Лекции по C/C++: Классы, часть 3 (полиморфизм)

Полиморфизмом в общем смысле называется способность функции обрабатывать данные разных типов. Основные области применения полиморфизма в C++ следующие:

1. **Перегрузка функций**. C++ позволяет определять несколько функций с одинаковым именем в одном пространстве имён. Такие функции называются перегруженными и компилятор различает их по спискам параметров. Перегрузка позволяет использовать универсальные имена для выполнения однотипных действий.

В классическом Си поддержка полиморфизма ограничена, например, функция взятия модуля числа имеет разные названия в зависимости от типа своего аргумента – `abs`, `fabs` или `labs`. В C++ для совместимости эти имена сохранены, но можно и не помнить 3 различных названия функций, а пользоваться полиморфной функцией `abs`. Компилятор сам выберет, какая конкретная версия функции выполняется:

```
int n = abs(-3);
double m = abs(-4.5);
 //ошибочный результат 4 в C, корректный 4.5 в C++
```

Приведём пример пользовательской перегрузки функции вывода значения объекта с именем `print`:

```
void print( char *s ) { cout << endl << s; }

void print( double dvalue, int prec=0)  {
 cout << endl << fixed << setprecision(prec) << dvalue;
}

//...
 char s[80]; strcpy (s,"Hello, world!");
 double d = 3.1415926;
 print (s);
 print (d,2);
```

Для работы примера нужно подключить следующие стандартные заголовки:

```
#include <iostream>
#include <iomanip>
using namespace std;
```

Многие функции C++-библиотек, таких как .NET или QT, также имеют по несколько перегрузок, работающих с разными типами или разными списками аргументов.

2. **Виртуальные методы базовых классов**, которые переопределяются в производных классах. То есть, производные классы предоставляют свои собственные определения и реализацию некоего стандартного действия ("установить",

"нарисовать", "стереть" и т.п.). Во время выполнения, когда клиент вызывает виртуальный метод, выполняется поиск типа объекта и вызывается перезапись виртуального метода. В качестве примера приведём класс `Number` (число), имеющий потомков `IntNumber` (целое число) и `DoubleNumber` (вещественное число):

```
class Number {
protected:
 virtual void set() {}
 virtual void show() {}
};
class IntNumber : public Number {
 int n;
public:
 virtual void set(int n) { this->n = n; }
 virtual void show() { cout << endl << n; }
};
class DoubleNumber : public Number {
 double n;
public:
 virtual void set(double n) { this->n = n; }
 virtual void show() {
  cout << endl << fixed << setprecision(2) << this->n;
 }
};
//...
 IntNumber n; n.set(5); n.show();
 DoubleNumber d; d.set(2.5); d.show();
```

Как видно из кода, мы обращаемся с объектами разных классов (числами различных типов) похожим образом.

3. **Перегрузка операторов в классах**. Оператор в C++ - это некоторое действие или *функция*, обозначенная специальным символом. Для того чтобы распространять эти действия на новые типы данных, при этом сохраняя естественный синтаксис, в C++ была введена возможность перегрузки операторов.

Перегружать можно следующие операторы:

```
+       -       *       /       %       ^       &       |
~       !       ,       >=      <=      >       <       =
++      --      <<      >>      ==      !=      &&      ||
+=      -=      /=      %=      ^=      &=      |=      *=
<<=     >>=     [ ]     ( )     ->      ->*     new     delete
```

Соответственно, даже среди существующих операторов есть такие, которые нельзя перегрузить. Например, оператор разрешения области видимости `::` или оператор `.` (точка), используемый для обращения к полям и методам объекта.

Для встроенных типов данных перегружать операторы также нельзя, потому что тип — это набор данных и операций над ними. Изменяя операции, выполняемые над данными типа, мы меняем само определение типа. Также нельзя изменить приоритет существующего оператора или определить новый оператор, отсутствующий в языке.

Переопределяя операторы "`,`" "`&&`" "`||`" мы можем потерять их "ленивые" свойства (возможность не вычислять выражение до конца, например, `true || условие == true`, соответственно, после получения первого значения `true` условие перестаёт вычисляться).

