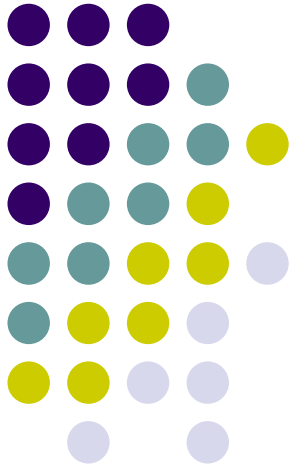
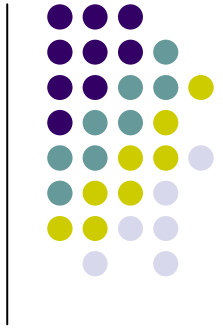
Программирование и алгоритмизация

Лекция 7

Различия между C, C++ и C#



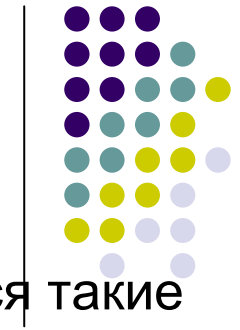


Различия между C и C++

1. Объекты

- В C++ появились классы и объекты. Технически класс C++ - это тип структуры в C, а объект - переменная такого типа. Разница только в том, что в C++ есть еще модификаторы доступа и полями могут быть не только данные, но и функции (функции-методы).
- Функция-метод - это обычная функция C, у которой первый параметр - это указатель на структуру, данные которой она обрабатывает: this. Если сравнить, как выглядят функции-методы в C++ и функции с параметром-указателем на структуру в C, то мы обнаружим, что всего лишь изменилась форма записи. В C++ получается короче, так как this и имя типа во многих случаях писать не обязательно (подразумевается по умолчанию).
- Модификаторы доступа - это слова public, private и protected. В C вместо них была внимательность программиста: public - значит с этими полями делаю, что хочу; private - значит к этим полям обращаюсь только с помощью методов этой структуры; protected - то же, что public, но еще можно обращаться из методов унаследованных структур (см. следующий пункт).

2. Наследование



- То, что в C++ - наследование, в C - это просто структура в структуре. При программировании в стиле C++ применяются такие красивые и звучные слова, как "класс Circle порожден от класса Point" или "класс Point наследуется от класса Circle и является производным от него". На практике все это словоблудие заключается в том, что структура Point - это первое поле структуры Circle.
- При этом реальных усовершенствований два. Первое - поля Point считаются так же и полями Circle, в результате доступ к ним записывается короче, чем в C. Второе - в обоих структурах можно иметь функции-методы, у которых имена совпадают с точностью до имени структуры. Например, Point::paint и Circle::paint . Следствие - не надо изобретать имена вроде Point_paint и Circle_paint, как это было в C, а префиксы Point:: и Circle:: в большинстве случаев можно опускать.

3. new и delete

- В C++ появились две новые операции: new и delete. В первую очередь это - сокращения для распространенных вызовов функций malloc и free:
- В стиле C:

  Point *p = (Point*) malloc(sizeof(Point));

  free(p);
- В стиле C++:

  Point *p = new Point;

  delete p;
- При вызове new автоматически вызывается конструктор, а при вызове delete - деструктор (см. следующий пункт). Так что нововведение можно описать формулой:

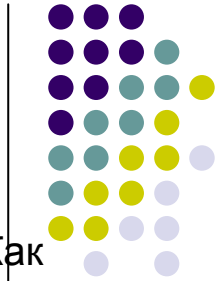
new = malloc + конструктор,

delete = free + деструктор.

4. Конструкторы и деструкторы

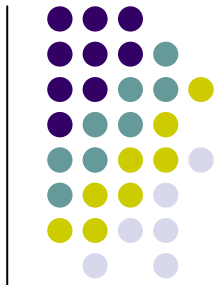
- Когда программируешь в стиле C, после того, как завел новую переменную типа структуры, часто надо ее инициализировать и об этом легко забыть. А перед тем как такая структура закончит свое существование, надо ее почистить, если там внутри есть ссылки на какие-то ресурсы. Опять-таки легко забыть.
- В C++ появились функции, которые вызываются автоматически после создания переменной структуры (конструкторы) и перед ее уничтожением (деструкторы). Во всех остальных отношениях это - обычные функции, на которые наложен ряд ограничений. Некоторые из этих ограничений ничем не оправданы и мешают: например, конструктор нельзя вызвать напрямую (дестркутор, к счастью, можно). Нельзя вернуть из конструктора или деструктора значение. Что особенно неприятно для конструктора. А деструктору нельзя задать параметры.

5. Виртуальные функции



- Из всех усовершенствований это вызывает наибольшую "щенячью радость". Как обычно, придуманы и звучно-научно-рекламные названия: "полиморфизм", "виртуальный", "абстрактный". Если отбросить разницу в терминологии, то что получим в сухом остатке? А получим мы очередное сокращение записи. И очень большое сокращение.
- При программировании на C часто бывает так, что имеется несколько вариантов одной и той же структуры, для которых есть аналогичные функции. Например, есть структура, описывающая точку (Point) и структура, описывающая окружность (Circle). Для них обоих часто приходится выполнять операцию рисования (paint). Так что, если у нас есть блок данных, где перемешаны точки, окружности и прочие графические примитивы, то перед нами стоит задача быстро вызвать для каждого из них свою функцию рисования.
- Обычное решение - построить таблицу соответствия "вариант структуры - функция". Затем берется очередной примитив, определяется его тип, и по таблице вызывается нужная функция. В C этот метод применять довольно нудно из-за того, что во-первых, надо строить эту таблицу, а во-вторых, внутри структур заводить поле, сигнализирующее о том, какого она типа, и следить за тем, чтобы это поле содержало правильное значение.
- В C++ всем этим занимается компилятор: достаточно обозначить функцию-метод как virtual, и для всех одноименных функций будет создана таблица и поле типа, за которыми следить будет опять-таки компилятор. Вам останется только пользоваться ими: при попытке вызвать функцию с таким именем, будет вызвана одна из серии одноименных функций в зависимости от типа структуры.

