Кафедра УИТС
Объектно-ориентированное программирование на с/с++
Лекция 1
Введение. Типы. Классы. Операторы. Перегрузка.

Автор курса:

доцент Разумовский Алексей Игоревич

oopCpp@yandex.ru

Книги и ссылки

1. Страуструп Б. Язык программирования С++. Москва: Вильямс, 2011
2. Мейерс С. - Эффективный и современный С++. 42 рекомендации по использованию С++11 и С++14, 2016
3. Д. Элджер - Библиотека программиста C++ - Гарвард: World Group, 2004.
4. **Стив Макконнелл Совершенный код**
5. **http://www.cplusplus.com**

Методология познания

Метод проб и ошибок.

Чем больше сделаете ошибок, тем быстрее научитесь.

Интегрированная Среда Разработки (IDE) для С++:

Microsoft Visual Studio vv. 2019-2022

Исследования, проведенные в 1970-х годах в Массачусетском технологическом институте и исследовательском центре Xerox Palo Alto Research Center, привели к разработке философии объектно-ориентированного программирования (ООП) и созданию языков реализации, включая Smalltalk, Java, C++.

Предполагается, что вы уже знаете понятия цикла (**for, while, do{ }while**), главной функции (**main/WinMain**), операторов **if/else, switch, return.**

Также предполагается, что вы знаете, что такое указатель, ссылка и их взаимоотношения, а также понимаете арифметику указателей, включая операторы инкремента (++) и декремента (--).

Я буду постепенно добавлять новые термины и использовать их в примерах.

Понятия типа, класса, полиморфной иерархии, глубокого копирования, контейнеров, итераторов, адаптеров, функторов, умных указателей, потоков выполнения будут раскрываться последовательно от лекции к лекции.

Запреты использования.

При решении задач на лабах / семинарах запрещено применять ключевое слово **auto**, а также двухсекционный цикл for :

for (auto l : v) .

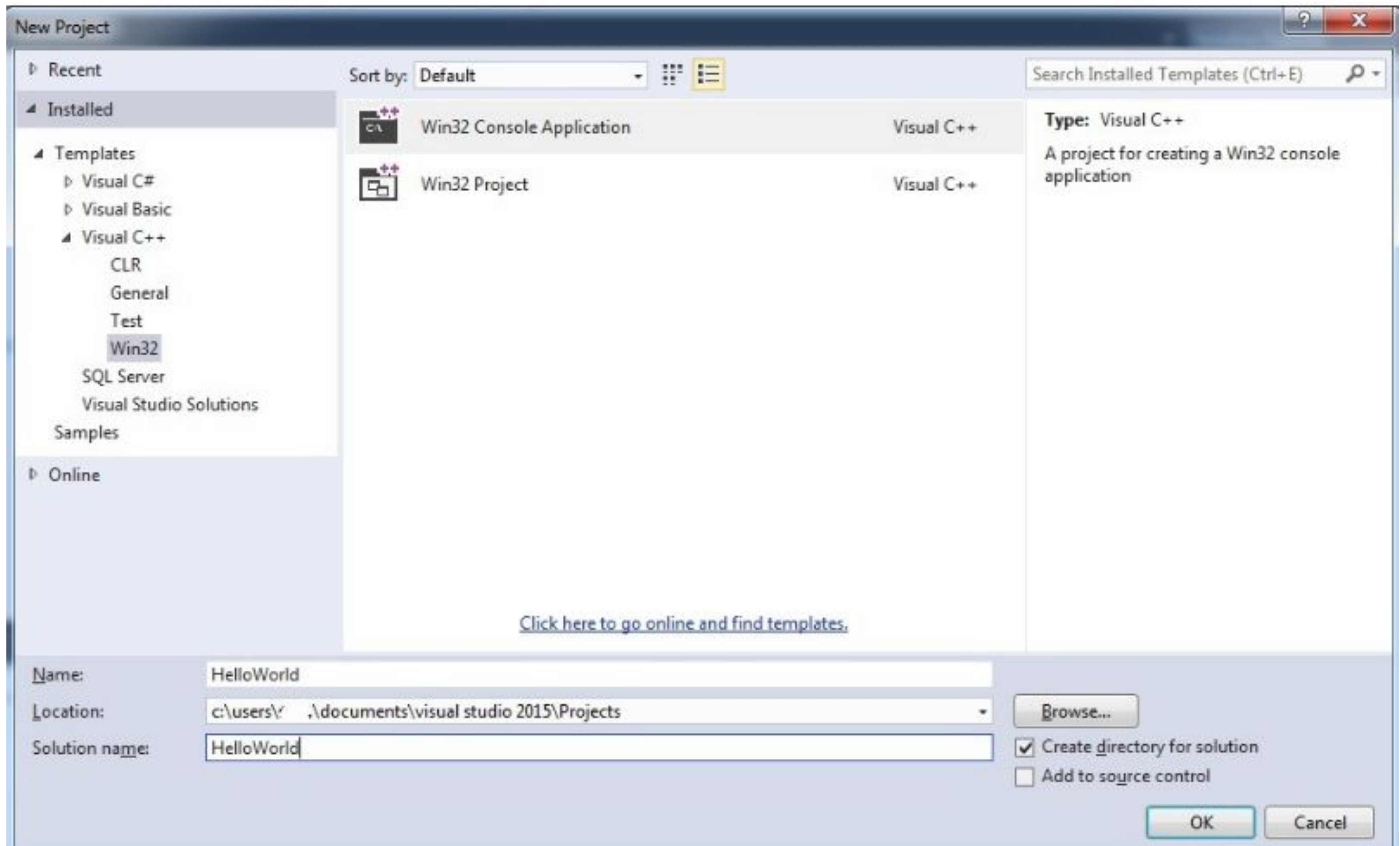
for следует пользоваться только трехсекционным, например:

for (T t = begin() ; t != end(); ++t)

Чего нет в книгах и учебниках

- Каждый вызов функции – это отражение вашего желания получить нужный результат. Это **здравый смысл** операции!
- Каждый элемент программы несет в себе **ответственность** за его использование. Поэтому, например, применение умных указателей, которые сами могут освободить выделенную память – лишает программиста ответственного поведения.

Создание проекта



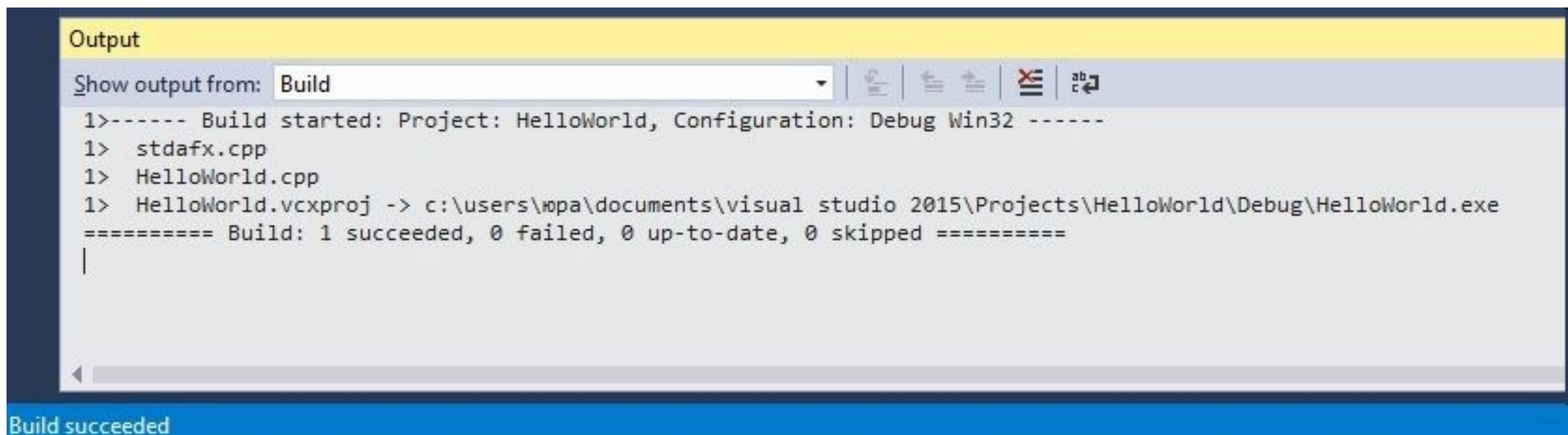
о

Компиляция проекта

HelloWorld - Microsoft Visual Studio Express 2015 for Windows Desktop

```
#include "stdafx.h"
#include <iostream>

int main()
{
    std::cout << "Hello world!" << std::endl;
    return 0;
}
```



