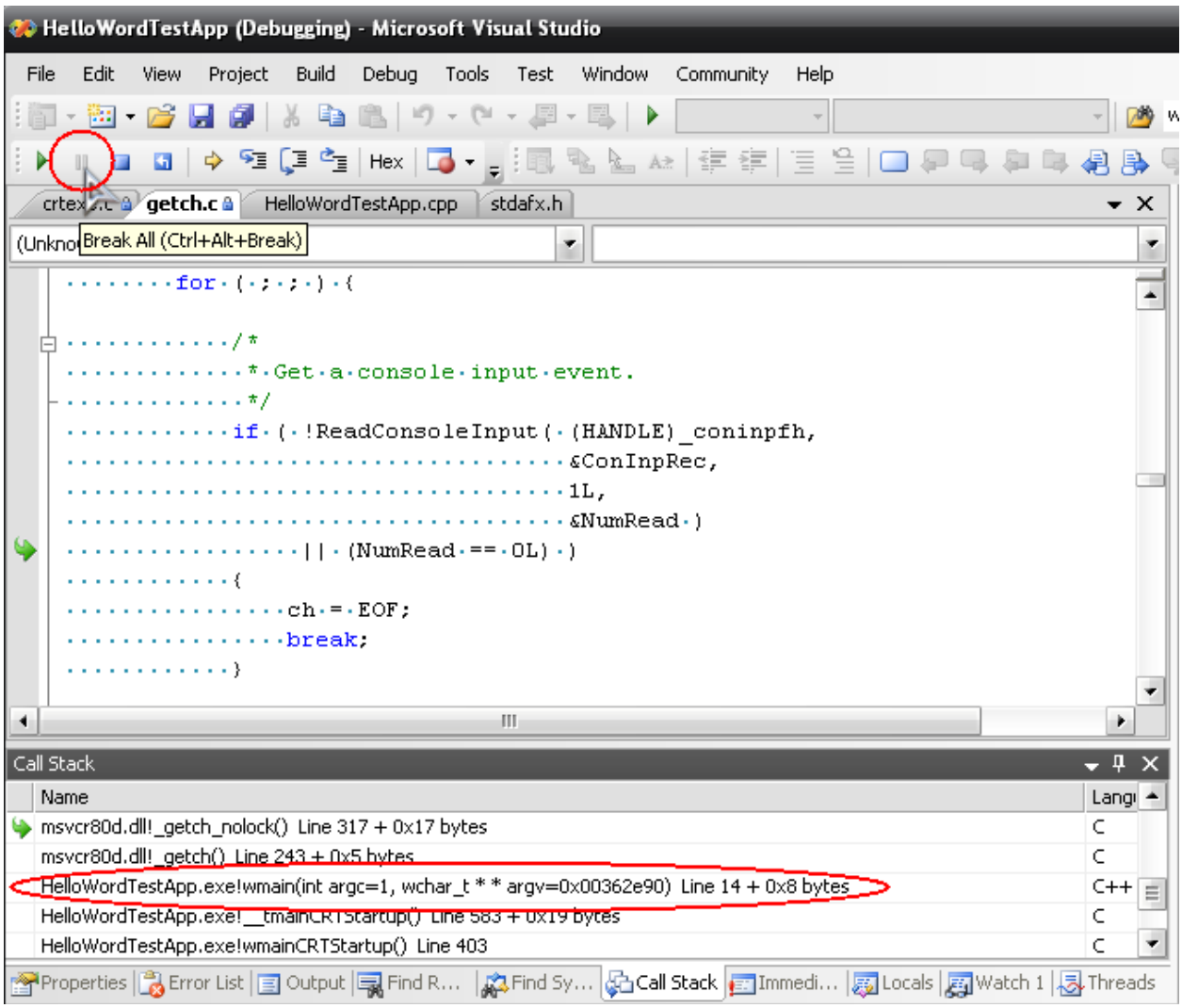

Рис. 5-11 Окно стека вызовов в ходе отладки приложения

В результате программа остановиться в процессе выполнения функции getch() в файле getch.c. Здесь вы можете видеть бесконечный цикл, в котором производится чтение событий ввода(ReadConsoleInput). В окне инструментария CallStak (открыть с помощью Debug->Windows->CallStack (Ctrl+D, C)) можно увидеть стек вызовов, в котором есть и наша функция main. Выполнив двойной клик на нужной строчке можно перейти к этому файлу кода (Рис. 5-12).

Как представлено на рисунке, зеленая стрелка показывает место курсора выполнения программы в текущем файле.

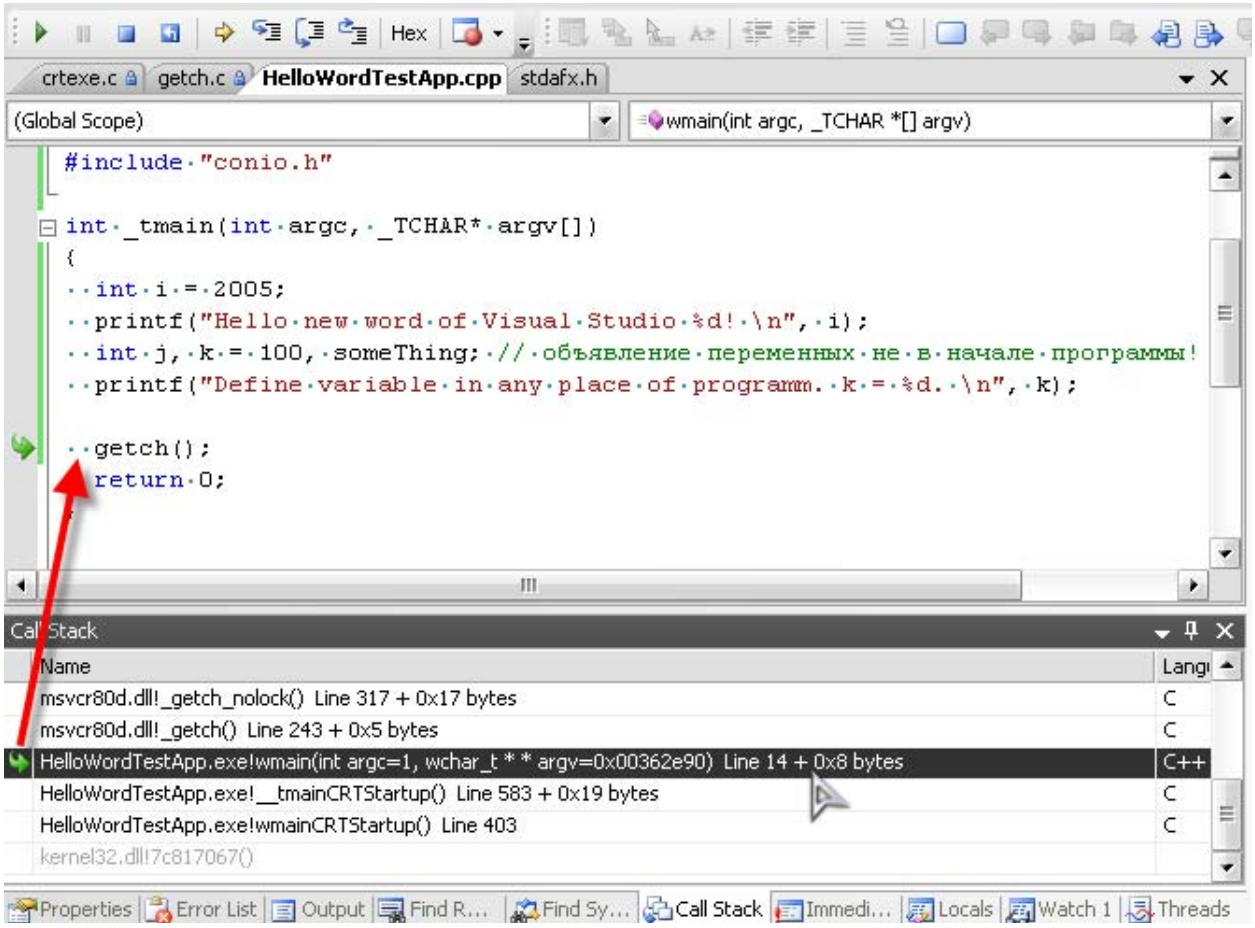

Рис. 5-12 Навигация по стеку вызовов

Нажмите на панели инструментов кнопки Continue (F5) или Stop Debugging (Shift+F5) для продолжения или завершения отладки.

Использование точек останова

Для установки точки прерывания программы (Breakpoint) установите курсор редактирования на нужную линию и нажмите F9 (или вызвать команду Debug->Toggle Breakpoint). Также можно просто кликнуть на левом поле левой кнопкой мыши. Так или иначе, в результате этой операции на поле должна появиться красная точка, означающая точку останова по положению (Рис. 5-13).