6. Исключения

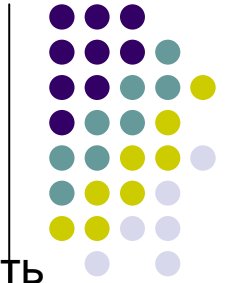


- Исключение по своей сути - это просто последовательность goto и return. Основан на обычной C-технологии setjmp/longjmp. try и catch - это setjmp с проверкой. throw - это longjmp. Когда вызывается throw, то проверяется: если он окажется внутри блока try, то выполняется goto на парный блок catch. Если нет, то делается return и ищется catch на уровень выше и так далее.
- Наличие в throw/catch параметра ничего принципиально не меняет: и в обычном C можно было заполнить какие-то переменные перед вызовом longjmp и потом их проанализировать.

7. Перегруженные операторы

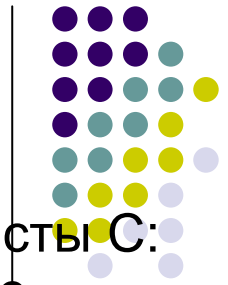
- Относитесь к ним как к уродливым функциям и все станет ясно. a + b, где a и b - типа Point это функция от двух аргументов a и b, возвращающая Point:
- Point operator+(Point a, Point b) Написать a+b равносильно вызову такой функции: operator+(a,b). Иногда эта технология удобна, а иногда вносит путаницу.

8. Ссылки



- Многие программисты изучали C на основе языка Pascal. В Pascal есть возможность возвращать из функции больше одного параметра. Для этого применялось магическое слово "var". В C для того, чтобы сделать то же самое, приходилось расставлять в тексте уйму символов "*".
- Разработчики C++ вняли стонам несчастных программистов и ввели слово var. А чтобы все это выглядел оригинально, "var" они переименовали в "&" и назвали "ссылкой". Это вызвало большую путаницу, так как в C уже были понятия "указатель" (та самая звездочка) и "адрес" (обозначался тем же символом &), а понятие "ссылка" звучит тоже как что-то указующе-адресующее. Вот если бы, не мудрствуя лукаво, добавили слово var…
- С одной стороны, использование ссылок намного сокращает текст программы. Но есть и неприятности. Во-первых, вызов функции, в которой параметр является ссылкой, выглядит так же, как вызов с обычным параметром. В результате "на глаз" незаметно, что параметр может измениться. А в C это заметно по значку &. Во-вторых, многочисленные звездочки в C напоминают программисту о том, что каждый раз выполняется дополнительная операция * разыменования указателя. Что побуждает сделать разумную оптимизацию. В C++ эти операции остаются незамеченными.

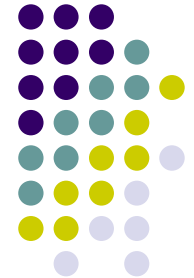
9. Inline, template и default-аргумент



- Аргумент по-умолчанию - это то, о чем мечтали программисты С: чтобы иногда не надо было при вызове задавать некоторые параметры, которые в этом случае должны иметь некоторое "обычное" значение.
- Желание программистов С контролировать типы параметров в define-ах породило в С++ inline-функции. Такая функция - это обычный define с параметрами, только не надо мучиться с символами "\" и проверяются типы.
- Желание узаконить в параметрах define имя типа породило template. Главный плюс template - то, что #define с одинаковыми параметрами породит два одинаковых куска кода. А template в компиляторе скорее всего будет соптимизирован: одинаковые куски кода будут соединены в один. Имеется небольшой контроль типов по сравнению с #define, но не очень удобный.

- В то же время template имеют ряд серьезных недостатков. Первый - ужасный, неудобный синтаксис. Чтобы это ощутить, достаточно попробовать. Уж лучше бы разрешили не контролировать типы некоторых параметров inline-функций. Второй недостаток - template остался так же неудобен при работе с отладчиком, как и #define.
- Ну и последнее нововведение, продиктованное, видимо, все тем же стремлением избавиться от #define. Это - тип "имя поля" (pointer to member). В C удобно было применять имена полей структур в define. В C++ тоже самое можно сделать с помощью операторов ::*, .* и ->*. Например &Circle::radius - это имя поля radius структуры Circle, а Circle::*radius - соответствующий тип. Такую величину можно передать, как параметр. Фактически речь идет о смещении этого поля относительно начала структуры. Бывает полезно. Примерно так же можно передать адрес функции-метода.

Примечание 1



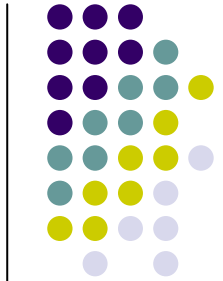
Директива #define определяет идентификатор и последовательность символов, которая будет подставляться вместо идентификатора каждый раз, когда он встретится в исходном файле. Идентификатор называется *именем макроса*, а сам процесс замены — *макрозаменой.* В общем виде директива выглядит таким образом:

#define *имя_макроса последовательность_символов*

Примеры:

```
#define ONE 1
#define TWO ONE+ONE
#define THREE ONE+TWO
#define E_MS "стандартная ошибка при вводе\n" /* ... */ printf(E_MS);
```