В начале файла программы следует указать нужные хидеры:

- #include<iostream>
- #include<fstream>
- #include<string>
- #include<vector> // или set или map
- #include<algorithm>
- #include<cmath>
- **using namespace std;** // использовать пространство имен std

Файлы

3 способа доступа к файлам данных:

- Windows API: CreateFile (и другие функции для работы с файлами)
- C: fopen (и прочие функции для работы с файлами из stdio.h)
- C++: fstream (ifstream, ofstream) (из stl)

Потоки данных

- C: printf и scanf (#include<stdio.h>)
- C++: cin и cout (#include<iostream>)

Файловые потоки:

(#include<fstream>)

Пример

```
#include <fstream>
#include <libc.h>

void error (char* s, char* s2 ="")
{
   cerr << s << ' ' << s2 << '\n';
   exit(1);
}

int main(int argc, char* argv[])
{
    ifstream from(“file1.txt”);
    ofstream to(“file2.txt”);
              char ch;
    while (from.get(ch)) to.put(ch);

    if (!from.eof() || to.bad())
       error("something strange happened");

    return 0;
 }
```

Стандартная библиотека шаблонов STL

Включает, основанную на методологии обобщенного программирования библиотеку классов, содержащую:

- Контейнеры (для хранения данных произвольного типа)
- Итераторы (для осуществления доступа к данным контейнеров)
- Алгоритмы
- Адаптеры контейнеров, итераторов и функционалов.
- Стримы (streams) для выполнения потоковых операций с данными
- Потоки исполнения (threads) и элементы синхронизации в потоках
- Прочие элементы, например, умные указатели

Пример stl-алгоритма: сортировка

```
#include<vector>
#include<algorithm>
using namespace std;

vector <int> v;
for ( int i=10; i< 0; i--)
   v.push_back(i);                // 10,9,8,7,6,5,4,3,2,1

vector <int>::iterator it= v.begin();

sort (v.begin(),v.end());       // 1,2,3,4,5,6,7,8,9,10
```

Три основных свойства ООП

- Абстракция (данных) (отвлечение)
- Инкапсуляция (скрытие)
- Полиморфизм (разнообразие)

Пример ООП- программы

```
  class A{
public:
A(){ }
~A(){ }
 virtual void print(){cout<<"a"<<endl;}
};

class B: public A{
public:
  void print(){cout<<"b"<<endl;}я
};

 void print_all ( A** v  , int size )  {
     // распечатать все
         int i = 0;
           while ( i<size )
           {
              (v [ i ]) -> print (   );
             i = i + 1;
           }
 }
```

```
int _tmain( ){
 A a;
B b;
 A m[2];
 m[1]= b;
A*m0[2]={&m[0],&m[1]};
    print_all(m0,2);
     A*m1[2];
  m1[0]=new A;
  m1[1]= new B;
          print_all( m1,2);
return 1;
}

// Результат выполнения программы:
a
a
a
b
```

Отладчик MS Visual Studio

Клавиши отладки:

F9 – поставить или снять точку останова программы

F10 – совершить одно отладочное действие: выполнить одну результирующую операцию.

F11 – войти внутрь функции

SHIFT + F11 – выйти из функции

Окна отладки:

В процессе отладки программы можно открывать большинство окон отладчика. Чтобы просмотреть список окон отладчика, установите точку останова и начните отладку. Когда точка останова будет достигнута и выполнение остановится, выберите пункт Отладка / Окна.

Окно	Сочетание клавиш	Раздел
Breakpoints	CTRL+ALT+B	Use Breakpoints
Exception Settings	CTRL+ALT+E	Manage Exceptions with the Debugger
Output	CTRL+ALT+O	Output Window
Watch	CTRL+ALT+W, (1, 2, 3, 4)	Watch and QuickWatch Windows
QuickWatch	SHIFT+F9	Watch and QuickWatch Windows
Autos	CTRL+ALT+V, A	Autos and Locals Windows
Locals	CTRL+ALT+V, L	Autos and Locals Windows
Call Stacks	CTRL+ALT+C	How to: Use the Call Stack Window
Immediate	CTRL+ALT+I	Immediate Window
Threads	CTRL+ALT+H	Debug using the Threads Window
Modules	CTRL+ALT+U	How to: Use the Modules Window
Tasks	CTR:+SHIFT+D, K	Using the Tasks Window
Processes	CTRL+ALT+Z	Debug Threads and Processes
Memory	CTRL+ALT+M, (1, 2, 3, 4)	Memory Windows
Disassembly	CTRL+ALT+D	How to: Use the Disassembly Window
Registers	CTRL+ALT+G	How to: Use the Registers Window

Окно	Сочетание клавиш	Раздел
Точки останова	CTRL+ALT+B	Использование точек останова
Параметры исключений	CTRL+ALT+E	Управление исключениями с помощью отладчика
Вывод	CTRL+ALT+O	Окно выходных данных
Контрольное значение	CTRL+ALT+W, (1, 2, 3, 4)	Окна "Контрольные значения" и "Быстрая проверка"
Контрольное значение	SHIFT+F9	Окна "Контрольные значения" и "Быстрая проверка"
Авто	CTRL+ALT+V, A	Окна переменных
Локальные	CTRL+ALT+V, L	Окна переменных
Интерпретация	CTRL+ALT+I	Окно интерпретации
Стеки вызовов	CTRL+ALT+C	Использование окна стека вызова
Потоки	CTRL+ALT+H	Использование окна потоков
Модули	CTRL+ALT+U	Использование окна модулей
Задачи	CTRL+SHIFT+D, K	Использование окна задач
Процессы	CTRL+ALT+Z	Отладка потоков и процессов
Память	CTRL+ALT+M, (1, 2, 3, 4)	Окно памяти
Дизассемблированный код	CTRL+ALT+D	Использование окна дизассемблирования
Регистры	CTRL+ALT+G	Использование окна регистров

Типы

Тип называется встроенным, если компилятор знает, как представить объекты такого типа и какие операторы к нему можно применять (такие как + и -) без уточнений в виде объявлений, которые создает программист в исходном коде.

Типы, не относящиеся к встроенным, называют типами, определенными пользователем. Они могут быть частью стандартной библиотеки (например, классы string, vector и ofstream), или типами, создаваемыми самим программистом.

Классы

Ключевым понятием C++ является класс: class.

<u>Класс</u> - это определяемый пользователем **<u>тип</u>**.

Классы обеспечивают упрятывание данных, их инициализацию, неявное преобразование пользовательских типов, динамическое задание типов, контролируемое пользователем управление памятью и средства для перегрузки операций.

В языке C++ концепции контроля типов и модульного построения программ реализованы более полно, чем в C.

Кроме того, C++ содержит усовершенствования, прямо с классами не связанные: символические константы, функции-подстановки, стандартные значения параметров функций, перегрузка имен функций, операции управления свободной памятью и ссылочный тип.