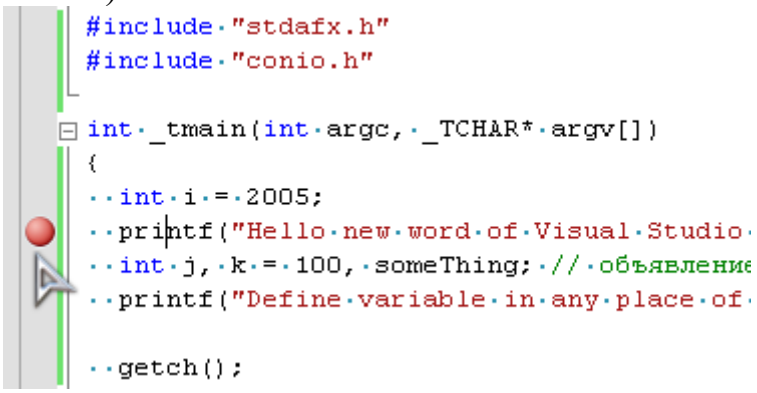

Рис. 5-13 Точка останова

Теперь при выполнении программа остановиться в этой точке (Рис. 5-14).

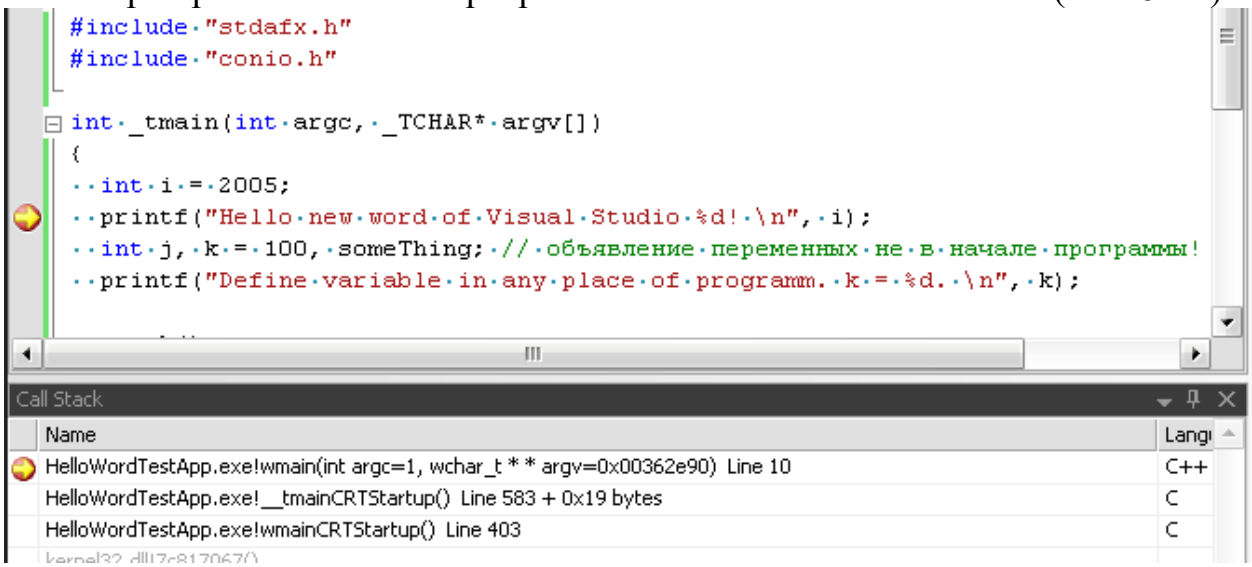

Рис. 5-14 Срабатывание точки останова

В этом режиме возможна пошаговая отладка приложения с помощью команд панели отладки.

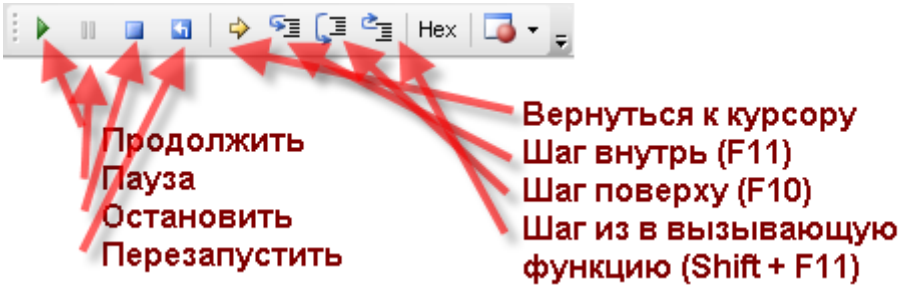

Рис. 5-15 Функции панели отладки

Например, при нажатии F11 (Шаг внутрь) выполнение программы перейдет внутрь функции printf. Для того чтоб вернуться обратно в функцию main нажмите Shift + F11 (Шаг из …).

Контролируя ход выполнения программы можно смотреть, какой вывод осуществляется в окно консоли.

Контроль значений переменных при пошаговом выполнении

Добавим в наше приложение пользовательский тип данных и добавим несколько строк в функцию main для использования этого типа данных:

Добавим в проект новый заголовочный файл myData.h. для этого в окне файлов проекта Solution Explorer вызовем команду контекстного меню Add->New Item… (Рис. 5-16). В открывшемся диалоге (Рис. 5-17) выберем в категории добавляемых элементов слева “Code” и “Header File” в заготовках файлов. Введем имя нового файла в поле Name.

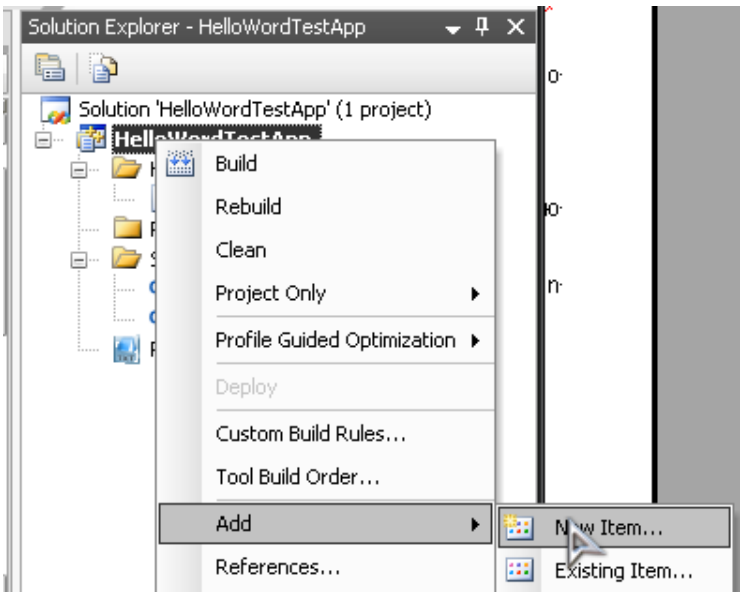

Рис. 5-16 Добавления нового элемента в проект

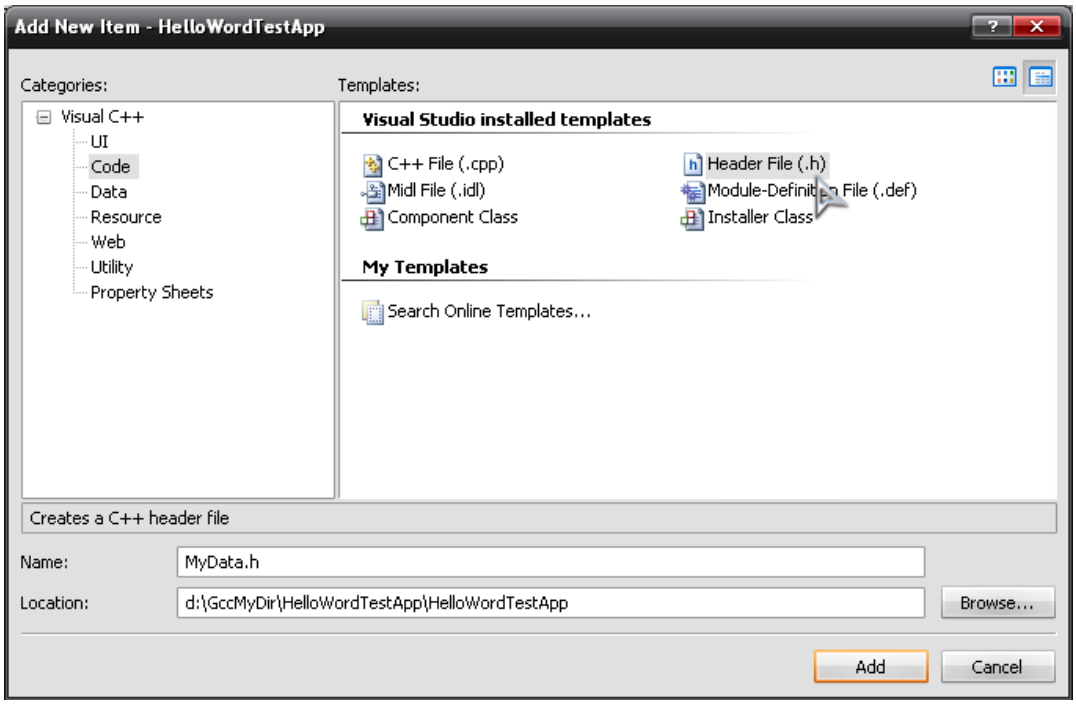

Рис. 5-17 Добавление заголовочного файла

В результате новый пустой файл будет добавлен в структуру файлов проекта. Аналогично возможно добавлять файлы кода (*.cpp) и даже новые классы с применением соседней контекстной команды Add->Class.