Примечание 2

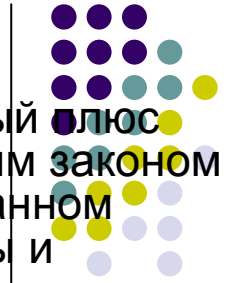


Пример template

```
template <typename T>
void printArray(const T * array, int count)
{
    for (int ix = 0; ix < count; ix++)
        cout << array[ix] << "   ";
    cout << endl;
} // конец шаблона функции printArray

    float  fArray[fSize] = {1.34, 2.37, 3.23, 4.8, 5.879, 6.345, 73.434, 8.82, 9.33, 10.4};
    char   cArray[cSize] = {"MARS"};
    cout << "\t\t Шаблон функции вывода массива на экран\n\n";
    // вызов локальной версии функции printArray для типа int через шаблон
    cout << "\nМассив типа int:\n"; printArray(iArray, iSize);
    // вызов локальной версии функции printArray для типа double через шаблон
    cout << "\nМассив типа double:\n"; printArray(dArray, dSize);
    // вызов локальной версии функции printArray для типа float через шаблон
    cout << "\nМассив типа float:\n"; printArray(fArray, fSize);
    // вызов локальной версии функции printArray для типа char через шаблон
    cout << "\nМассив типа char:\n";printArray(cArray, cSize);
```

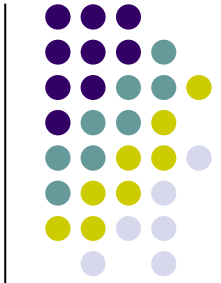
10. Язык более высокого уровня?

- Когда появились все эти нововведения, то многим стало видно то, что раньше было видно не столь многим. Это нормально: код упростился, а значит те его свойства, которые раньше были замаскированы лишними символами, стали заметны при меньшем напряжении мозгов. В этом и только в этом заключается более высокий уровень C++. Чуть больше абстракции, чуть меньше деталей - можно сосредоточится на более крупных блоках.
- Существует мнение, что писать в стиле C на C++ - дурной стиль. Это мнение - всего лишь дань моде. Если в стиле C++ получается короче, лучше, надежнее, то глупо писать в стиле C. Это так, но верно и обратное!
- Простой пример: у вас есть большой массив из 100 тысяч структур Point, который инициализируется один раз (все поля на 0) и много раз копируется в такие же массивы. Писать для элемента такого массива конструктор оказывается накладно. При его создании будет вызван конструктор для каждого элемента. Потом вы создадите массив, куда его надо копировать - и снова вызовы конструкторов. Затем вы выполняете копирование и затираете результаты второй инициализации. Мало того, что 100 тысяч вызовов конструктора просто не сопоставимы с одним вызовом memset, но эта серия вызовов будет повторяться не один раз, а много.



- Такие примеры можно привести для каждого нововведения C++. Каждый плюс неизбежно тянет за собой минусы. Для хорошего программиста главным законом должна быть не мода, а конечный результат и трезвый расчет: что в данном конкретном случае выгоднее с точки зрения эффективности программы и времени, затраченного на ее разработку.
- Что касается объектно-ориентированного программирования, то на самом деле оно не имеет отношения к разнице между C и C++. Благодаря ряду усовершенствований, код на C++ компактнее и надежнее, чем на C. Часть этих усовершенствований связана с ООП, а часть - нет. Например, аргументы функций по умолчанию и inline-функции к ООП не имеют никакого отношения. Они имеют отношение к ужесточению контроля типов.
- ООП - это просто идея: "в зависимости от данных, выполнить процедуру". А ПОП (процедурно ориентированное программирование) - "в зависимости от процедуры изменить данные". Глупо молиться на ООП или на ПОП или отвергать что-то из них и тем более ужасаться при их смешивании. Разумно использовать тот и другое, смотря как будет точнее, проще, быстрее, компактнее.
- Не уподобляйтесь консерватору, который говорит: "Я назло не буду использовать ООП, так как это - глупая новомодная штучка." Такой консерватор обычно упрямо применяет только C и при этом не замечает, что давно пишет в стиле ООП, но на чистом C. Он думает, что раз он использует C, его никто не заподозрит в излишнем умничаньи.
- Не уподобляйтесь моднику, который говорит: "Я буду использовать ООП везде, так как хочу прослыть прогрессивным человеком, который быстро осваивает все новое!" Такой "передовик" упрямо применяет классы и template где надо и где не надо. Он громогласно вопит об ООП, но сколько-нибудь сложная часть его кода обычно написана в стиле ПОП: потому, что он ценит ООП только как признак прогрессивности, но не понимает простого смысла, заключенного в нем.

Примечание



Функция memset

#include <string.h>

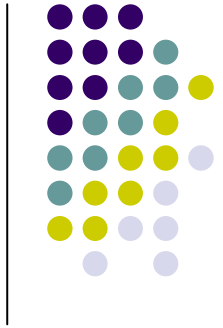
void *memset(void **buf*, int *ch*, size_t *count*);

Функция memset() копирует младший байт параметра *ch* в первые *count* символов массива, адресуемого параметром *buf*. Функция возвращает значение указателя *buf*.

Чаще всего функция memset() используется для инициализации области памяти некоторым известным значением.

Пример:

memset(buf, '\0', 100); memset(buf, 'X', 10); printf(buf);



Основные различия между C# и C++

1.Вместо включаемых header-файлов(.h) используются сборки(.dll), которые необходимо добавить в проект, а затем подключить пространство имен с помощью ключевого слова using, #include тоже можно. Системное окружение считается фиксированным (в отличие от других систем, где фиксированным является только системный интерфейс), вся системная часть как бы include'ится по умолчанию.

2. В Си при доступе к полям через объект используется точка, а через указатель стрелочка. В java и C# в обоих случаях используется точка.

3.Операция области видимости в C++ ::

в C# (.)