Операции

Как и встроенные типы, большинство типов, определенных пользователем, описывают операции. Например, класс vector содержит операции [ ] и size(), класс ofstream — операцию <<, а определенный программистом класс Shape — операции:

add (Point) и set_color (int)

```
class Shape {
  void add  (Point& p);
  void  set_color (int c) ;
}
```

Какие типы можно признать хорошими?

Типы являются хорошими, если они позволяют прямо отразить идею в коде. Когда мы пишем программу, нам хотелось бы непосредственно воплощать идеи в коде так, чтобы мы сами, наши коллеги и компилятор могли понять, что мы написали.

Когда мы хотим выполнять арифметические операции над целыми числами, нам подойдет тип **int.**

Когда хотим манипулировать текстом, класс **string** - хороший выбор.

Когда хотим манипулировать входной информацией для калькулятора, нам нужны классы Token и Token_stream.

Необходимость классов имеет два аспекта

- Представление. Тип знает, как представить данные, необходимые в объекте.
- Операции. Тип знает, какие операции можно применить к объектам.

Эту концепцию, лежащую в основе многих идей, можно выразить так: "нечто" имеет данные для представления своего текущего значения — которое иногда называют текущим состоянием, — и набор операций, которые к ним можно применить.

Подумайте о компьютерном файле, веб-странице, плеере, чашке кофе, телефоне, телефонном справочнике: все они характеризуются определенными данными и имеют более или менее фиксированный набор операций, которые можно выполнить. В каждом случае результат операции зависит от данных — текущего состояния объекта.

Пример класса

```
class X {
public:
   int m_member;                // данные-члены
   X(): m_member(0){ }        //  конструктор по умолчанию
   X(int a): m_member(a){ } //  конструктор
 virtual ~X() { }                     //  виртуальный деструктор
    int Func(int v) ;               // функция-член
private:
    // закрытые члены    // интерфейсы
};
int Func( int v) {    // определение функции вне класса
     int old = m_member;
     m_member = v;
   return old;
 }
       // применение класса
X var;                         // var — переменная типа X
var.m_member =7;  // присваиваем значение члену m_member объекта var
int x = var.Func(9);  // вызываем функцию-член Func(int) объекта var
```

Разработка класса

```
class Date {
public:
int y; // год
int m; // месяц года
int d; // день месяца
};
    // создаем объект
 Date today;
  //  простая инициализация не безопасна
today.y = 2018; today.m = 9; today.d = 17;
    //  появление ошибки
today.m = -1; /*или*/ today.m = 25;
   // тогда введем в класс функцию:
void init_day(Date& dd, int y, int m, int d) {
// проверяет, является ли (y,m,d) правильной датой
// если да, то инициализирует объект dd
}
```

Если мы забудем немедленно после создания инициализировать объект today, то до вызова функции init_day() этот объект будет иметь неопределенное значение.

Чтобы объекты были гарантированно корректными, важно скрыть представление, предусматривая конструктор, создающий только корректные объекты, и разработать все функции-члены так, чтобы они получали и возвращали только корректные значения.

```
class Date {
private:
int y; // год
int m; // месяц года
int d; // день месяца
public:
  Date (int y, int m, int d) ;
  int month() { return m; }
  int day() { return d; }
int year() { return y; }
};
```

Сообщения об ошибках

Что делать при обнаружении некорректной даты? В каком месте кода происходит поиск некорректных дат? Самым очевидным местом для этого является место создания объекта класса Date, т.е. конструктор.

```
class Date {
  public:
class Invalid { }; // используется как исключение
Date (int y, int m, int d) ; // проверка и инициализация даты
  private:
int y, m, d; // год, месяц, день
bool check(); // если дата правильная, возвращает true
};
Date::Date(int yy, int mm, int dd)
: y(yy), m(mm), d(dd) // инициализация данных - членов класса
{
if (!check ()) throw Invalid (); // проверка корректности
}
```

```
bool Date::check ()  // возвращает true, если дата корректна
{
if (m<1 || 12<m)   return false;
else    return   true;
}
 //  можно написать следующий код:
void  f (int x, int y)
 try {
Date dxy (2019,x,y);
cout << dxy <<  endl;
}
catch(  Date::Invalid) {
error("invalid date");
}
```

Операторы

В пользовательских классах существует возможность определения операторов:

```
class  T {
 int i=0;
 public:
  operator ++ () { i++;}
}:
```

Некоторые арифметические операторы:

```
R& T::operator =(S b);
R T::operator +(S b);
R T::operator -(S b);
R T::operator +();
R T::operator -();
R T::operator *(S b);
R T::operator /(S b);
R T::operator %(S b);
```

Некоторые операторы сравнения и логические операторы :

```
R T::operator ==(S b);
R T::operator !=(S b);
R T::operator >(S b);
R T::operator <(S b);
R T::operator !();
R T::operator &&(S b);
R T::operator ||(S b);
```

Операторы действия с указателями и обращения к члену класса :

```
R T::operator [ ](S b);
R T::operator *();
R T::operator &();
R* T::operator ->();
```

Перегрузка операторов и функций

Перегрузка операторов - это мощная возможность, которая позволяет для типов, определяемых пользователем, использовать те же операторы, что и для встроенных типов.

Сторонники других языков, которые не поддерживают перегрузку операторов, утверждают, что эта возможность сбивает с толку и очень сложна, и, следует признать, может быть перегружено очень много операторов, соответствующих любому поведению.

Но когда дело касается простого инкремента и декремента, хорошо иметь возможность изменить поведение класса так, как этого хочется.

Перегрузка оператора ++

```
 Date operator++(Date d){
 // префиксный инкрементный оператор:
     // увеличивает дату на 1 день
int d,m,y;
d= d.day();
m=d. month() ;
y=d. year();
  if(d<31){
    d=d+1;
   d.set_day( d);
  }
else {
 if( m<12){
   m++;   // тоже, что m=m+1
    d= 1;
    d.set_day( d);
    d.set_ month( m);
  }  }
else{
    y++;
    d= 1;
    m=1;
   d.set_day( d);
   d.set_month( m);
    d.set_year( y);
  }
 return d;
)
```

Константные функции-члены

Некоторые переменные должны изменяться, потому они так и называются, а некоторые— нет; иначе говоря, существуют переменные, которые не изменяются. Обычно их называют константами, и для них используется ключевое слово **const**.

```
void some_function (Date& d, const Date& const_d) {
 int a = d.day();   // OK
 int b = const_d.day () ;
 d.set_day(3) ;     // OK
 const_d.set_day (3) ; // ошибка
}
```

Здесь подразумевается, что переменная d будет изменяться, а переменная const_d - нет; другими словами, функция some_function () не может изменить переменную const_d.

```
class Date {
    public:
// . . .
int day() const;     // константный член: не может изменять объект
int month() const; // константный член: не может изменять  объект
int year() const;  //  константный член: не      может  изменять  объект
void set_day(int n); //   неконстантный    член:     может    изменять объект
void set_month(int n);  // неконстантный член: может изменять объект
void set_year(int n);   // неконстантный член: может изменять объект
    private:
int y;      // год
int m;
int d;      // день
};
Date d ( 2000, 3, 20); const Date cd(2018, 9, 21);
cout << d.day() << " - " << cd.day()<< endl; // OK
d. set_day(1); // OK
cd. set_day(1); // ошибка: cd — константа
```

Выражения в c++

Разрешение области видимости	
N :: m	m находится в пространстве имен N; N — имя пространства имен или класса
:: m	m находится в глобальном пространстве имен