В созданном заголовочном файле, используя директивы препроцессора для предотвращения повторного включения кода, объявим нашу структуру данных:

```
#ifndef MYDATA_STRUCT
#define MYDATA_STRUCT

typedef struct _myStruct
{
```

```
    bool _bBoolVal;
    int _nIntVal;
    char _sStr[30];
  } MyStruct;

  #endif
```

Подключим заголовочный файл нашей структуры в файле функции main и добавим в код этой функции создание и инициализацию полей переменной нашего нового типа данных:

```
  ...
  #include "MyData.h"

  int _tmain(int argc, _TCHAR* argv[])
  {
    int i = 2005;
    printf("Hello new word of Visual Studio %d! \n", i);

    int j, k = 100, someThing; // объявление переменных не в начале
программы!
    printf("Define variable in any place of programm. k = %d. \n", k);

    MyStruct myStructVal;
    myStructVal._bBoolVal = false;
    myStructVal._nIntVal = 10;
    sprintf(myStructVal._sStr, "Test string! Test string");

    getch();
      return 0;
  }
```

Поставим точку останова в вызове функции getch(), скомпилируем (F6) и запустим программу на выполнение (F5).

Если навести курсор на переменную myStructVal нашего типа, появится всплывающая подсказка, которая будет содержать значение переменной и её внутренних полей (Рис. 5-18). При этом внутренняя строка объявленная как массив символов может быть полностью отображена.

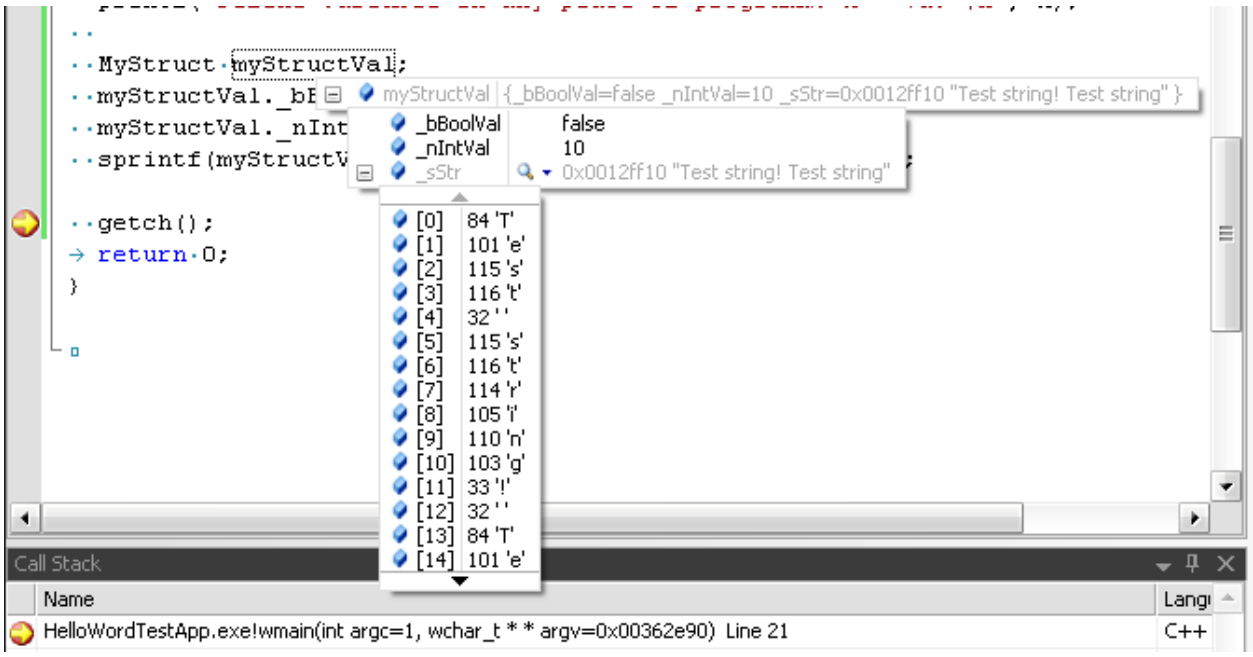

Рис. 5-18 Просмотр данных во время отладки

Также для наблюдения за значениями выбранных переменных используется окно Watch (открытие по команде Debug->Windows->Watch->Watch1 (Ctrl + D, W)). Для добавления новой переменной в окно наблюдение вызовите контекстную команду Add Watch, наведя курсор на имя переменной (Рис. 5-19):

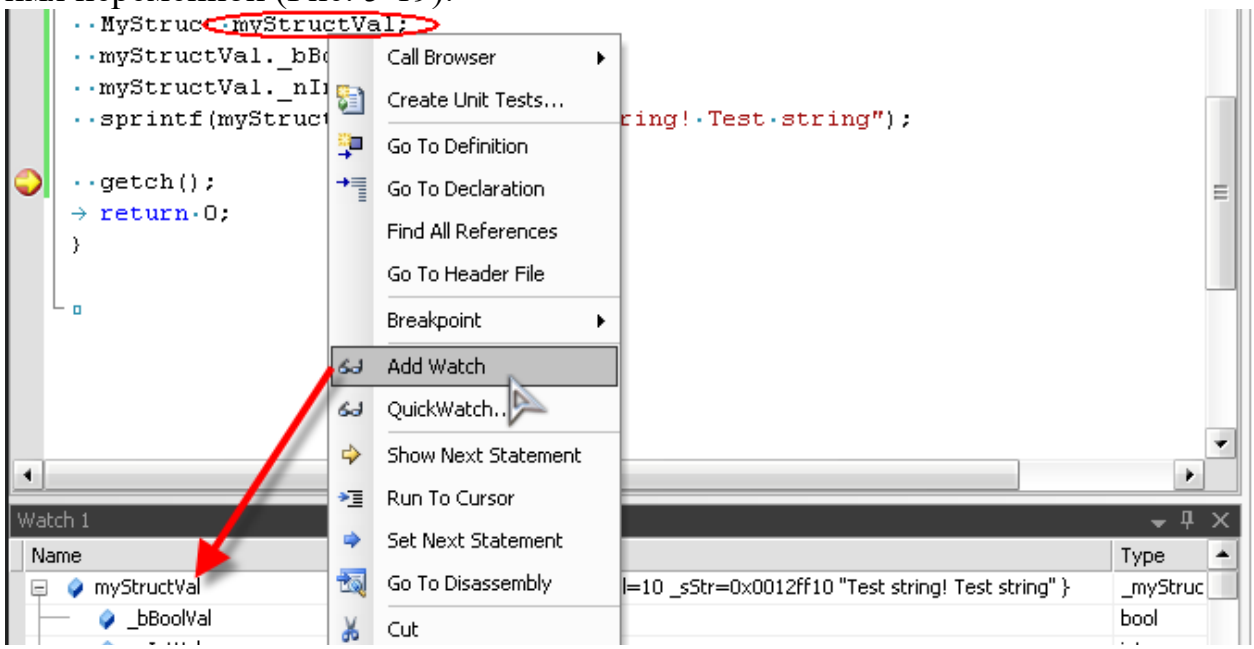

Рис. 5-19 Добавление переменной в окно наблюдения

Ознакомившись с базовыми операциями создания и отладки программы в MS Visual Studio 2005, перейдем к основному содержанию лабораторной работы.

Использование структур данных пользователя

Для ввода в программу пользовательских структур данных используются ключевые слова typedef и struct. Ключевое слово typedef используется для определения более лаконичного (короткого) названия спецификатора имени структуры.