Операторы "`->`", "`[]`", "`()`", "`=`" и "`(type)`" можно переопределить только как методы класса.

Перегрузка операторов бывает полезна в следующих типовых случаях:

- вместо синтаксиса `get(i)` удобнее использовать `[i]`;
- `a.plus(b)` или `plus(a,b)` удобнее заменить на `a + b`;
- вместо `p.get()->print()` удобнее использовать `p->print()`.

И т.д., например, в классе "матрица" удобно и естественно переопределить оператор "`*`" для умножения матриц, как сделано в Mathcad.

Следует понимать, что операторы делятся на *унарные* (применяемые к одному объекту, например, инкремент и декремент) и *бинарные* (имеющие два операнда, например, сложение или умножение). Единственную в C/C++ тернарную операцию условного оператора `(… ? … : …)` переопределить также нельзя.

4. **Использование шаблонов**. Шаблоны функций - это инструкции, согласно которым создаются локальные версии шаблонизированной функции для определенного набора параметров и типов данных. Например, без использования шаблонов для решения типовой задачи вывода в консоль элементов массива нам пришлось бы создать столько функций, сколько типов массивов мы обрабатываем:

```
void print(int *a, int n) {
 cout << endl;
 for (int i=0; i<n; i++) cout << a[i] << " ";
}
void print(double *a, int n) {
 cout << endl;
 for (int i=0; i<n; i++) cout << a[i] << " ";
}
//...
int a[] = {1,2,3}; print (a,3);
double b[] = { 1.5, 2.5 }; print(b,2);
```

Функции отличаются только типом параметров, такой подход крайне нерационален по затратам времени. С помощью ключевого слова `template` мы можем создать универсальный шаблон функции печати массива с любым типом элементов:

```
template <typename T>
void print(T *a, int n) {
 cout << endl;
 for (int i=0; i<n; i++) cout << a[i] << " ";
}
```

Компилятор сам создаст локальные копии функции-шаблона.

Все шаблоны функций начинаются со слова `template`, после которого идут угловые скобки, в которых перечисляется список параметров. Каждому параметру должно предшествовать зарезервированное слово `class` или `typename`:

```
template <class T>
```

или

```
template <typename T>
```

или

```
template <typename T1, typename T2>
```

Ключевое слово `typename` говорит о том, что в шаблоне будет использоваться встроенный тип данных, такой как `int`, `double`, `float`, `char` и т.д. А ключевое слово `class` сообщает компилятору, что в шаблоне функции в качестве параметра будут использоваться пользовательские типы данных, то есть, прежде всего, классы.

5. **Контейнеры из разнотипных объектов**. С помощью полиморфизма можно поместить объекты разных классов в один массив с типом базового класса. Например, создадим три класса с двумя методами – невиртуальный метод `Info` выводит информацию о животном, а виртуальный метод `Say` сообщает, что это животное "говорит". Виртуальный метод переопределён (override) в классах-наследниках. Невиртуальный метод просто скрыт в наследниках новой реализацией (невиртуальные методы нельзя переопределять).

```
class Animal {
public:
 void Info() { cout << "Animal" << endl; }
 virtual void Say() { cout << "Nothing to say" << endl; }
};

class Cat : public Animal {
public:
 void Info() { cout << "Cat" << endl; }
 virtual void Say() { cout << "Meow" << endl; }
};

class Dog : public Animal {
public:
 void Info() { cout << "Dog" << endl; }
 virtual void Say() { cout << "Bow-wow" << endl; }
};
```

При создании объекта будет иметь значение, в переменную какого типа записан объект:

```
Dog *dog1 = new Dog();
Animal *dog2 = new Dog();
 //Невиртуальный метод - вызовется метод класса, указанного у переменной
dog1->Info(); // напишет Dog
dog2->Info(); // напишет Animal
 //Виртуальный метод - вызовется метод класса, который в нём реализован
dog1->Say(); // напишет Bow-wow
dog2->Say(); // напишет Bow-wow
```