4.Введен оператор foreach, который повторяет группу вложенных операторов для каждого элемента массива или коллекции объектов, реализующих интерфейс System.Collections.IEnumerable или System.Collections.Generic.IEnumerable<T>.

5. Делегаты в C# введены вместо указателей на функции в C++.

Примечание 1

- **Пространство имён** (англ. *namespace*) — некоторое множество, под которым подразумевается модель, абстрактное хранилище или окружение, созданное для логической группировки уникальных идентификаторов (то есть имён). Идентификатор, определенный в пространстве имён, ассоциируется с этим пространством. Один и тот же идентификатор может быть независимо определён в нескольких пространствах. Таким образом, значение, связанное с идентификатором, определённым в одном пространстве имён, может иметь (или не иметь) такое же значение, как и такой же идентификатор, определённый в другом пространстве. Языки с поддержкой пространств имён определяют правила, указывающие, к какому пространству имён принадлежит идентификатор (то есть его определение).
- Пример:

```
using namespace std;
```

Примечание 2: Пример с foreach

```
class ForEachTest
 { static void Main(string[] args)
 { int[] fibarray = new int[] { 0, 1, 1, 2, 3, 5, 8, 13 };
 foreach (int element in fibarray) { System.Console.WriteLine(element);
    } System.Console.WriteLine();
for (int i = 0; i < fibarray.Length; i++) {
    System.Console.WriteLine(fibarray[i]); }
    System.Console.WriteLine(); // You can maintain a count of the
    elements in the collection.
 int count = 0; foreach (int element in fibarray) { count += 1;
    System.Console.WriteLine("Element #{0}: {1}", count, element); }
 System.Console.WriteLine("Number of elements in the array: {0}",
    count); }
```

6. Неявно типизированные переменные

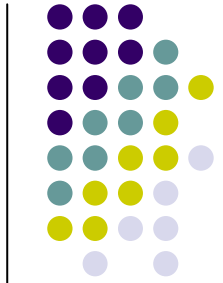
- Неявно типизированная переменная объявляется с помощью ключевого слова **var** и должна быть непременно инициализирована. Для определения типа этой переменной компилятору служит тип ее инициализатора, т.е. значения, которым она инициализируется:

```
var i = 12; // переменная i инициализируется целочисленным
    литералом
var d = 12.3; // переменная d инициализируется литералом с
    плавающей точкой,
                  // имеющему тип double
var f = 0.34F; // переменная f теперь имеет тип float
```

- Единственное отличие неявно типизированной переменной от обычной, явно типизированной переменной, — в способе определения ее типа. Как только этот тип будет определен, он закрепляется за переменной до конца ее существования.

7. Программа в исходном коде явно описывается как Class

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace ConsoleApplication1
{
    class Program
    {
        static void Main(string[] args)
        {
            Console.WriteLine("Привет!");
        }
    }
}
```

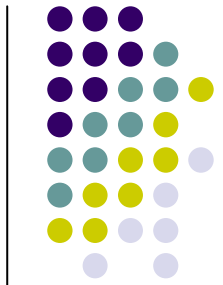


8. Любая переменная воспринимается как объект определенного класса и с ней работают как с объектом, обращаясь к функциям соответствующего класса

9. Существует переменная типа Type для описания типа переменных (объектов)

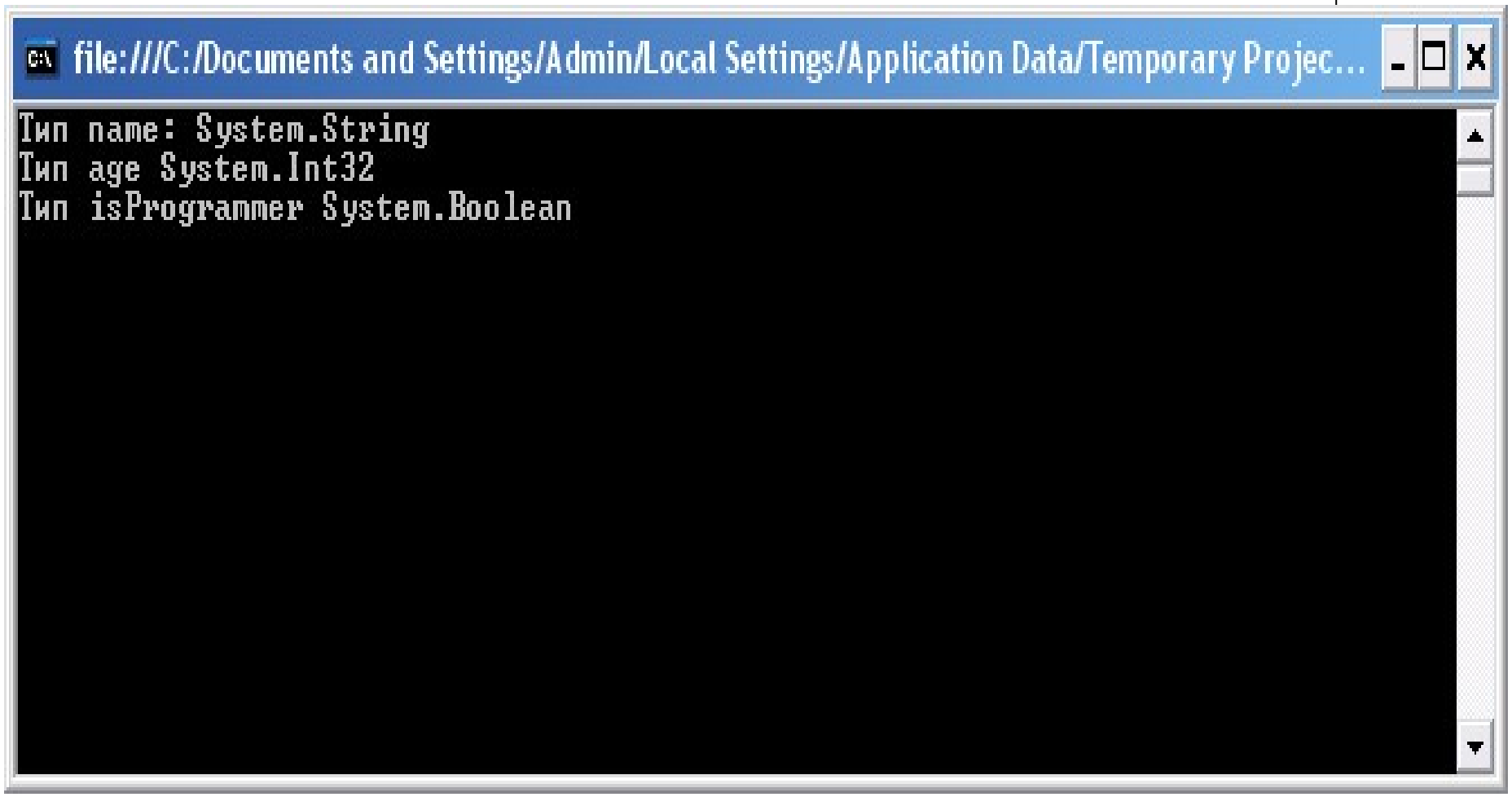
10. В консольном приложении существует класс Console и объект Console, используемые для ввода-вывода

Пример 1

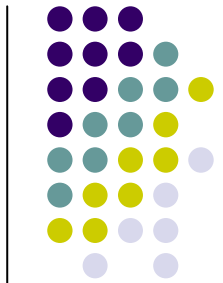