Например, структурный тип определяющий данные товара на складе, возможно задать следующим образом:

```
struct tovar
{
  char* name;    // название
  long price;    // оптовая цена
  float percent;// процент наценки
  int volume;    // объем партии
  char date[9]; // дата поставки
};
```

Объявление переменной объявленного таким образом типа будет следующим:

```
struct tovar igrushka_myagkaya
```

Для более лаконичной записи имени структуры, используем операцию переопределения типов typedef:

```
typedef struct _tovar
{
  char* name;    // название
  long price;    // оптовая цена
  float percent;// процент наценки
  int volume;    // объем партии
  char date[9]; // дата поставки
} Tovar;
```

Теперь переменная данного пользовательского типа данных объявляется следующим образом:

```
Tovar igrushka_myagkaya;
```

Построение консольного приложения с диалоговым меню

Приложение обрабатывающее некоторые запросы пользователя должно «уметь» продолжать работу, пока оно необходимо пользователю. Т.е. пользователь должен иметь возможность управлять ходом выполнения приложения для получения нужных ему результатов.

Простейшим способом организации диалога с пользователем для консольных приложений является повторяющееся меню выбора нужных пользователю операций.

В качестве сквозного примера в данной лабораторной, рассмотрим приложения для расчета площади набора геометрических фигур, таких как круг, прямоугольник и треугольник.

В начале работы с приложением пользователю предоставляется следующее меню:

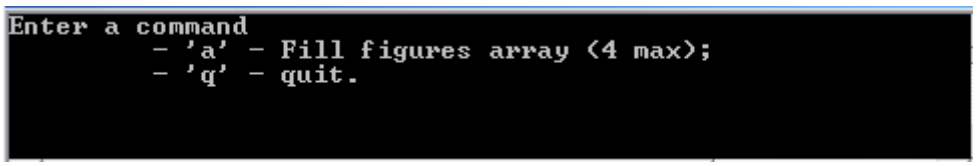

Рис. 5-20 Начальное меню приложения

После выполнения команды “a” и заполнения необходимого набора фигур из 4-х элементов приложение снова выдает меню выбора но уже с расширенным набором команд ‘s’ и ‘p’ для обработки введеных фигур:

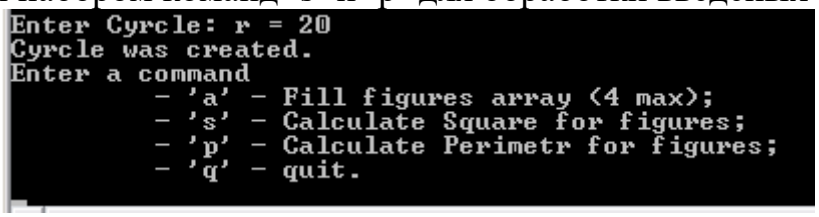

Рис. 5-21 Меню приложения после ввода фигур

Каждая команда показывает результат расчетов произведенных для введенных фигур и возвращает пользователю возможность выбора новых команд:

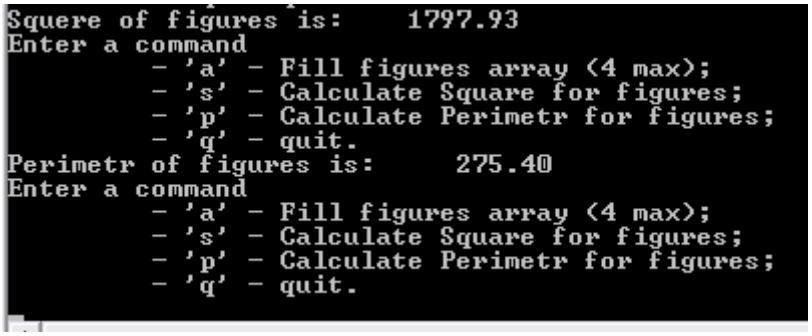

Рис. 5-22 Отработка команд в приложении

Завершение (закрытие) приложения осуществляется при вводе команды ‘q’.

Рассмотрим «каркас приложения» реализующие представленное меню:

```
int _tmain(int argc, _TCHAR* argv[])
{
  // флаг завершения программы
  bool finish = false;
  int realCount = 0;
  while(!finish)//выполнение циклического вывода меню с условием выхода
  {
    printf("Enter a command \n");
    printf("\t - \'a\' - Fill figures array (4 max); \n");
    if(realCount > 0) // условие отображения дополнительных команд
    {
      printf("\t - \'s\' - Calculate Square for figures; \n");
      printf("\t - \'p\' - Calculate Perimeter for figures; \n");
    }
    printf("\t - \'q\' - quit. \n");
    int key = getch();
```

```
      if(key == 'q' || key == 'Q') // проверка необходимости завершения
      {
        finish = true; // установка флага завершения
        continue;
      }
      // оператор выбора для введенных команд
      switch (key)
      {
        case 'a':
        case 'A':
        /// ...
          break;
        default:
          printf("Bad command. Try again or Quit.");
      }
    }
    return 0;
  }
```

В представленном коде цикл while выполняется до тех пор, пока переменная finish имеет значение false. В теле цикла выполняется следующие этапы программы:

Вывод доступных команд меню (с условным выводом доступных команд)

Считывания символа с использованием функции getch

Проверка условия выхода (key == 'q')

Выполнение заданной пользователем команды с применением оператора switch.

В соответствии с введенным пользователем символом, код которого записывается в переменной key, выполняется соответствующая ветка в операторе switch. Например, в case ‘a’ должен выполняться ввод геометрических фигур.

После ввода кода приложения необходимо его скомпилировать, что позволит своевременно обнаружить синтаксические ошибки ввода на данном этапе разработки приложения.

Для выбранного индивидуального задания реализуйте главное меню приложения.

Индивидуальные задания на лабораторную работу

По представленным темам индивидуальных заданий необходимо реализовать программу. Структура программы и ее функции должны быть сформированы по аналогии с представленным примером программы расчета площади и периметра двухмерных геометрических фигур. В реализации задачи должны быть использованы 2 различных типа структур данных (с различным набором полей данных), размещаемых в общий массив элементов для последующей обработки.

Расчет 3D фигур: куба, цилиндра, призмы. Расчет общего объема и общей площади поверхностей.

Расчет 3D фигур: пирамида с квадратным основанием, пирамида треугольным основанием, конус. Расчет общего объема и общей площади поверхностей.

Расчет многоугольников: (пятиугольник, шестиугольник, восьмиугольника) и овалов. Расчет площади и общего периметра.

Расчет стоимости маршрутов: типы - город, трасса, грунт; данные - путь, тип, расход бензина, средняя скорость. Расчет стоимости и времени движения.

Расчет электрической мощности электроприборов, нагрузка на розетку по току.

Программа расчета цены продуктов в корзине цена=вес*(стоимость кг)*(количество продукта).

Программа грубого расчета дальности выстрела от заданного угла и боеприпаса по баллистической траектории.

Программа решения набора квадратных уравнения (ax2+bx+c=0).

Программа расчета давления в баллоне от температуры (формула из интернета, разные баллоны на разные предельные давления, разные газы, разные объемы и температуры).

Программа расчета калорийности продуктов (вес*калорийность и сложение ингредиентов).

Программа расчета рейтингов для группы магистров кафедры (5-6 чел) (5 предметов, рейтинг студента, средний для группы).

Программа расчета тех обслуживания парка авто в месяц (масло, фильтры, балансировка, мойка, расход средний, общий).

Расчет площади для парковки авто (тип авто, площадь, расчет средней общей).

Программа расчета потребляемой электрической энергии за сутки в цехе (станки, мощность, часы работы).

Мощность эл. микро элементов в игрушке (свето-диоды, лампочки, эл. моторы (тип мотора), ток, мощность, время работы игрушки. количество батареек).

Расчет листов и стоимости работ при печати авторефератов/брошюр (типы печати А4, А5, А3, параметры - цвет, переплет, кол-во, обложка).

Реализация пользовательских структур данных и само-идентификацией типа в программе

В рамках программы или пользовательской библиотеки для набора однотипных объектов можно задать поле, предназначенное для идентификации типа конкретного объекта. Введем перечисления для наглядного обозначения типа объектов.

```
// Перечисление типов фигур программы
typedef enum _figures
{
  FCyrcle = 1,
  FRectangle,
  FTriangle
} Figures;
```

Далее введем набор пользовательских типов данных, которые описывают геометрические фигуры:

```
// Прямоугольник
typedef struct _rect
{
  Figures kind;
  int len; // длинна
  int width; // ширина
  int x, y;
}Rect;

// Круг
typedef struct _cyrcle
{
  Figures kind;
  int Rad; // радиус
  int x;
  int y;
}Cyrcle;
```

```
// Треугольник
typedef struct _triangle
{
  Figures kind;
  int side_a; // сторона a
  int side_b; // сторона b
  int side_c; // сторона c
}Triangle;
```

После ввода кода приложения необходимо его скомпилировать, что позволит своевременно обнаружить синтаксические ошибки ввода на данном этапе разработки приложения.