Постфиксные выражения	
x.m	Доступ к члену класса; x должен быть объектом класса
p->m	Доступ к члену класса; p должен быть указателем на объект класса; эквивалентно (*p).m
p[x]	Индексирование; эквивалентно *(p+x)
f(lst)	Вызов функции; вызов функции f со списком аргументов lst
T(lst)	Создание: создание объекта T со списком аргументов lst
v++	(Постфиксная) инкрементация; значение v++ равно значению v до инкрементации
v--	(Постфиксная) декрементация; значение v-- равно значению v до инкрементации
typeid(x)	Идентификация типа объекта x во время выполнения программы
typeid(T)	Идентификация типа T во время выполнения программы
dynamic_cast<T>(x)	Приведение объекта x к типу T с проверкой во время выполнения программы
static_cast<T>(x)	Приведение объекта x к типу T с проверкой во время компиляции программы
const_cast<T>(x)	Непроверяемое преобразование, которое сводится к добавлению или удалению спецификатора const у типа объекта x, чтобы получить тип T

Выражения в c++

Унарные выражения

`sizeof(T)`	Размер типа `T` в байтах
`sizeof(x)`	Размер типа, к которому относится объект `x` (в байтах)
`++v`	(Префиксная) инкрементация; эквивалентно `v+=1`
`--v`	(Префиксная) декрементация; эквивалентно `v-=1`
`~x`	Дополнение к `x`; `~` — побитовая операция
`!x`	Отрицание `x`; возвращает `true` или `false`
`&v`	Адрес переменной `v`
`*p`	Содержание объекта, на который ссылается указатель `p`
`new T`	Создает объект типа `T` в свободной памяти
`new T(lst)`	Создает объект типа `T` в свободной памяти и инициализирует его объектом `lst`
`new(lst) T`	Создает объект типа `T` в области памяти, заданной аргументом `lst`
`new(lst) T(lst2)`	Создает объект типа `T` в области памяти, заданной аргументом `lst`, и инициализирует его аргументом `lst2`
`delete p`	Удаляет объект, на который ссылается указатель `p`
`delete[] p`	Удаляет массив объектов, на который ссылается указатель `p`
`(T)x`	Приведение в стиле языка С; приведение объекта `x` к типу `T`

Выражения в с++

`x*y`	Умножение **x** на **y**
`x/y`	Деление **x** на **y**
`x%y`	Деление по модулю (остаток от деления) **x** на **y** (не для типов с плавающей точкой)
`x+y`	Сложение **x** и **y**
`x-y`	Вычитание **y** из **x**
`x<y`	**x** меньше **y**; возвращает объект типа `bool`
`x<=y`	**x** меньше или равно **y**
`x>y`	**x** больше **y**
`x>=y`	**x** больше или равно **y**
`x==y`	**x** равно **y**; возвращает значение типа `bool`
`x!=y`	**x** не равно **y**
`x&&y`	Логическое "и"; возвращает значения `true` и `false`; вычисляет значение **y**, только если **x** имеет значение `true`
`x||y`	Логическое "или"; возвращает значения `true` и `false`; вычисляет значение **y**, только если **x** имеет значение `false`
`x?y:z`	Если **x** равно `true`, то результат равен **y**; иначе — равен **z**

Выражения в c++

v=x	Присваивает **x** переменной **v**; результат равен **v**
v*=x	Аналог **v=v*(x)**
v/=x	Аналог **v=v/(x)**
v%=x	Аналог **v=v%(x)**
v+=x	Аналог **v=v+(x)**
v-=x	Аналог **v=v-(x)**

Основные термины и понятия

Типы **встроенные и пользовательские**.

Класс - это определяемый пользователем **тип.**

Большинство типов, определенных пользователем, описывают **операции.**

Конструкторы - по умолчанию, пользовательские, копирующие, перемещающие.

Константные и статические функции-члены.

Если нужно явно сослаться на объект, из которого вызвана функция-член, то можно использовать зарезервированный указатель **this.**

Функция, не являющаяся членом класса, может получить доступ ко всем членам класса, если ее объявить с помощью ключевого слова **friend.**

Члены класса, являющиеся целочисленными константами, функциями или типами, могут быть определены как в классе, так и вне его.

Класс можно определить **производным** от других классов. В этом случае он **наследует** члены классов, от которых происходит (своих базовых классов).

Виртуальная функция — это функция-член, определяющая интерфейс вызова функций, имеющих **одинаковые** имена и **одинаковые** типы аргументов в производных классах.

Абстрактный класс - это класс, который можно использовать только в качестве базового класса.

Деструктор – виртуальный и невиртуальный – функция, где освобождаются ресурсы класса.

Конструкторы

По умолчанию – без параметров

Пользовательские (заданные программистом, в том числе и конструктор по умолчанию)

Копирующие конструкторы

Перемещающие конструкторы

Копирование объектов классов

Всегда следует создавать объекты, предусматривая инициализацию и конструкторы.

Это самые важные члены класса: для того чтобы написать их, необходимо решить, **как** инициализировать объект и что значит **корректность** его значений.

Есть два вида копирования:

Конструктор копирования

```
Date (const Date &obj) {
  y= obj.y;
  m= obj.m;
  d = obj.d;
}
```

Оператор присваивания копированием

```
Date & Date::operator=(const Date &obj){
    y= obj.y;
      m= obj.m;
      d = obj.d;
   return *this;
}
```

Конструкторы по умолчанию

```
string s;  // значение по умолчанию: пустая строка ""
vector<string> v1;// значение по умолчанию: вектор без элементов
vector<string> v2 (10) ; // вектор, по умолчанию содержащий 10 строк

string s = string (); // вызывается конструктор по умолчанию
vector<string> v1 = vector<string> () ; // вызывается конструктор по умолчанию:
vector<string> v2 (10,string()) ; // вектор, содержащий 10 строк, для каждой   из
                                  // которых вызывается конструктор по умолчанию
```

Пользовательские конструкторы

К пользовательским конструкторам относятся конструкторы с параметрами.

```
string s (“test”);     // значение “test”
vector< int > v1 (1,1);  //  вектор с одним элементом значения 1

vector<string> v2 (10, ”test”  ) ; // вектор, содержащий 10 строк, для каждой   из
                                   // которых вызывается пользовательский
                                   // конструктор с одним параметром “test”
```

Конструкторы копирования и перемещения

```
MyClass (const MyClass&  obj);  // сигнатура конструктора копирования
MyClass (  MyClass&&  obj);  // сигнатура конструктора перемещения
```

Деструктор

- Это функция, где освобождаются ресурсы класса.

```
class MyClass {
public:
  int a;
  char* c;
 MyClass () { a=23; c= new char(‘2’); }  // конструктор по умолчанию
 ~MyClass (){  delete c;}  // невиртуальный деструктор освобождает ресурс

};
class MyBaseClass { // если это базовый класс полиморфной иерархии
                           // тогда деструктор обязан быть виртуальным
 …                         // какие-то члены данных
public:
 virtual ~MyClass (){  delete c;}  //  виртуальный деструктор

};
```

Дружественные классы и функции

Функция, не являющаяся членом класса, может получить доступ ко всем членам-класса, если ее объявить с помощью ключевого слова **friend**.

```
// требует доступа к членам классов Matrix и Vector members:
Vector operator*   (const Matrix & , const Vector & );

class Vector {
  friend Vector operator*(const Matrix & , const Vector & ); // есть доступ
};
class Matrix {
  friend Vector operator*(const Matrix & , const Vector & ); // есть доступ
};
```

Другое предназначение ключевого слова **friend** - обеспечивать функцию доступа, которую нельзя вызывать как функцию-член.

```
class Iter {
 public:
 int distance_to(const iter& a) const;
 friend int difference(const Iter& a, const Iter& b);
};
void  f (Iter& p, Iter& q) {
int x = p.distance_to(q) ; // вызов функции-члена
int y = difference (p,q);  // вызов с помощью математического синтаксиса
```

Функцию, объявленную с помощью ключевого слова **friend**, нельзя объявлять виртуальной.