```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
namespace ConsoleApplication1
 {
  class Program
    {  static void Main(string[] args)
          {
              var name = "Alex Erohin";
              var age = 26;
              var isProgrammer = true; // Определяем тип переменных
              Type nameType = name.GetType();
              Type ageType = age.GetType();
              Type isProgrammerType = isProgrammer.GetType(); // Выводим результаты
              Console.WriteLine("Тип name: {0}",nameType);
              Console.WriteLine("Тип age {0}",ageType);
              Console.WriteLine("Тип isProgrammer {0}",isProgrammerType);
              Console.ReadLine();
          }
     }
 }
```

Результат


file:///C:/Documents and Settings/Admin/Local Settings/Application Data/Temporary Projec...
Тип name: System.String
Тип age System.Int32
Тип isProgrammer System.Boolean

Пример 2



```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
namespace ConsoleApplication5
{
  class Program
    {
      static void Main(string[] args)
       {
            Console.WriteLine("input - int double string:");
            int i = Convert.ToInt32(Console.ReadLine());
            double d = Convert.ToDouble(Console.ReadLine());
            string s = Console.ReadLine();
            Console.WriteLine("i = " + i + " d = " + d + " s = " + s);
            Console.WriteLine("i = {0} d = {1} s = {2}", i, d, s);
            Console.Read();
       }
    }
}
```

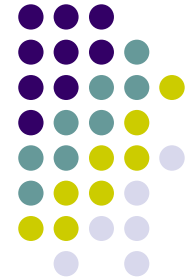
11. C# спроектирован и разработан специально для применения с .NET Framework

- ***Назначение .NET Framework*** — служить средой для поддержки разработки и выполнения сильно распределенных компонентных приложений. Она обеспечивает совместное использование разных языков программирования, а также безопасность, переносимость программ и общую модель программирования для платформы Windows.
-

Базовые функциональные возможности платформы .NET включают в себя:

- **Возможность обеспечения взаимодействия с существующим программным кодом**
  - Эта возможность, несомненно, является очень хорошей вещью, поскольку позволяет комбинировать существующие двоичные единицы COM (т.е. обеспечивать их взаимодействие) с более новыми двоичными единицами .NET и наоборот. С выходом версии .NET 4.0 эта возможность стала выглядеть даже еще проще, благодаря добавлению ключевого слова dynamic.
- **Поддержка для многочисленных языков программирования**
  - Приложения .NET можно создавать с помощью любого множества языков программирования (C#, Visual Basic. F#. S# и т.д.). При этом в .NET код, написанный на любом языке компилируется в код на промежуточном языке (Intermediate Language - IL).

Базовые функциональные возможности (2)

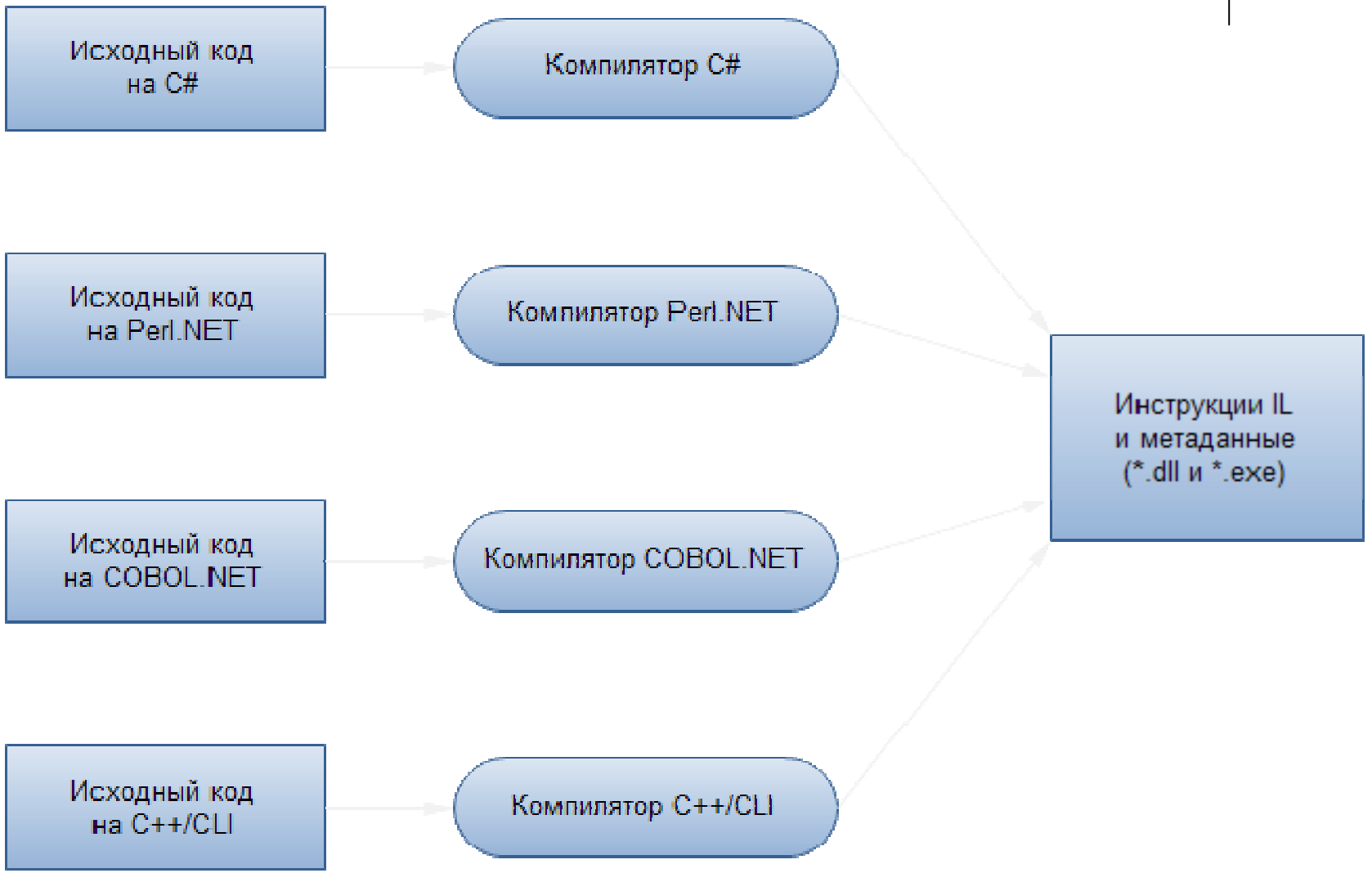


- **Полная интеграция языков**
  - В .NET поддерживается межъязыковое наследование, межъязыковая обработка исключений и межъязыковая отладка кода. При этом .NET использует общий исполняющий механизм, основным аспектом которого является хорошо определенный набор типов, который способен понимать каждый, поддерживающий .NET язык.
  - Так же в .NET был полностью переделан способ разделения кода между приложениями за счет введения понятия сборки (assembly) вместо традиционных библиотек DLL. Сборки обладают формальными средствами для управления версиями и допускают одновременное существование рядом нескольких различных версий сборок.