Как представлено, во всех структурах первое поле – это идентификатор типа, который может быть использован для операций приведения типа. Все объекты представленных типов можно создать динамически и разместить указатели на объекты разных типов в один массив указателей void*. Например, в один запуск программы пользователю нужно произвести вычисления для 2-х треугольников и 2-х окружностей, а в другой раз для 3-х прямоугольников и 1-го треугольника. Т.е. каждый раз будет новый набор объектов различных типов данных.

Для размещения объектов различного типа (создаваемых динамически) используем массив указателей типа void*:

```
// массив указателей на 4 фигуры, для указателей заданы значения NULL
void* ppFigArray[]={NULL,NULL,NULL,NULL};
```

При этом если указанный массив заполнить значениями указателей, которые ссылаются на объекты различных типов, то получить идентификатор каждого объекта можно получить следующим образом:

```
Figures figure_kind = *((Figures*)ppFigArray[0]);
```

Здесь указатель типа void* приводится к типу указателя на переменную типа (Figures*), а затем для этого указателя выполняется операция * (операция разъименования указателя). Таким образом получается значение первого поля kind структуры.

Представленную возможность получения значения kind из указателя на неизвестный объект показывает схема на рисунке ниже:

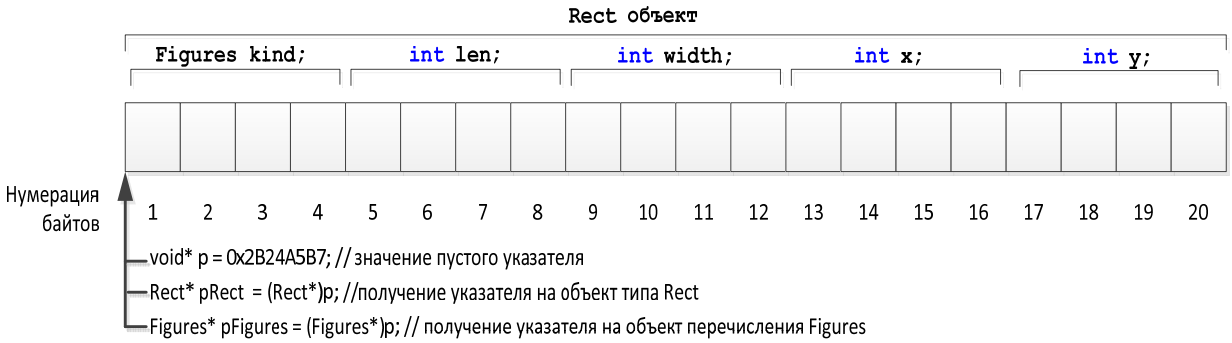

Рис. 5-23 Схема размещения данных структуры Rect

Объект структуры Rect занимает 20 байт. Первое поле структуры kind имеет тип Figurer и в текущей программе используется для идентификации типа объекта. Указатель на объект структуры имеет – это адрес. И этот адрес одинаковый для объекта структуры и объекта первого поля этой структуры - поля kind. Поэтому будут справедливый следующие приведения типов для указателей:

```
void* p = 0x2B24A5B7; // значение пустого указателя
Rect* pRect  = (Rect*)p; //получение указателя на объект типа Rect
Figures* pFigures = (Figures*)p; // получение указателя на объект
                                 //перечисления Figures
```

Аналогичным образом организованы поля в структуре Cyrcle и Triangle в программе – первое поле структуры это поле kind типа Figures.

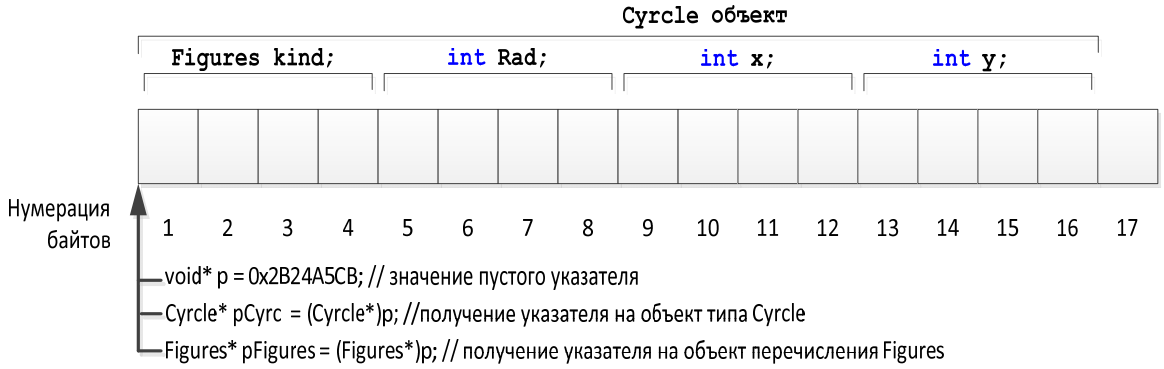

Рис. 5-24 Схема размещения данных структуры Cyrcle

Для выбранного индивидуального задания реализуйте структуры данных приложения.

Построение приложения с динамическими объектами

Использование динамически создаваемых объектов позволяет приложению распределять необходимый объем памяти во время работы, т.е. в ходе обработки запросов и операций пользователя или специализированной логики выполнения программы. Использование динамически создаваемых

объектов неразрывно связано с понятием указателя в языке C++. Указатель используется для хранения значения адреса памяти переменной (объекта) заданного типа. Например, в программе реализована структура для работы с данным окружности на координатной плоскости XOY:

```
typedef struct _cyrcle
{
  Figures kind; // ID фигуры
  int Rad;  // Радиус окружности
  int x;
  int y;
}Cyrcle;
```

Пример создания динамического объекта структуры с помощью метода malloc в специализированной функции - CreateCyrcle:

```
// методы для динамического создания фигуры окружности
// Возвращает созданный объект инициализированный пользователем
Cyrcle* CreateCyrcle()
{
  int r;
  printf("Enter Cyrcle: r = "); // запрос радиуса
  scanf("%d", &r);
  // проверка значения
  if(r < 0)
    return NULL;
  // создание динамического объекта
  Cyrcle* p = (Cyrcle*)malloc(sizeof(Cyrcle));
  // проверка что память выделена
  if(p != NULL)
    return NULL;
  printf("Cyrcle was created. \n");
  p->kind = 1; // заполнение полей динамического объекта
  p->Rad = r;
  return p;
}
```

В методе CreateCyrcle сначала у пользователя запрашиваются данные необходимые для создания объекта круга. Если введенные данные корректные - производится выделение памяти под объект типа Cyrcle. Поля динамически созданного объекта инициализируются полученными значениями. Как результат выполнения функция возвращает указатель на созданный в ней объект. Если по какой-либо причине объект не был создан возвращается значение NULL.

```
  Аналогичным образом реализуются функции для создания динамических объектов
прямоугольника (Rect) и треугольника (Triangle):
  // создание прямоугольника
  Rect* CreateRect()
  {
    int a,b;
    printf("Enter Rectangle (length width):");
    scanf("%d %d", &a, &b);

    if(a < 0 || b < 0)
      return NULL;
    printf("Rect was created. \n");
    Rect* p = (Rect*)malloc(sizeof(Rect));
    p->kind = FRectangle;
    p->len = a;
    p->width = b;
```

```
  return p;
}
// создание треугольника
Triangle* CreateTriangle()
{
  int a,b,c;
  printf("Enter Triangle (sides a b c):");
  scanf("%d %d %d", &a, &b, &c);
  if(a + b <= c || b + c <= a || c + a <= b)
  {
    printf("Summ of 2 sides should be less than other one! \n");
    return 0;
  }
  printf("Triangle was created. \n");
  // если все нормально то создаем объект
  Triangle* p = (Triangle*)malloc(sizeof(Triangle));
  p->kind = FTriangle;
  p->side_a = a;
  p->side_b = b;
  p->side_c = c;
  return p;
}
```

Для выбранного индивидуального задания реализуйте методы для динамического создания и инициализации объектов структур.

Реализация функции для заполнения массива объектов

При выполнении команды «‘a’ – Fill figures array» пользователь вводит параметры нужных ему геометрических объектов. Выполнение данной команды удобнее представить в виде специализированной функции, что упростит обзор кода приложения и также позволит, впоследствии, использовать эту функцию для аналогичных целей, например в другом приложении.

Функции для ввода геометрических фигур будет иметь следующий прототип:

```
// Метод для обобщенного ввода фигур
int AddFigures(void**);
```

В качестве параметра функция принимает указатель на массив указателей типа void*, что соответствует типу void** (указатель на указатель типа void*). В эту функцию будет передаваться созданный ранее массив указателей ppFigArray для его заполнения указателями на создаваемые динамические объекты геометрических фигур.

Схема на рисунке ниже демонстрирует размещение указателей типа void* в массиве указателей ppFigArray. Значение адреса первого элемента массива представляется указателем p, тип этого указателя – указатель на указатель void* (т.е. void**). Для инициализации значения первого элемента массива (тип элемента массива void*) можно использовать операцию индексации p[0]. Функция CreateCyrcle возвращает адресс размещения динамически созданного объекта Cyrcle.