Статические члены класса

(фрагмент из книги Б. Страуструпа)

Наиболее успешная реализация некоторых типов требует, чтобы все объекты этого типа имели некоторые общие данные. Лучше, если эти данные можно описать как часть класса. Например, в операционных системах или при моделировании управления задачами часто нужен список задач:

```
class task {
   // ...
   static task* chain;
};
```

Описав член chain как статический, мы получаем гарантию, что он будет создан в единственном числе, т.е. не будет создаваться для каждого объекта task. Но он находится в области видимости класса task, и может быть доступен вне этой области, если только описан в общей части. В этом случае имя члена должно уточняться именем класса:

```
if (task::chain == 0)  // какие-то операторы
```

Использование статических членов класса может заметно сократить потребность в глобальных переменных.

Описывая член как статический, мы ограничиваем его область видимости и делаем его независимым от отдельных объектов его класса. Это свойство полезно как для функций-членов, так и для членов, представляющих данные:

```
class task {
   // ...
   static task* task_chain;
   static void shedule(int);
   // ...
};
```

Но описание статического члена - это только описание, и где-то в программе должно быть единственное определение для описываемого объекта или функции, например, такое:

```
task* task::task_chain = 0;
void task::shedule(int p) { /* ... */ }
```

Шаблоны

Шаблон (**template**) — это класс или функция, параметризованные набором типов и/или целыми числами.

```
template<class T>
class vector {
public:
  int size() const;
private:
  int sz; T* p;
};
template <class T>
int vector<T>::size() const {
    return sz;
}
```

В списке шаблонных аргументов ключевое слово **class** означает тип; его эквивалентной альтернативой является ключевое слово **typename**.

Функция-член шаблонного класса по умолчанию является шаблонной функцией с тем же списком шаблонных аргументов, что и у класса.

Шаблонные аргументы

Аргументы шаблонного класса указываются каждый раз, когда используется его имя.

```
vector<int> v1;   // OK
vector v2;      // ошибка: пропущен шаблонный аргумент
vector<int,2> v3;       // ошибка: слишком много шаблонных аргументов
vector<2> v4;       // ошибка: ожидается тип шаблонного аргумента
```

Аргументы шаблонной функции обычно выводятся из ее аргументов.

```
template< class T>
 T find(vector<T>& v, int i) {
   return v[i];
}
vector<int> v1;
vector<double> v2;
int x1 = find (v1, 2) ; // здесь тип T – это  int
int x2 = find (v2,2) ;   // здесь тип T – это  double
```

Специализация шаблонов

Вариант шаблона для конкретного набора шаблонных аргументов называется **специализацией**. Процесс генерации специализаций на основе шаблона и набора аргументов называется конкретизацией шаблона. Как правило, эту задачу решает компилятор, но программист также может самостоятельно определить отдельную специализацию. Обычно это делается, когда общий шаблон для конкретного набора аргументов неприемлем.

```
template< class T> struct Compare { //  обобщенное  сравнение
  bool operator ()  (const T& a, const T& b) const {
   return a<b;
  }
};
template< > struct Compare < const char*> {  // сравнение C-строк
 bool operator()(const char* a, const char* b) const {
   return   strcmp(a,b)==0;
  }
};
Compare<int> c2;              // общее сравнение
Compare<const char*> c;    // сравнение C-строк
bool b1 = c2(1,2);              // общее сравнение
bool b2 = c ("asd", "dfg");     // сравнение C-строк
```

Шаблонные типы членов-классов

Шаблон может иметь как члены, являющиеся типами, так и члены, не являющиеся типами (как данные-члены и функции-члены). Это значит, что нельзя сказать, относится ли имя члена к типу или нет. По техническим причинам, связанным с особенностями языка программирования, компилятор должен знать это, поэтому мы ему должны каким-то образом передать эту информацию. Для этого используется ключевое слово **typename**.

```
template< class T> struct Vec {
  typedef T valuetype; //  имя члена класса
  static int count;          //  член - данное
};
template< class T> void my_func (Vec<T>& v){
   int x = Vec<T>::count;   // имена членов по умолчанию
                            // считаются не относящимися к типу
   v.count = 7;             // более простой способ сослаться
                            // на член, не являющийся типом
  typename Vec<T>:: valuetype xx = x; // здесь нужно слово typename
}
```

Для желающих быстро научиться - проекты

На следующих слайдах – 15 проектов, простых! Кто желает, можете их сделать и прислать мне на почту до следующей лекции (суббота — 10.09 вечером).

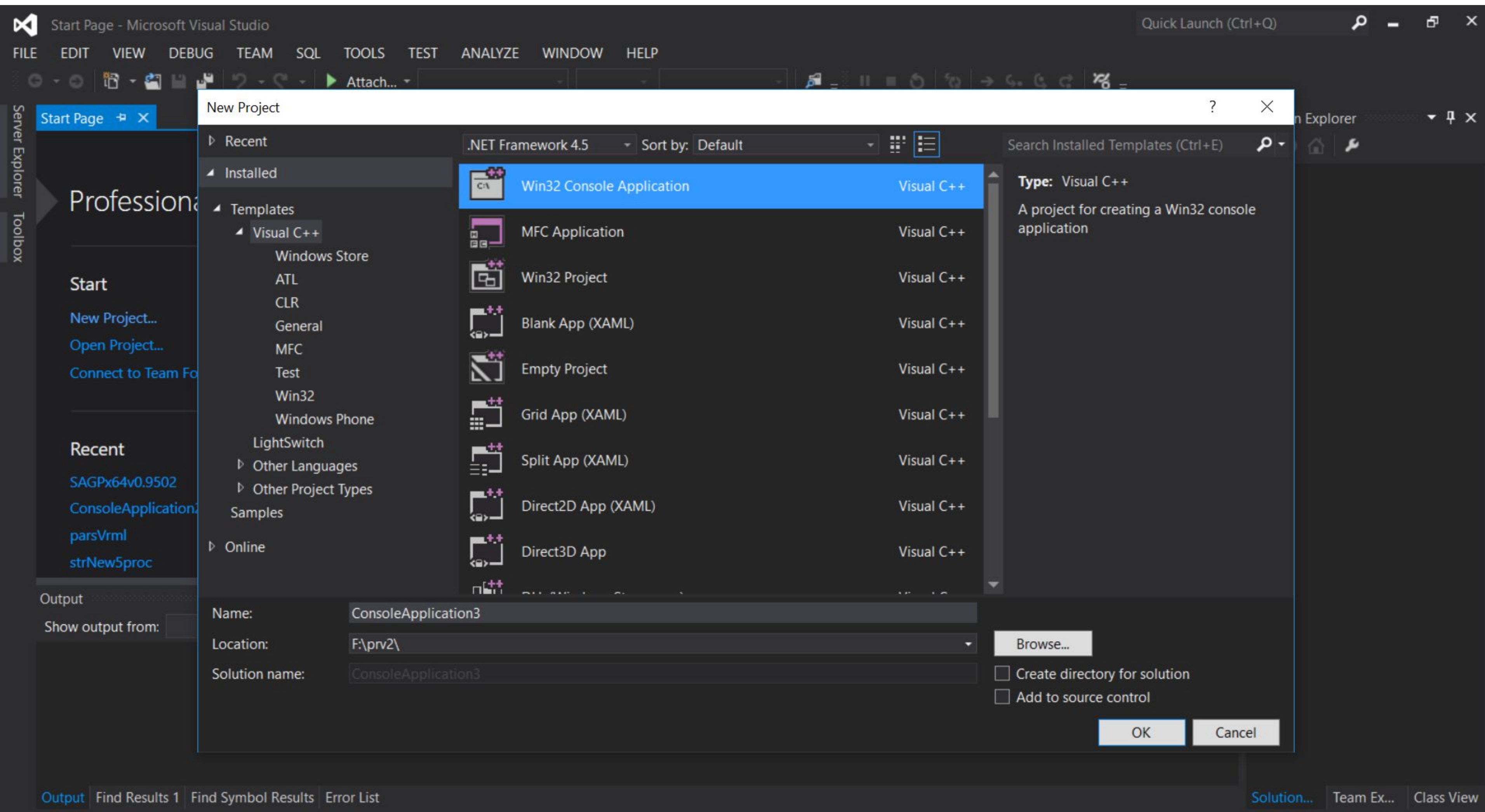
На каждой лекции я буду давать по подобному домашнему заданию.