А теперь частая ситуация, когда полиморфизм нужен - при создании массива объектов:

```
Dog *dog = new Dog();
Cat *cat = new Cat();
Animal *animal = new Animal();
Animal *animals[3];
animals[0] = dog;
animals[1] = cat;
animals[2] = animal;
 //заполним массив указателей вперемешку разными животными
for (int i=0; i<3; i++) animals[i]->Say(); //и пусть каждое "скажет"
 //вызовется правильный метод
 //у невиртуальных методов так сделать нельзя! Полиморфизм в действии
```

Ключевым в понимании полиморфизма является то, что он позволяет вам манипулировать объектами различной степени сложности путём создания общего для них стандартного интерфейса для реализации похожих действий (принцип "**один интерфейс – множество методов**"). В целом полиморфизм позволяет писать более абстрактные программы и повысить степень повторного использования кода.

Пример 1. Теперь добавим к классу, созданному в предыдущей лекции (проект прикреплён внизу по ссылке), две основных возможности из перечисленных:

- создание списка, в который могут входить экземпляры как предка, так и потомка (потомков);
- переопределение операторов в рамках класса.

В описание класса `Student` из предыдущей лекции добавим следующие элементы (файл `student.h`):

```
#define MAXSIZE 30 /*до тела класса; максимальный размер списка */
```

Поля в разделе public:

```
static int count;
  //текущее количество элементов списка
static Student **students;
  //указатель на массив экземпляров, входящих в список
static Student *begin;
  //указатель на начало списка в памяти
```

Ключевое слово `static` в применении к свойству класса означает, что свойство существует в единственном числе для всех экземпляров класса (является статическим). Такие свойства можно использовать, например, как счётчики числа объектов или указатели на их списки.

Вне всех функций, то есть, глобально добавим в файл `student.cpp` инициализацию статических членов класса. Она выполняется именно таким образом, подобно инициализации глобальных переменных:

```
int Student::count=0;
const int size=MAXSIZE;
Student ** Student::students = new Student * [size];
Student * Student::begin = Student::students[0];
```

Переменная `size` здесь служебная и предназначена для инициализации максимального размера списка.

Опишем прототипы новых методов, также в разделе `public` базового класса (файл `student.h`):

```
static void print();
 //статический метод для печати текущего списка;
 //этот метод тоже имеет единственную точку входа для всех экземпляров
int add (void);
 //метод для добавления элемента в список
int search (Student *); //метод для поиска объекта Student в списке
 //(по совпадению значения свойства Name, впрочем, это как запрограммируем)
```

Функция `print`, как видно из листинга, также объявлена статической. В частности, это означает, что её можно вызывать *без* создания экземпляра класса кодом вида `Student::print();`

Наконец, предусмотрим в нашем классе переопределение операторов. Как сказано выше, переопределённый оператор понимается просто как перегруженная функция, а переопределять можно многие, но не все операторы C++.

При переопределении оператора его приоритет и порядок выполнения операций не меняются, но в ряде случаев действуют специальные правила написания переопределённой функции, например, чтобы отличать перегруженный постфиксный `++` от префиксного.

Общий вид переопределения оператора следующий:

```
ИмяТипа operator СимволОперации  (СписокПараметров)
```

Добавим в публичную секцию описания класса `Student` типичные примеры функций переопределения операторов:

```
int operator ! ();
```

Здесь мы переопределили унарный оператор `!` (отрицание). Его результат - величина типа `int`, которая будет принимать значения 1 (истина) или 0 (ложь). Это соответствует обычной логике данного оператора. Параметры такой функции не требуются, так как она будет работать с текущим объектом `this`.

```
void operator += (char *);
```

Прототип функции, переопределяющей оператор `+=`, который не будет создавать нового объекта, а лишь изменять текущий, "прицепляя" строку-параметр к свойству `Name` текущего объекта. Поэтому тип функции указываем `void`. Это не значит, что нельзя было реализовать функцию `+=`, возвращающую значение некоторого типа. В общем случае нужно помнить о вычислениях "по цепочке" и избегать операторов, которые ничего не возвращают. Например, наш оператор будет работать в синтаксисе

```
Student *s = new Student ("Popov",210);
*s += " I.A.";
```

но не

```
Student *t = new Student ("Smirnov",210);
Student s2 = (*t += " E.S.");
```

С другой стороны, ничто не мешает написать несколько реализаций функции-оператора, отличающихся списком параметров и возвращаемых значений, так же, как мы поступали с обычными функциями (писать перегруженные функции, которые отличаются *только* типом возвращаемого значения, нельзя).

```
Student operator + (Student &);
```

Переопределили бинарный оператор сложения, он получает ссылку на прибавляемый объект, стоящий справа от знака "+" (`Student &`) и возвращает новый объект, полученный в результате сложения. Объект слева от знака "+" доступен через `this`.