- **Усовершенствованная поддержка для создания динамических веб-страниц**
  - Хотя в классической технологии ASP предлагалась довольно высокая степень гибкости, ее все равно не хватало из-за необходимости использования интерпретируемых сценарных языков, а отсутствие объектно-ориентированного дизайна часто приводило к получению довольно запутанного кода ASP. В .NET предлагается интегрированная поддержка для создания веб-страниц с помощью ASP.NET. В случае применения ASP.NET код создаваемых страниц поддается компиляции и может быть написан на любом поддерживающем .NET языке высокого уровня, например, C# или Visual Basic 2010. В новой версии .NET эта поддержка улучшилась еще больше, сделав возможным применение новейших технологий вроде Ajax и jQuery.

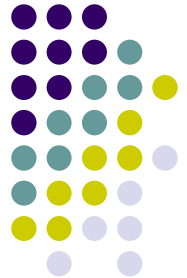
Базовые функциональные возможности (3)

- **Эффективный доступ к данным**
  - Набор компонентов .NET, известный под общим названием ADO.NET, позволяет получать эффективный доступ к реляционным базам данных и многим другим источникам данных. Также предлагаются компоненты, позволяющие получать доступ к файловой системе и каталогам. В частности, в .NET встроена поддержка XML, позволяющая манипулировать данными, импортируемыми и экспортируемыми на платформы, отличные от Windows.
- **Установка с нулевым воздействием**
  - Сборки бывают двух типов: разделяемые и приватные. Разделяемые сборки представляют собой обычные библиотеки, доступные всему программному обеспечению, а приватные сборки предназначены для использования только с определенными программами. Приватные сборки являются полностью самодостаточными, поэтому процесс их установки выглядит просто. Никакие записи в системный реестр не добавляются; все нужные файлы просто размещаются в соответствующей папке файловой системы.
- **Visual Studio 2010**
  - Вместе с .NET поставляется среда разработки Visual Studio 2010, которая способна одинаково хорошо справляться как с кодом на языке C++, C# и Visual Basic 2010, так и с кодом ASP.NET или XML. В Visual Studio 2010 интегрированы все наилучшие возможности сред конкретных языков из всех предыдущих версий этой IDE-среды.

Сборки

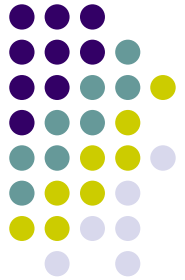

Исходный код на C#
Компилятор C#
Исходный код на Perl.NET
Компилятор Perl.NET
Исходный код на COBOL.NET
Компилятор COBOL.NET
Исходный код на C++/CLI
Компилятор C++/CLI
Инструкции IL и метаданные (*.dll и *.exe)

12. Класс Object



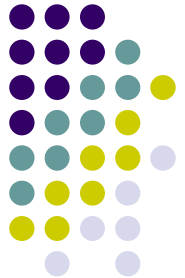
- В C# предусмотрен специальный **класс object**, который неявно считается базовым классом для всех остальных классов и типов, включая и типы значений. Иными словами, все остальные типы являются производными от object. Это, в частности, означает, что переменная ссылочного типа object может ссылаться на объект любого другого типа. Кроме того, переменная типа object может ссылаться на любой массив, поскольку в C# массивы реализуются как объекты. Формально имя object считается в C# еще одним обозначением класса System.Object, входящего в библиотеку классов для среды .NET Framework.
- Практическое значение этого в том, что помимо методов и свойств, которые вы определяете, также появляется доступ к множеству общедоступных и защищенных методов-членов, которые определены в классе Object. Эти методы присутствуют во всех определяемых классах.

Методы класса Object



- ToString()
- GetHashCode()
- Equals() и ReferenceEquals()
- Finalize()
- GetType()
- Clone()

Пример 3

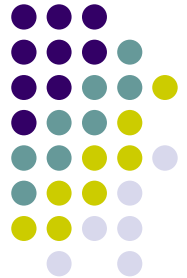