Кто станет выполнять задания и присылать их результаты регулярно мне – получит на экзамене автоматом отличную оценку.

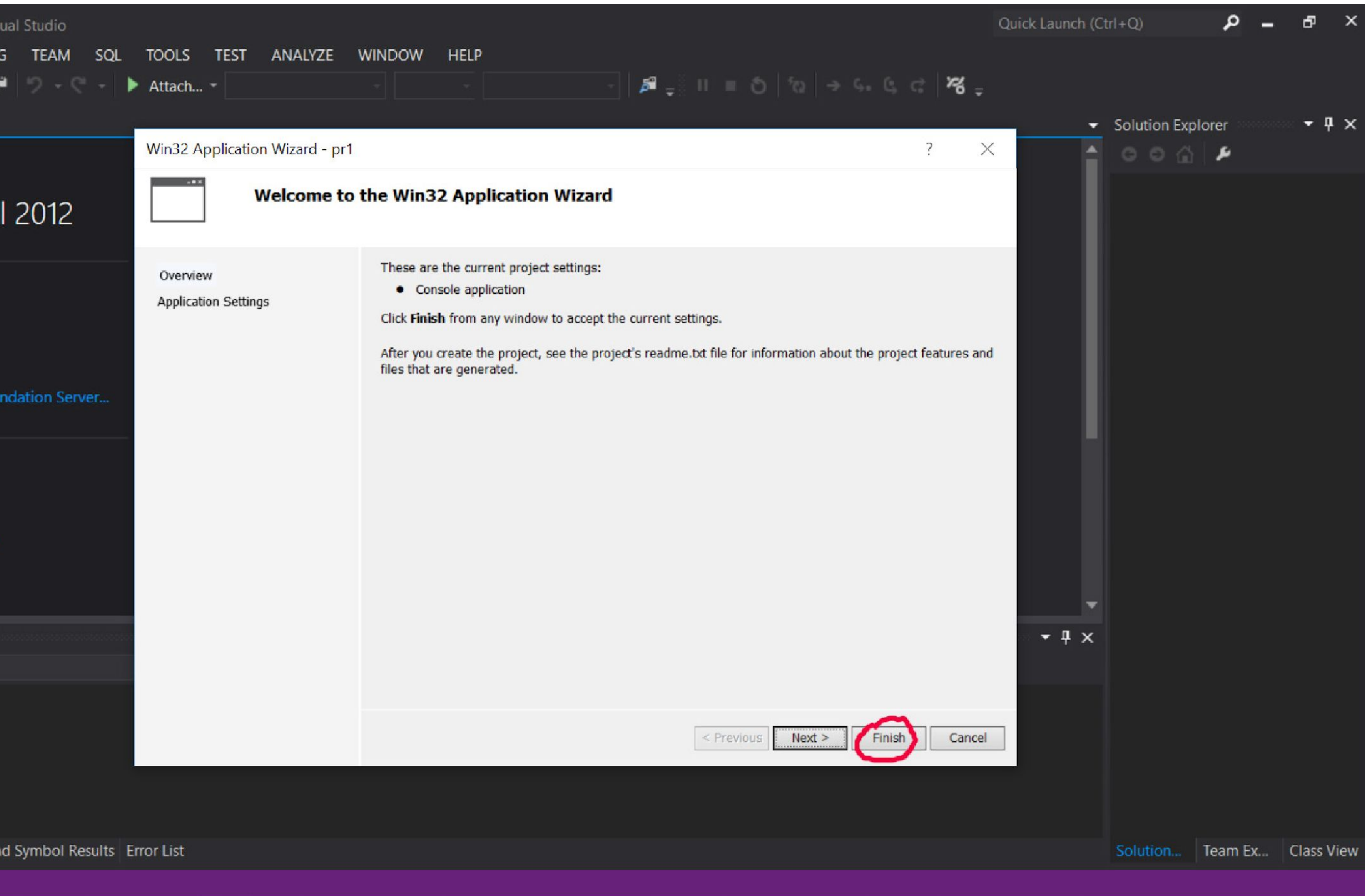
Кроме того, можете приходить ко мне заниматься индивидуально, только предварительно надо договориться – шлите письмо.

Также можете присылать любые вопросы.