В ряде случаев экономичнее возвращать только ссылку на объект класса, а не сам объект (`Student &` вместо `Student`).

В публичной секции описания дочернего класса `Hobbit` также переопределим 2 операции - префиксный и постфиксный операторы `++`, правила их переопределения видны из кода:

```
Hobbit & operator ++ (); //префиксный
Hobbit operator ++ (int); //постфиксный
```

Напишем реализацию новых и изменённых свойств и методов для обоих классов.

Метод `add` добавляет объект родительского или дочернего класса в единый список (с контролем предельного заполнения списка):

```
int Student::add (void) {
 if (count<size) {
  students[count++]=this;
  //внесли в список указатель текущий объект и увеличили счётчик
  if (count<size) students[count] =NULL;
  //на всякий случай помечаем конец списка NULL
  return 1;
 }
 return 0; //не удалось добавить объект – список заполнен!
}
```

Удалять объект из динамического списка будет деструктор базового класса, соответственно, его реализация изменится:

```
Student::~Student () {
 int Found = search (this);
 if (Found!=-1) {
  for (int i=Found; i<count-1; i++) {
   students[i]=students[i+1];
  }
  students[count-1]=NULL;
  count--;
 }
 if (Name) delete [] Name;
}
```

Деструктор ссылается на новый метод `search`, который ищет фамилию студента `student` в текущем списке:

```
int Student::search (Student *student) {
 for (int i=0; i<count; i++) {
  if (stricmp(student->getName(),students[i]->getName())==0) return i;
  //найдено - вернуть номер в списке
 }
 return -1; //не найдено
}
```

Статический метод печати выводит весь текущий список:

```
void Student::print(void) {
 printf ("\nВсего: %d",count);
 for (int i=0; i<count; i++) {
  if (students[i]==NULL) break;
  students[i]->showStudent(); //dynamic_cast
 }
}
```

Закомментированное действие лучше выполнять с помощью оператора динамического приведения типа `<dynamic_cast>` (так как метод `showStudent` переопределён в дочернем классе и, вообще говоря, может быть перегруженным).

Наконец, напишем реализацию функций, переопределяющих стандартные операторы в наших классах.

Оператор `!` мы приспособили для проверки того, есть ли объект, к которому он применяется, в списке:

```
int Student::operator ! () {
 int Found = search (this);
 return (Found==-1? 0 : 1);
}
```

Оператор `+=` будет добавлять строку, переданную параметром, к фамилии студента:

```
void Student::operator += (char *Name) {
 if (strlen(Name)>0) {
  char *buf = new char [strlen(this->Name)+strlen(Name)+1];
  buf[0]='\0';
  strcat (buf,this->getName()); strcat (buf,Name);
  this->setName(buf);
  delete [] buf;
 }
}
```

Напомним, что метод `setName` должен вызываться только для созданного и проинициализированного объекта, иначе явное освобождение памяти, выполняемое оператором `delete [] Name` в этом методе, может привести к "преждевременному" удалению объекта, например, при неявном вызове конструктора копирования. Одним из решений проблем, связанных с такими трудноуловимыми ошибками, стала фактическая ликвидация деструкторов в языках Java, PHP и др. Возможность вызова деструктора в этих языках заменяется процедурой "сборки мусора" (garbage collection), периодически выполняемой системой.