```
using System;
 using System.Collections.Generic;
 using System.Linq;
 using System.Text;
 namespace ConsoleApplication1
   {
     class Program { static void Main(string[] args)
       {
         var m = Environment.Version;
         Console.WriteLine("Тип m: "+m.GetType());
         string s = m.ToString();
         Console.WriteLine("Моя версия .NET Framework: " + s);
         Version v = (Version)m.Clone();
         Console.WriteLine("Значение переменной v: "+v);
         Console.ReadLine();
       }
    }
 }
```

Результат:


file:///C:/Documents and Settings/Admin/Local Settings/Application Data/Temporary Projec...
Тип m: System.Version
Моя версия .NET Framework: 4.0.30319.1
Значение переменной v: 4.0.30319.1

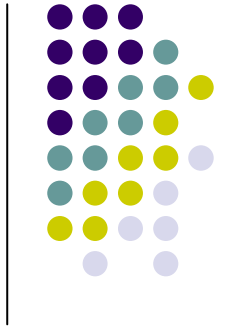
Пример 4



```
using System;
 using System.Collections.Generic;
 using System.Linq; using System.Text;
 namespace ConsoleApplication1
  {
    class Program { static void Main(string[] args)
     {
       var myOS = Environment.OSVersion;
       object[] myArr = { "Строка", 120, 0.345m, 2.34f, myOS, 'Z' };
       foreach (object obj in myArr)
            Console.WriteLine("Элемент \"{0}\" его тип - {1}",obj,obj.GetType());
       Console.ReadLine();
      }
   }
}
```

Результат:


file:///C:/Documents and Settings/Admin/Local Settings/Application Data/Temporary Projec...
Элемент "Строка" его тип – System.String
Элемент "120" его тип – System.Int32
Элемент "0,345" его тип – System.Decimal
Элемент "2,34" его тип – System.Single
Элемент "Microsoft Windows NT 5.1.2600 Service Pack 3" его тип – System.Operatin
gSystem
Элемент "Z" его тип – System.Char



Пример консольного приложения

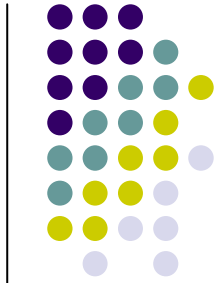
Исходный код на C++Builder

```
#include <stdio.h>
#include <conio.h>
#pragma hdrstop


#pragma argsused
int main(int argc, char* argv[])
{
        printf("Привет!");
        getch();
        return 0;
}
```

Исходный код на Visual C++

```
#include "stdafx.h"

using namespace System;

int main(array<System::String ^> ^args)
{
    Console::WriteLine("Привет!");
    return 0;
}
```

Исходный код на C#

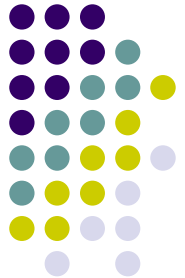


```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace ConsoleApplication1
{
    class Program
    {
        static void Main(string[] args)
        {
            Console.WriteLine("Привет!");
        }
    }
}
```

Пример оконного приложения

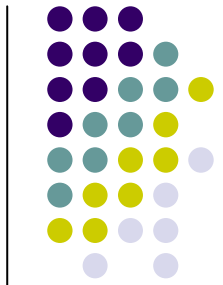
При нажатии на кнопку button1 вместо label1 выводится текст «Привет!»

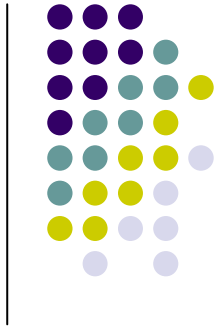
Исходный код на C++Builder.
Файл *.cpp, на следующем слайде *.h



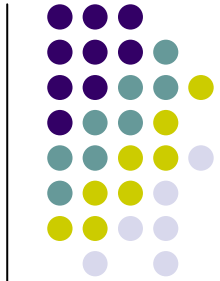
```
#include <vcl.h>
#pragma hdrstop

#include "Unit1.h"
//---------------------------------------------------------------------
#pragma package(smart_init)
#pragma resource "*.dfm"
TForm1 *Form1;
//---------------------------------------------------------------------
__fastcall TForm1::TForm1(TComponent* Owner)
        : TForm(Owner)
{
}
//---------------------------------------------------------------------

void __fastcall TForm1::Button1Click(TObject *Sender)
{
    this->Label1->Caption="Привет!";
}
```