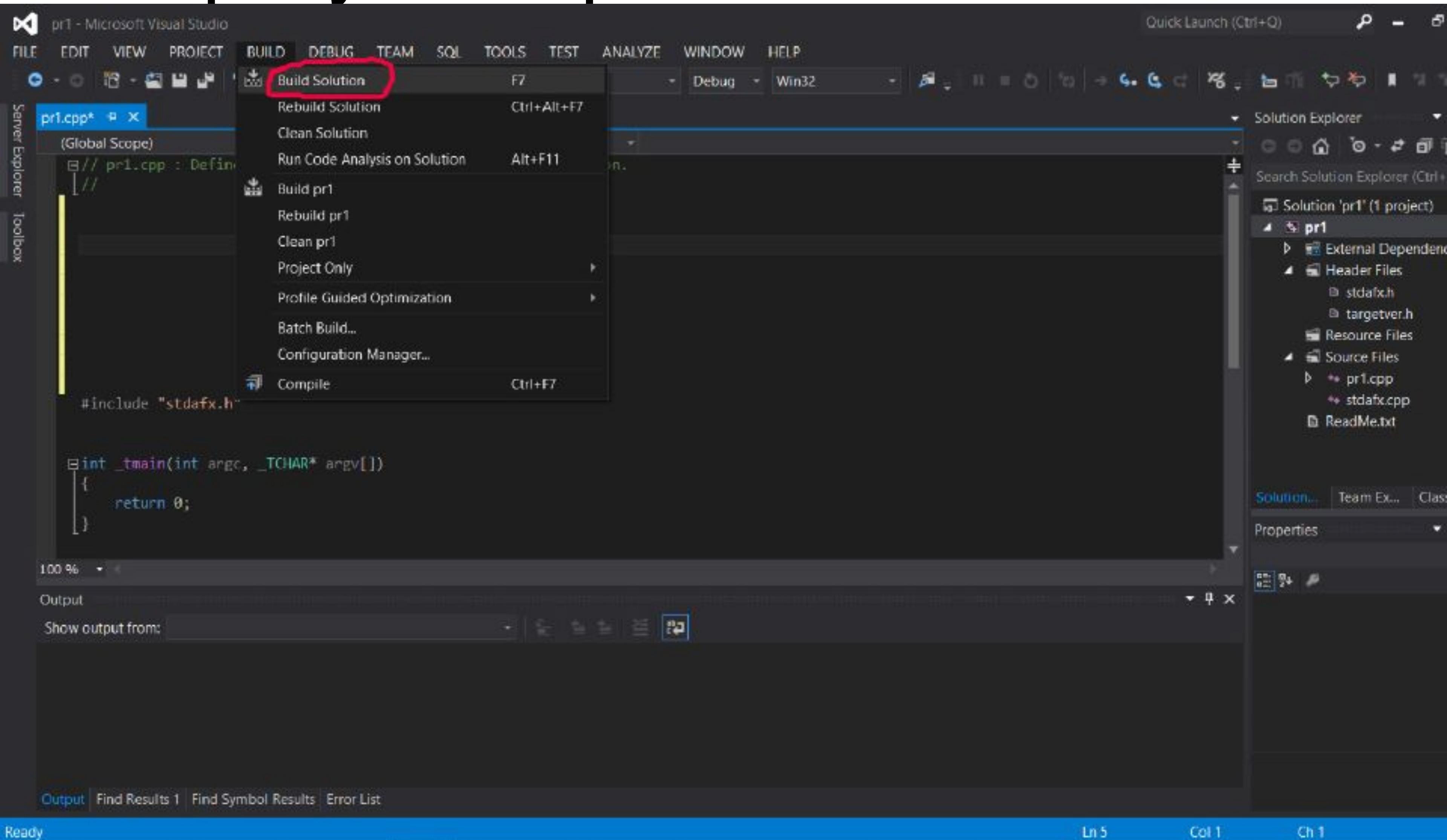
Домашнее задание. Создаем проект 1



Сразу как задали имя проекта выбираем кнопку Finish



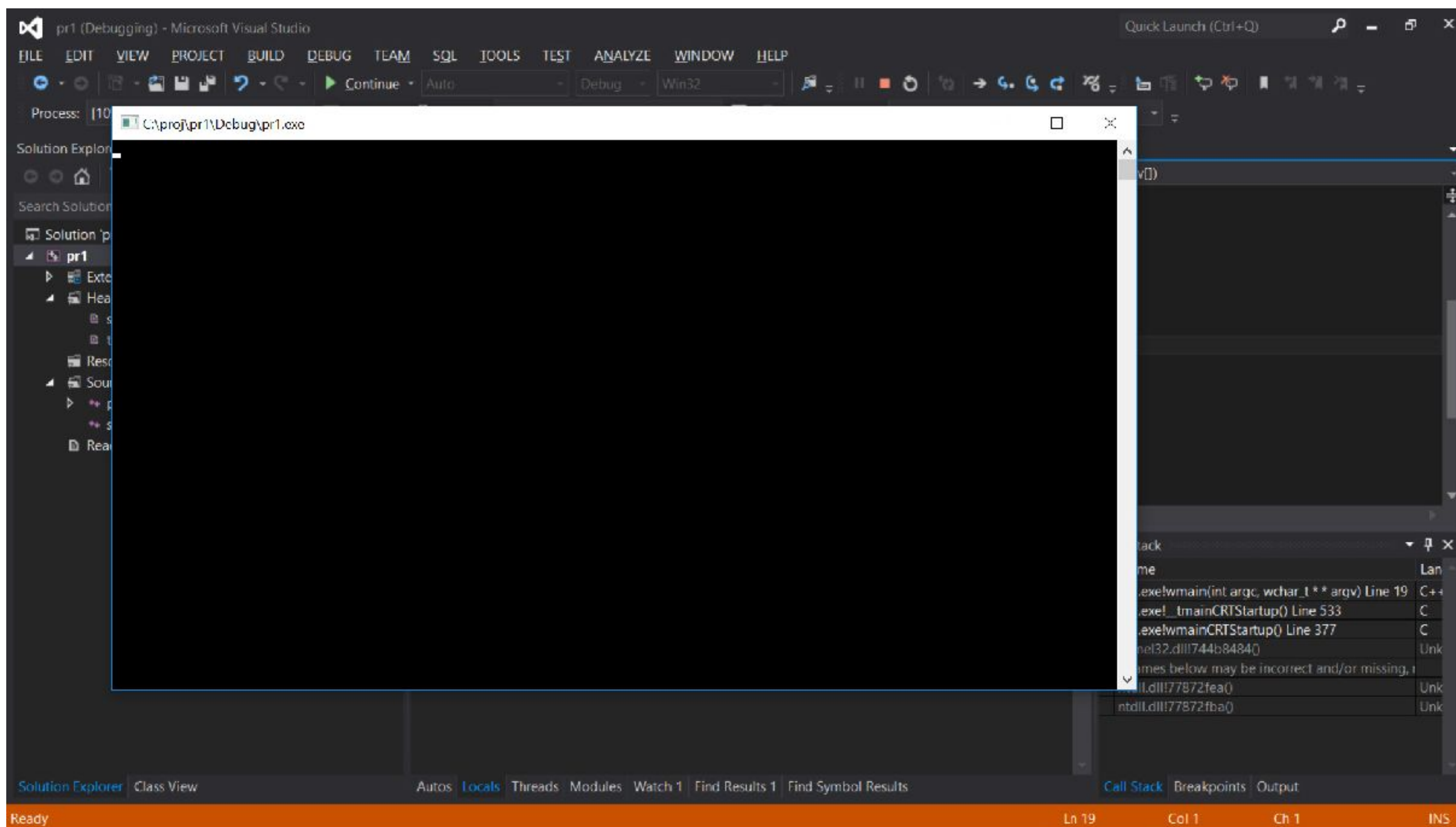
Ничего не делаем, сразу выбираем Build Solution



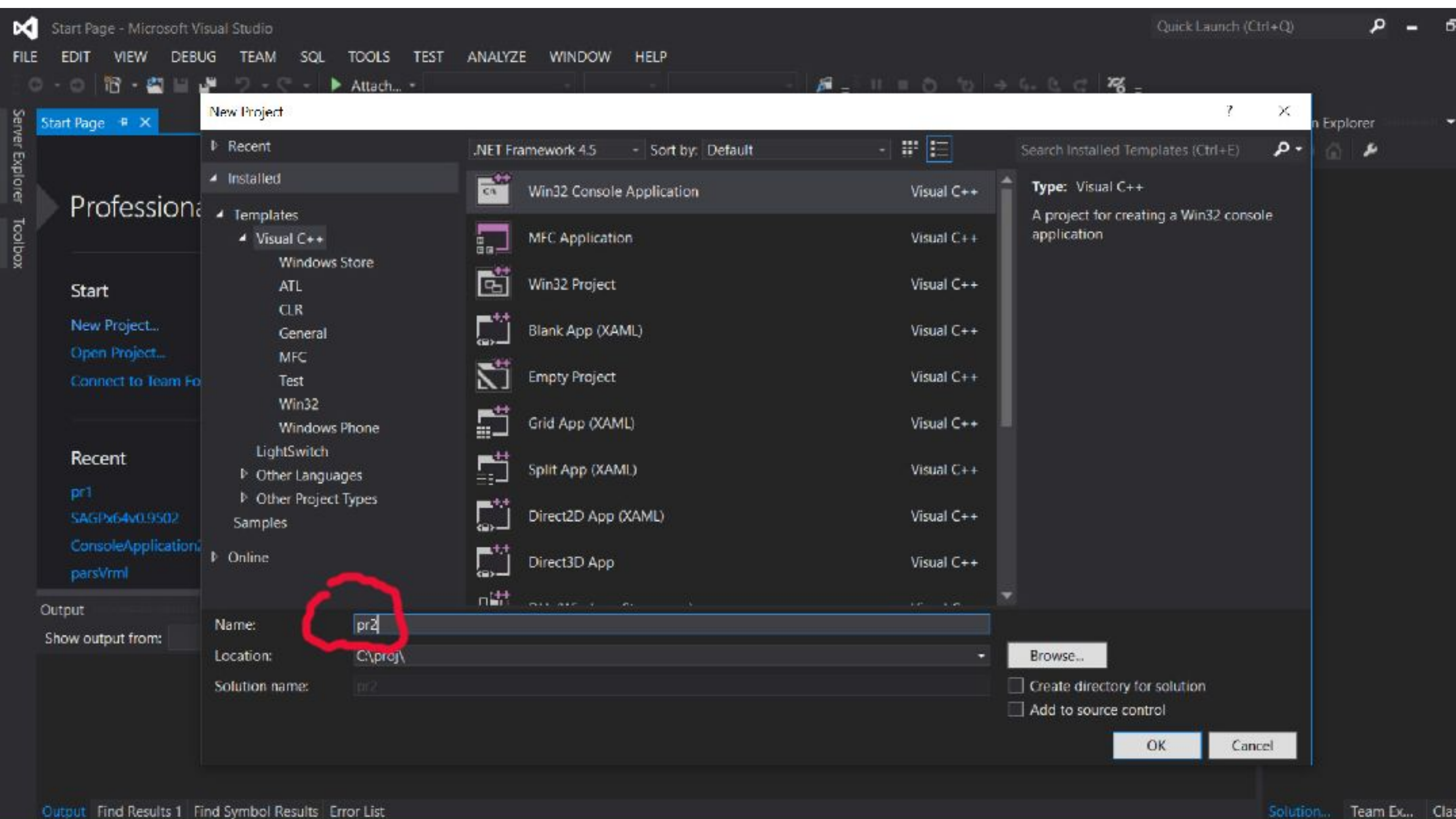
В свойствах проекта меняем Unicode на Multi Byte