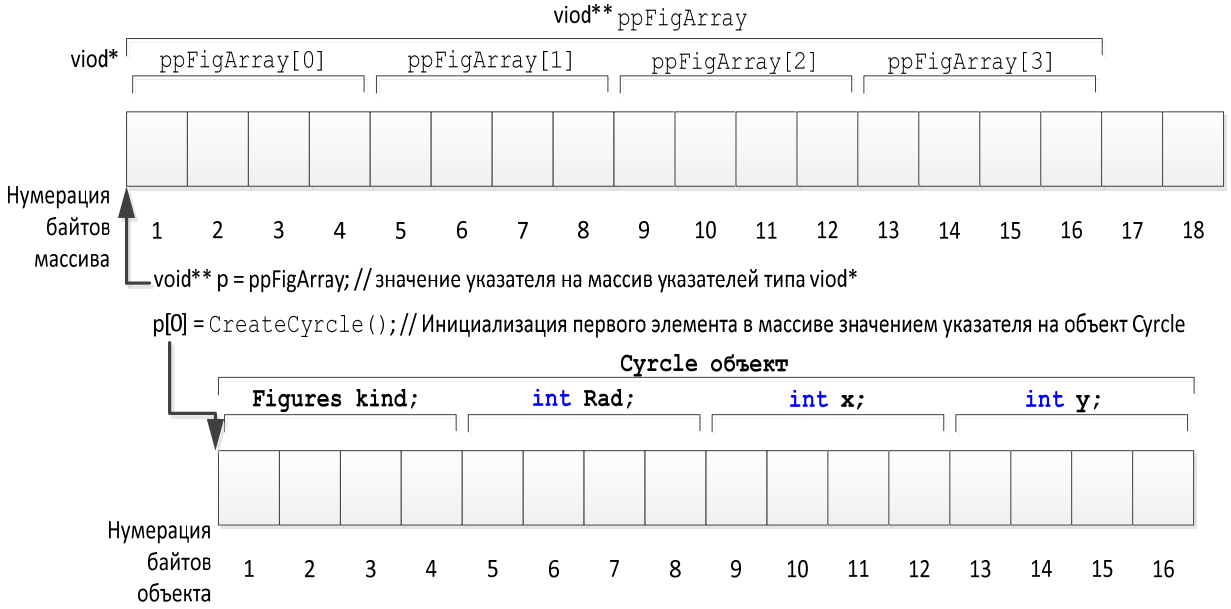

Рис. 5-25 Схема размещения указателей на элементы массива динамических объектов

Для ввода пользователем определенного количества объектов различного типа (прямоугольник, круг, треугольник) приложение должно предоставить пользователю соответствующее меню с возможностью выбора нужного типа объекта. Для создания динамических объектов реализованы функции CreateCyrcle, CreateRect, CreateTriangle (см. раздел описания выше).

При реализации функция AddFigures выводит меню, для выбора нужного для ввода объекта:

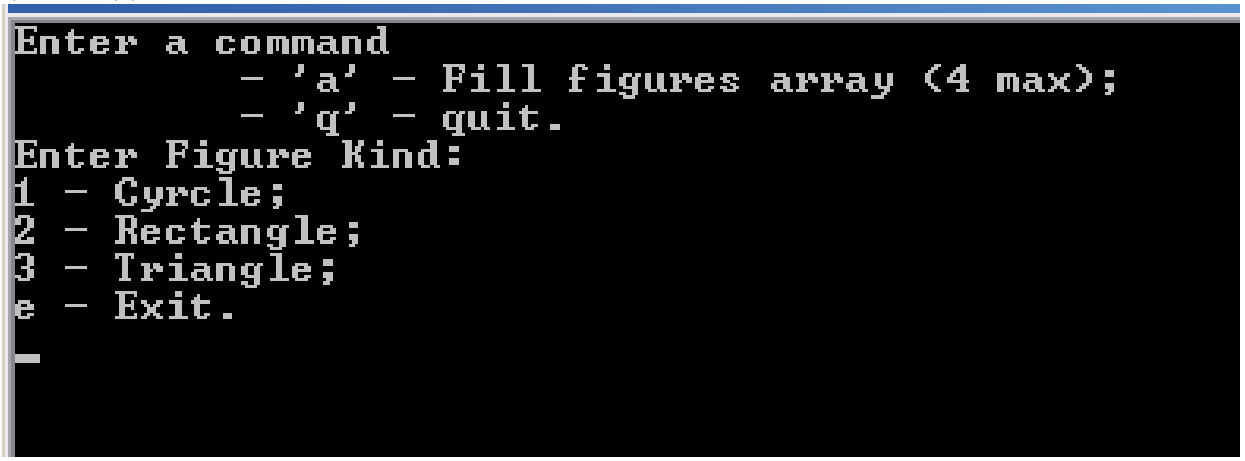

Рис. 5-26 Меню выбора типа геометрической фигуры для добавления в массив

После выбора нужного типа фигуры правление передается в соответствующую функцию инициализации объекта, которая запрашивает данные для ввода:

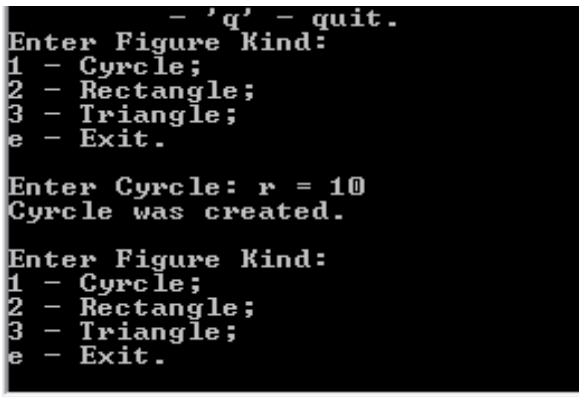

Рис. 5-27 Ввод данных фигуры круга

При успешном вводе данных функция создания объекта возвращает указатель на объект, который размещается в массиве.

Как и в функции main вывод меню осуществляется циклически, пока не будет достигнуто максимальное значение введенных элементов массива или пользователь выберет команду ‘e’ - Exit. При выполнении команды Exit перед выходом из метода, сообщается количество введенных элементов в массив. Управление передается обратно в метод main и выводится основное меню программы для выбора операций по работе с фигурами.

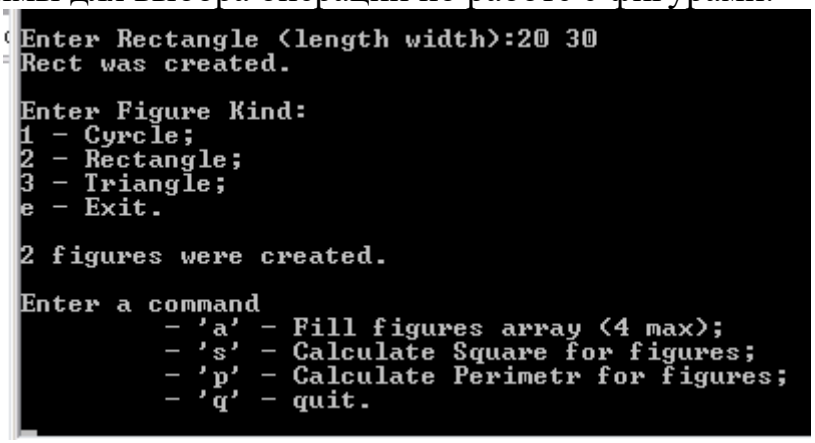

Рис. 5-28 Выход из функции добавления геометрических фигур

Реализация метода ввода фигур AddFigures представлено ниже:

```
// Метод для общего ввода фигур
int AddFigures(void** ppFigs)
{
  int counter = 0;
  bool quit = false;
  while(!quit)
  {
    if(counter > 3)
      return counter;
    printf("Enter Figure Kind: \n");
    printf("1 - Cyrcle;\n");
    printf("2 - Rectangle;\n");
    printf("3 - Triangle;\n");
    printf("e - Exit.\n\n");
```

```
      int key = getch();
      switch(key)
      {
      case '1':
        { // блок для автоматической переменной c
          Cyrcle* c =  CreateCyrcle();
          if(c != NULL) // проверка что объект был создан
          {
            ppFigs[counter] = c;
            counter++;
          }
          break;
        }
      case '2':
        { // блок для автоматической переменной r
          Rect* r =  CreateRect();
          if(r != NULL) // проверка что объект был создан
          {
            ppFigs[counter] = r; // добавление указателя на объект в массив
            counter++;
          }
          break;
        }
      case '3':
        {
          Triangle* t =  CreateTriangle();
          if(t != NULL) // проверка что объект был создан
          {
            ppFigs[counter] = t;
            counter++;
          }
          break;
        }
      case 'e':
      case 'E':
        printf("%d figures were created.\n\n", counter);
        return counter;
        break;
      default:
        printf("Enter correct number. Try again of Exit.\n");
      }
    }
    printf("%d figures were created", counter);
    return counter;
  }
```

Как представлено, код метода добавления фигу достаточно структурирован, поскольку реализация деталей ввода данных скрыты в функциях создания динамических объектов. В методе AddFigures реализуется только меню ввода, обработка команд пользователя в операторе switch, счетчик добавленных объектов (counter) и условия добавления не нулевых указателей созданных объектов в массив ppFigs.