```
#ifndef Unit1H
#define Unit1H
//---------------------------------------------------------------------------
#include <Classes.hpp>
#include <Controls.hpp>
#include <StdCtrls.hpp>
#include <Forms.hpp>
#include <ExtCtrls.hpp>
#include <jpeg.hpp>
//---------------------------------------------------------------------------
class TForm1 : public TForm
{
__published:        // IDE-managed Components
        TLabel *Label1;
        TButton *Button1;
        TImage *Image1;
        void __fastcall Button1Click(TObject *Sender);
private:   // User declarations
public:                 // User declarations
        __fastcall TForm1(TComponent* Owner);
};
//---------------------------------------------------------------------------
extern PACKAGE TForm1 *Form1;
//---------------------------------------------------------------------------
#endif
```

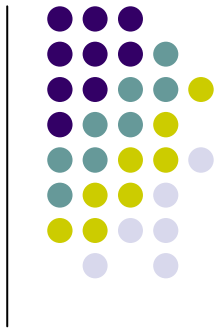




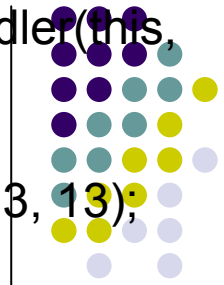
Исходный код на Visual C++



```
#pragma once


namespace test2 {
	using namespace System;
	using namespace System::ComponentModel;
	using namespace System::Collections;
	using namespace System::Windows::Forms;
	using namespace System::Data;
	using namespace System::Drawing;
	public ref class Form1 : public System::Windows::Forms::Form
	{
	public:
		Form1(void)
		{
			InitializeComponent();
		}
	protected:
		~Form1()
		{
			if (components)
			{
				delete components;
			}
		}
```



```
            private: System::Windows::Forms::Button^  button1;
            protected:
            private: System::Windows::Forms::Label^  label1;
            private: System::Windows::Forms::PictureBox^  pictureBox1;
            private:
            System::ComponentModel::Container ^components;
#pragma region Windows Form Designer generated code
                        void InitializeComponent(void)
                        {
            System::ComponentModel::ComponentResourceManager^  resources =
(gcnew System::ComponentModel::ComponentResourceManager(Form1::typeid));
                                    this->button1 = (gcnew System::Windows::Forms::Button());
                                    this->label1 = (gcnew System::Windows::Forms::Label());
                                    this->pictureBox1 = (gcnew
System::Windows::Forms::PictureBox());
                                    (cli::safe_cast<System::ComponentModel::ISupportInitialize^
>(this->pictureBox1))->BeginInit();
                                    this->SuspendLayout();
                                    this->button1->Location = System::Drawing::Point(161, 13);
                                    this->button1->Name = L"button1";
                                    this->button1->Size = System::Drawing::Size(75, 23);
                                    this->button1->TabIndex = 0;
                                    this->button1->Text = L"button1";
                                    this->button1->UseVisualStyleBackColor = true;
```

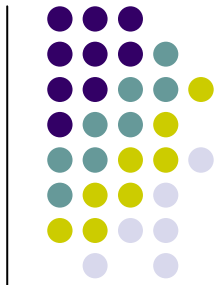


```
                this->button1->Click += gcnew System::EventHandler(this,
&Form1::button1_Click);
                this->label1->AutoSize = true;
                this->label1->Location = System::Drawing::Point(23, 13);
                this->label1->Name = L"label1";
                this->label1->Size = System::Drawing::Size(35, 13);
                this->label1->TabIndex = 1;
                this->label1->Text = L"label1";
                this->pictureBox1->Image =
(cli::safe_cast<System::Drawing::Image^  >(resources-
>GetObject(L"pictureBox1.Image")));
                this->pictureBox1->InitialImage =
(cli::safe_cast<System::Drawing::Image^  >(resources-
>GetObject(L"pictureBox1.InitialImage")));
                this->pictureBox1->Location = System::Drawing::Point(45,
42);
                this->pictureBox1->Name = L"pictureBox1";
                this->pictureBox1->Size = System::Drawing::Size(191, 212);
                this->pictureBox1->SizeMode =
System::Windows::Forms::PictureBoxSizeMode::StretchImage;
                this->pictureBox1->TabIndex = 2;
                this->pictureBox1->TabStop = false;
                this->pictureBox1->WaitOnLoad = true;
                this->AutoScaleDimensions = System::Drawing::SizeF(6,
13);
```

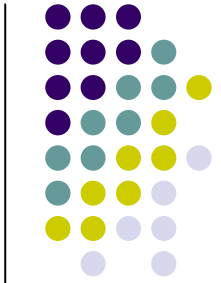
```
                this->AutoScaleMode =
System::Windows::Forms::AutoScaleMode::Font;
                this->ClientSize = System::Drawing::Size(292, 266);
                this->Controls->Add(this->pictureBox1);
                this->Controls->Add(this->label1);
                this->Controls->Add(this->button1);
                this->Name = L"Form1";
                this->Text = L"Form1";
        (cli::safe_cast<System::ComponentModel::ISupportInitialize^  >(this-
>pictureBox1))->EndInit();
                this->ResumeLayout(false);
                this->PerformLayout();
            }
#pragma endregion
        private: System::Void button1_Click(System::Object^  sender,
System::EventArgs^  e)
        {
            label1->Text="Привет!";     }
        };
}
```

Исходный код на C#

```
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Linq;
using System.Text;
using System.Windows.Forms;

namespace WindowsFormsApplication1
{
    public partial class Form1 : Form
    {
        public Form1()
        {
            InitializeComponent();
        }

        private void button1_Click(object sender, EventArgs
e)
        {
            label1.Text = "Привет!";
        }
    }
}
```



Ссылки:



http://professorweb.ru/my/csharp/charp_theory/level1/index.php