General	
Output Directory	$(SolutionDir)$(Configuration)\
Intermediate Directory	$(Configuration)\
Target Name	$(ProjectName)
Target Extension	.exe
Extensions to Delete on Clean	*.cdf;*.cache;*.obj;*.ilk;*.resources;*.tlb;*.tli;*.tlh;*.tmp;*.rsp;*.pgc;*.pgd;*.m
Build Log File	$(IntDir)\$(MSBuildProjectName).log
Platform Toolset	**Visual Studio 2012 (v110)**
Enable Managed Incremental Build	No
Project Defaults	
Configuration Type	**Application (.exe)**
Use of MFC	Use Standard Windows Libraries
Use of ATL	Not Using ATL
Character Set	**Use Multi-Byte Character Set**
Common Language Runtime Support	Not Set
Whole Program Optimization	Use Unicode Character Set
Windows Store App Support	Use Multi-Byte Character Set
	<inherit from parent or project defaults>

Запускаем исполнение программы (клавиша F5 или через меню) – если ошибок при создании программы нет, то:



Создаем проект 2



Добавляем заголовочные файлы, получаем код:

```
#include "stdafx.h"
#include<iostream>
#include<fstream>
#include<string>
#include<set>
#include<map>
#include<vector>
#include<algorithm>
#include<cmath>
using namespace std;
```

Кроме того, добавляем строки в главную функцию программы:

```
int _tmain( )
{
string s;
s= "test";
cout<<s<<endl;
 getchar();
return 0;
}
```

Затем запускаем построение программы, и затем саму программу

Опять, если нет ошибок, должно открыться черное окно, которое теперь не закроется, а выведет строку **test**. Закрыть окно можно нажав клавишу «Return»

Создаем проекты 3 и 4, в которые добавляем по одному новому элементу. Смотрим результат.

Новые элементы соответственно:

<u>для 3-го</u>

```
 ofstream of;  // создает объект-поток
of.open("myfile.txt");// создает файл
for(int i=0;i<10;i++){
 of<<i<<endl; // записывает в файл
}
// откройте файл и посмотрите внутрь


ifstream ifile;  // создает объект-поток
ifile.open("myfile.txt");// открывает файл
string s0,s1;
   for(; ; ){
   ifile>>s0 ; // читает из файла
   if(  ifile.eof()) break;
   s1+=s0+"\n";
   }   cout<<s1;
```

<u>для 4</u>

```
ifstream ifile;  // создает объект-поток
 ifile.open("myfile.txt");// открывает файл
 int a;
vector <int> v;
for(; ; ){
ifile>>a ;
 if(  ifile.eof()) break;
v.push_back(a); // добавляем в вектор
}
for(int i=0; i<v.size(); i++){
 cout<<v[i]<<endl; // вывод данных
}
```

Создаем проект 5, в который также добавляем новый элемент, меняющий поведение программы

```
 ifstream ifile;            // создает объект- файловый поток
ifile.open("myfile.txt"); // открывает файл
 int a;
vector <int> v;
for(; ; ){
   ifile>>a ;
 if(  ifile.eof()) break;
    v.push_back(a);
}

 sort(v.begin(),v.end());
 // sort(v.rbegin(),v.rend()); // раскомментируйте и сравните рез-т

for(int i=0; i<v.size(); i++){
 cout<<v[i]<<endl;
}
```

Создаем проект 6, в который также добавляем новый элемент

```
//   Обернем код классом:
class Reader{
vector <int> m_v;
public:
Reader(){ }
void read(const char* name);
void print();
};

void Reader::read(const char* name){
  ifstream ifile;
 ifile.open(name);
   int a;
 for(; ; ){
   ifile>>a ;
 if(  ifile.eof()) break;
    m_v.push_back(a);
 }
}
```

```
void Reader::print( ){
  for(int i=0; i<m_v.size(); i++){
    cout<<m_v[i]<<endl;
  }
}


int _tmain(int argc, _TCHAR* argv[])
{
Reader r;
r.read("myfile.txt");
r.print();

  getchar();

return 0;
}
```

Создаем проект 7, в который добавляем новый элемент – класс с деструктором

```
class Test{
public:
 Test(){
  cout<<"Construct" <<endl;
 }
 ~Test(){
  cout<<"Destruct" <<endl;
 }
};
```

```
int _tmain(int argc, _TCHAR* argv[])
{
vector <Test*>v(10);
for(int i=0;i<v.size();i++){
v[i]=new Test();
}

for(int i=0;i<v.size();i++){
delete v[i] ; v[i]=0;
}
return 0;
}
```

Можете попробовать закомментировать строку delete v[i] ; v[i]=0;
и увидите, что только на вас лежит забота о созданном объекте

Активно используем отладчик: клавиша F9 устанавливает точку остановки – в этот момент можно посмотреть, что внутри переменных

```
    if(  ifile.eof()) break;
   st.insert(a);
    }

      // sort(st.begin(),st.end());
        set <int>::iterator  it=st.begin();
      for( ; it!= st.end() ;++it){
        cout<< *it
        }

   }
```

st { size=10 }

[0]	0
[1]	1
[2]	2
[3]	3
[4]	4
[5]	5
[6]	6
[7]	7
[8]	8
[9]	9
[Raw View]	0x0073f820 {...}

100 %

Locals

Name		Type
st		std::se
ifile	00000000 <NULL> _Set_egptr=0x000	std::b
a		int
it		std::_T

Call Stack

Name
pr3.exe!func() Line 55
pr3.exe!main(int argc, char * * argv) Line 66
pr3.exe!_tmainCRTStartup() Line 536
pr3.exe!mainCRTStartup() Line 377
kernel32.dll!759184840

Контрольная работа 1

Создать полиморфную иерархию из двух классов — **базового** и **производного** . Названия им дать по своим имени и фамилии латиницей. Внутри разместить по одному члену данных типа int и по конструктору, в котором присвоить значение этому члену данных – в каждом классе свое. Также определить в класс деструкторы. В полиморфной иерархии деструктор базового класса должен быть – виртуальным.

В главной функции программы создать по одному объекту ваших типов. Затем поместить их в хранилище типа vector<T>, где T — тип данных хранения полиморфных объектов, он должен быть указательным

vector < Base* > v.

Помещать их в вектор так:

```
Base* b= new  Base ;
 v.push_back (b); // или безымянный: v.push_back (new  Base );
```

В конце главной функции – освободить память в цикле:

```
delete v[i];
```