В качестве результата работы метода возвращается значение counter – т.е. колличество введенных пользователем объектов.

Таким образом, реализация команды «’a’ - Fill figures array (4 max)» в главном меню приложения будет следующим:

```
    ...
    switch (key)
    {
      case 'a':
```

```
      case 'A':
        realCount = AddFigures(ppFigArray);
        if(!realCount)
          printf("No figures were putted for calculation!!!\n");
        break;
      case 's':
      ...
```

Для выбранного индивидуального задания реализуйте метод заполнения массива динамических объектов.

Реализация функций обработки данных массива элементов

Для реализации операций расчета и обработки данных геометрических фигур, ссылки на которые размещены в массиве ppFigArray, возможно реализовать специализированные функции, например, функция расчета общей площади фигур, общего периметра, выбора фигуры с наибольшей площадью, вычисления средней площади фигуры и т.п. Остановимся на первых двух – расчет общей площади и общего периметра для фигур.

Для реализации выбранных расчетов в функции достаточно будет передать в функцию в качестве параметров:

Адрес первого элемента массива указателей ppFigArray. Т.е. первый параметр функции будет иметь тип void**.

Количество элементов, для которых будет рассчитано значение общего периметра или площади.

Функций для расчета площади и периметра в качестве результата выполнения должны возвращать результат своих расчетов, т.е. число типа float (поскольку площадь и периметр круга рассчитываются с использования не целого числа Пи).

Прототипы функций расчета необходимо разместить перед функцией main, для того чтоб компилятор узнал, что такие функции есть в приложении и смог проконтролировать правильность определения их вызовов:

```
// методы для обработки фигур
float GetFuguresS(void** ppFigures, int count); // расчет площади
float GetFiguresP(void** ppFigures, int count); // расчет периметра
```

В реализации этих функций необходимо обойти элементы массива, определить и тип, и в соответствии с типом произвести расчет периметра или площади, добавить это значение в общую сумму. Для обхода элементов массива используем цикл for. Для определения типа объекта воспользуемся особенностью реализации структур геометрических фигур. Первое поле данных структур содержит идентификатор типа структуры.

Для использования математических функций в приложение подключим заголовочный файл math.h. Область подключения заголовочных файлов будет выглядеть следующим образом:

```
#include <stdio.h> // /* printf, scanf, NULL */
#include <conio.h>
#include <tchar.h>
```

```
#include <stdlib.h> // /* malloc, free, rand */
#include <math.h>
```

Реализация функции расчета площади геометрических фигур представлена ниже:

```
// методы для обработки фигур
float GetFuguresS(void** ppFigures, int count)
{ // рсчитываемая площадь
  float S = 0;
  // цикл обхода заданного числа элементов массива
  for(int i = 0; i < count; i++)
  {
    void* p = ppFigures[i]; // указатель на объект фигуры
    // получаем тип фигуры из указателя на объект
    Figures kind = *((Figures*)p);
    switch(kind)
    {
    case FCyrcle:
      S += pow((float)((Cyrcle*)p)->Rad,2)*3.14;
      break;
    case FRectangle:
      S += ((Rect*)p)->width*((Rect*)p)->len;
      break;
    case FTriangle:
      {
      Triangle* t = (Triangle*)p;
      // полупериметр
      float pper = (t->side_a+t->side_b+t->side_c)/2;
      // площадь треугольника
      S+= sqrtf(pper*(pper - t->side_a)
                    *(pper - t->side_b)
                    *(pper - t->side_c));
      break;
      }
    default:
      // случай ошибки в данных
      printf("Error in Calculation of Square!!! \n");
      return 0;
    }
  }
  return S;
}
```

Реализация функции расчета периметра геометрических фигур организована аналогичным образом – в цикле получаем указатель на объект, определяем его тип и в зависимости от типа вычисляем периметр:

```
float GetFiguresP(void** ppFigures, int count)
{ // периметр
  float P = 0;
  for(int i = 0; i < count; i++)
  {
    void* p = ppFigures[i]; // указатель на фигуру
    // получаем тип фигуры
    Figures kind = *((Figures*)p);
   switch(kind)
    {
    case FCyrcle: // круг
```

```
        P += 2*((Cyrcle*)p)->Rad*3.14;
        break;
      case FRectangle: // прямоугольник
        P += 2*(((Rect*)p)->width+((Rect*)p)->len);
        break;
      case FTriangle:
        {// автоматическая переменная указателя на треугольник
        Triangle* t = (Triangle*)p;
        //периметр треугольника
        P+= t->side_a + t->side_b + t->side_c;
        break;
        }
      default:
        printf("Error in Calculation of Perimeter!!! \n");
        return 0;
      }
    }
    return P;
  }
```

Использование функций расчета в функции main будет производиться при обработке соответствующих команд «'s' - Calculate Square for figures.» и «‘p' - Calculate Perimeter for figures».

```
...
    switch (key)
    {
      case 'a':
      case 'A':
      ...
      break;
      case 's':
      case 'S':
        if(!realCount) // проверка наличия фигур для расчета
        {
          printf("No figures were putted for calculation!!!\n");
          break;
        }
        float s = GetFuguresS(ppFigArray,realCount);
        printf("Squere of figures is: %f", s);
        break;
      case 'p':
      case 'P':
        if(!realCount) // проверка наличия фигур для расчета
        {
          printf("No figures were putted for calculation!!!\n");
          break;
        }
        float p = GetFiguresP(ppFigArray,realCount);
        printf("Perimeter of figures is: %d", p);
        break;
    ...
```

Для выбранного индивидуального задания реализуйте методы обработки данных для расчета нужных значений в приложении. Результаты вычислений должны выводиться в функции wmain.

Использование указателя на функцию

Обобщая реализованные части приложения можно заключить следующее:

Приложение реализует меню ввода геометрических объектов разного типа (круг, прямоугольник, треугольник) на основе ввода символов с клавиатуры и оператора выбора switch;

Ссылки (адреса) динамически созданных объектов фигур хранятся в одном массиве;

Реализованы функции обработки данных имеющихся геометрических фигур, например, расчет общей площади фигур и расчет общего периметра фигур.

Операции обработки (функции) геометрических фигур явно имеют одинаковые сигнатуры – на вход они принимают адрес первого элемента массива типа void*, т.е. void**, и фактичееское количество элементов в этом массиве. Типом возвращаемого значения этих функций является float. Прототип этих функций имеет обобщенно следующий вид:

```
float someOperation (void**, int);
```

Соответственно различия будут только в имени функции. В таком случае возможно применение указателя на функции обработки массива геометрических фигур. Например, можно объявить указатель на функцию данного типа, инициализировать в операторе выбора и вызвать функцию на выполнение.

Для удобства объявления указателя на функцию, целесообразно использовать ключевое слово typedef, которое позволит создать тип данных указателя на функцию:

```
// объявление метода обратного вызова - указателя на метод
typedef float (*callbackMethod)(void**, int );
```

Представленное объявление определяет тип указателя на функцию “callbackMethod”. Функция на которую будет ссылаться данный указатель должна возвращать значение типа float и принимать 2 аргумента соответственно типов void** и int.

Таким образом, можно модифицировать содержание функции main:

```
bool executeCalc = false; // флаг необходимости выполнения функции
// объявление указателя на функцию
callbackMethod method = NULL;
switch (key)
{
  case 'a':
  ...
  case 's':
  case 'S':
    if(!realCount)
    {
      printf("No figures were putted for calculation!!!\n");
      break;
    }
    method = GetFuguresS;
    printf("Squere of figures is: ");
    break;
  case 'p':
  case 'P':
    if(!realCount)
    {
      printf("No figures were putted for calculation!!!\n");
      break;
```

```
        }
        method = GetFiguresP;
        printf("Perimeter of figures is: ");
        break;
      default:
        printf("Bad command. Try again or Quit.\n");
    }

    // проверка инициализации указателя на функцию
    if(method != NULL)
    { // выполнение выбранного метода для фигур
      float value = method(ppFigArray,realCount);
      printf("%10.2f\n", value);
    }
```

Как представлено в операторе выбора swithc инициализируется указатель на соответствующую функцию и выводится нужный текст. Далее при условии, что указатель на функцию проинициализирован, выполняется вызов метода расчета.

Для выбранного индивидуального задания реализуйте прототип метода обратного вызова и используйте для уканаия на нужные функции обработки данных заполнения массива динамических объектов.

Наблюдение за утечками памяти в инструментарии VisualStudio 2005

Как возможно предположить разработанное приложение будет иметь утечки памяти, в случае если указатели в массиве ppFigArray будут перезаписываться, без удаления соответствующих объектов. Таким образом, объекты в памяти приложения будут оставаться, занимать нужное пространство даже если приложение уже не имеет ни одного указателя на эти объекты и никогда его уже не будет использовать.

Инструментарий среды разработки VisualStudio 2005 предоставляет возможность показать такие объекты в памяти приложения, которые остаются не уничтоженными после завершения работы приложения. Т.е. фактические утечки памяти. Кроме этого, с использованием специальных отладочных библиотек имеется возможность определить момент создания такого объекта , который в последствии будет «потерян» (своевременно не уничтожен) приложением.

Для активации режима отслеживания утечек памяти необходимо выполнить следующие действия:

Подключить нужные заголовочные файлы библиотек окружения разработки (stdlib.h и crtdbg.h).

Вызвать функцию сбора данных об утечках памяти _CrtDumpMemoryLeaks в конце выполнения функции wmain.

Реализация этих изменений в коде будет выглядеть следующим образом:

В файле подключения заголовочных файлов stdafx.h:

```
#include <stdio.h> // /* printf, scanf, NULL */
#include <conio.h>
```

```
#include <tchar.h>
#include <stdlib.h> // /* malloc, free, rand */
#include <math.h>
#include <crtdbg.h> // определение утечки памяти
```

В завершении метода wmain вызываем производим вызов функции _CrtDumpMemoryLeaks:

```
    _CrtDumpMemoryLeaks(); // собираем утечки памяти
    return 0;
}
```

Теперь если в приложении произвести ввод фигур повторно, т.е. при наличии фигур в массиве вызвать команду «'a' - Fill figures array» и ввести фигуры заново, то ссылки на первоначально расположенные в массиве фигуры будут «потеряны» приложением. Объекты будут занимать память процесса но будут не доступны. При завершении работы приложения будет вызвана функция _CrtDumpMemoryLeaks, которая выведет информацию о наличии таких потерянных объектов в окно Вывод (Output). Точнее сказать будет выведена информация о наличии блоков памяти определенного размера, которые были выделены приложением но не были очищены.

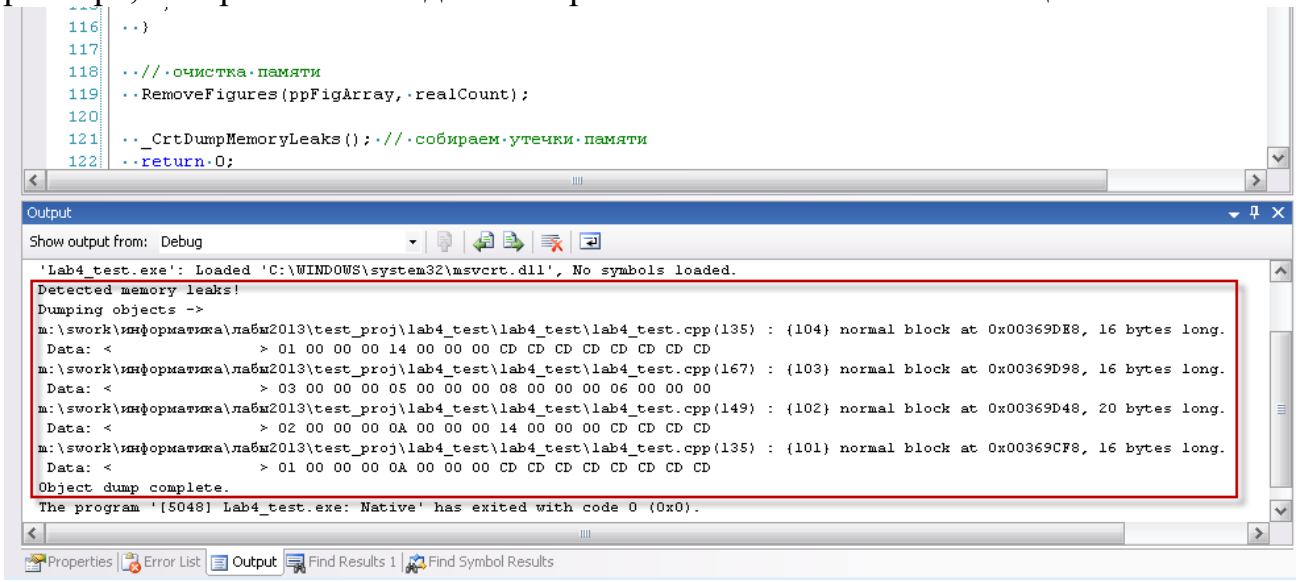

Рис. 5-29 Вывод информации об утечках памяти в инструментальном окружении VisualStudio 2005

При детальном рассмотрении информации об утечках памяти можно выявить информацию необходимую для определения времени и места создания объекта, который в результате не был уничтожен корректно. Например, следующая информация об утечках показывает:

```
Dumping objects ->
m:\tmp\lab4_test\lab4_test\lab4_test.cpp(135) : {104} normal block at
0x00369DE8, 16 bytes long.
 Data: < > 01 00 00 00 14 00 00 00 CD CD CD CD CD CD CD CD
m:\tmp\lab4_test\lab4_test\lab4_test.cpp(167) : {103} normal block at
0x00369D98, 16 bytes long.
 Data: < > 03 00 00 00 05 00 00 00 08 00 00 00 06 00 00 00
```

```
...
```

1. Имя файла, в котором произошла ошибка работы с динамически распределяемой памятью (m:\tmp\lab4_test\lab4_test\lab4_test.cpp);

2. Номер строчки, в которой произошло распределение памяти (135) – эта информация доступна не во всех случаях.

3. Номер операции по распределению памяти (индекс – {104}), который возможно использовать впоследствии для установки специальной точки останова и выявления момента создания объекта (выделения памяти под объект). Индексы указываются в обратном хронологическом порядке.

4. Адрес по которому располагается объект – фактически это значение указателя на объект который был потерян.

5. Размер распределенной области памяти (16 bytes long)

6. Символьное представление и значения первых байтов в обозначенной области памяти
(Data:< > 03 00 00 00 05 00 00 00 08 00 00 00 06 00 00 00).

Таким образом, можно определять утечки памяти в приложении.

Для выбранного индивидуального задания реализуйте контроль за утечками памяти.

Уничтожение динамически созданных объектов

Для корректной очистки памяти достаточно создать метод - RemoveFigures, который будет освобождать память, выделенную для объектов геометрических фигур. В реализации метода также будет необходимо определять тип объекта, на который ссылается указатель, так же как это сделано в функциях расчета. Метод очистки будет принимать в качестве параметров указатель на массив указателей объектов void** и фактическое количество объектов, которые нужно удалить из памяти.

```
// метод для очистки памяти фигур
void RemoveFigures(void** ppFigs, int count)
{
  int counter = 0; // индекс текущего объекта
  void* p = ppFigs[counter]; // указатель на текущий объект
  while(p != NULL)
  {
    // получаем тип фигуры
    Figures kind = *((Figures*)p);
    switch(kind)
    {
    case FCyrcle:
      free((Cyrcle*)p); // очистка выделенной памяти
      printf("Cyrcle was removed. \n");
      break;
    case FRectangle:
      free( (Rect*)p);
      printf("Rect was removed. \n");
      break;
    case FTriangle:
      free( (Triangle*)p);
      printf("Triangle was removed. \n");
      break;
```

```
      default:
        printf("Error in RemoveFigures!!! \n");
        return;
      }
      ppFigs[counter] = NULL; // обнуление указателя в массиве
      counter++;
      if(counter == count)
        return;
      p = ppFigs[counter];
    }
  }
```

Вызов метода необходимо разместить в конце функции main и в начале функции AddFigures.

В функции main вызов будет выглядеть следующим образом:

```
  // очистка памяти
  RemoveFigures(ppFigArray, realCount);

  _CrtDumpMemoryLeaks(); // собираем утечки памяти
  return 0;
}
```

В методе AddFigures очистка массива фигур будет производиться при условии, что первый указатель в массив0е не равен NULL:

```
int AddFigures(void** ppFigs)
{
  if(ppFigs[0] != NULL) // проверка что первый элемент массива не нулевой
    RemoveFigures(ppFigs, 4);

  int counter = 0;
  bool quit = false;
  ...
```

Для выбранного индивидуального задания реализуйте функцию для корректного удаления динамически созданных объектов. Используйте функцию в методе заполнения массива динамических объектов.