Более корректное определение +=, работающее по цепочке, было бы, например, таким:

```
Student & Student::operator += (char *Name) {
 if (strlen(Name)>0) {
  char *buf = new char [strlen(this->Name)+strlen(Name)+1];
  buf[0]='\0';
  strcat (buf,this->getName()); strcat (buf,Name);
  this->setName(buf);
  delete [] buf;
 }
 return *this;
}
```

Пример вызова такого метода:

```
Student *t = new Student ("Smirnov",210);
Student s2 = (*t += " E.S.");
```

"Сложение" студентов в нашем случае будет означать сцепление их фамилий и сложение номеров групп. Само по себе это действие бессмысленно, но оно иллюстрирует, как переопределять бинарный оператор:

```
Student Student::operator + (Student &s) {
 Student *temp= new Student ();
 char *buf = new char [strlen(this->Name)+strlen(s.getName()+1)];
 buf[0]='\0';
 strcat (buf,this->getName()); strcat (buf,s.getName());
 temp->setName (buf);
 temp->setGroup (this->getGroup()+s.getGroup());
 return *temp;
}
```

В классе-потомке `Hobbit` префиксный оператор `++`, который должен возвращать ссылку на объект, увеличивает на 1 код символа, соответствующего свойству `Hobby`:

```
Hobbit & Hobbit::operator ++ () {
 this->setHobby (this->getHobby()+1);
 return *this;
}
```

Оператор постфиксного инкремента отличается тем, что имеет неиспользуемый параметр типа `int`. Оператор возвращает временный объект *с прежним* значением поля `Hobby`, но всё равно увеличивает значение `Hobby` текущего объекта `this`. Именно так будет обеспечена корректная работа постфиксного оператора, срабатывающего *после* вычисления выражения, в котором он встретился:

```
Hobbit Hobbit::operator ++ (int) {
 Hobbit temp(*this);
 this->setHobby(this->getHobby()+1);
 return temp;
}
```

Продемонстрируем запрограммированные действия в `main.cpp`:

```
#include <iostream>
#include "student.h"
using namespace std;

int main() {
 setlocale (LC_ALL,"Russian");
 Student *Vova = new Student ("Vova",0);
 Student *Petya = new Student ("Petya",113);
 Hobbit *Frodo = new Hobbit ("Frodo",114,'A');
 Vova->add(); Frodo->add(); Petya->add();
 Hobbit *Sam= new Hobbit ("Sam",114,'B');
 Sam->add();
 printf ("\nStart list:"); Student::print();
 delete Sam; delete Vova; delete Petya;
 printf ("\nList after deleting:"); Student::print();

 printf ("\n!Frodo=%d",!*Frodo);
 delete Frodo;
 printf ("\nAfter deleting !Frodo=%d",!*Frodo);

 Hobbit Mike("Mike",221,'P');
 Mike+=" Robbins"; Mike.showStudent();
 Hobbit Mike1 = Mike++; printf ("\nMike++=");  Mike1.showStudent();
 printf ("\nThen, Mike++=");  Mike.showStudent();
 Hobbit Mike2 = ++Mike; printf ("\n++Mike=");  Mike2.showStudent();

 Student *Name = new Student ("Name",100);
 Hobbit *SurName = new Hobbit ("SurName",201);
 Name->showStudent(); SurName->showStudent();
 Student Sumkin = *Name + *SurName;
 printf ("\n*Name + *SurName="); Sumkin.showStudent();

 cin.sync(); cin.get(); return 0;
}
```

Пример 2. Определим небольшой класс комплексных чисел и проиллюстрируем на нём некоторые особенности переопределения операторов.

```
#include <stdio.h>
#include <windows.h>
#include <locale.h>

class C { //C - имя класса комплексных чисел
 float re,im;
  //свойства для хранения действительной и мнимой частей
 public:
 C (float re=0, float im=0) { this->re=re; this->im=im; } //конструктор
 C operator + (C &);
  //переопределяем БИНАРНЫЙ "+" для сложения комплексных чисел
```

```
 C& operator + (void);
  //переопределяем УНАРНЫЙ префиксный "+" для смены знака мнимой части
 C& operator ++ (void);
  //переопределяем префискный ++
 C operator ++ (int);
  //переопределяем постфискный ++
 void show (void); //метод для вывода числа в консоль
};

C C::operator + (C &c2) {
 //реализация БИНАРНОГО "+" - сложения чисел
 C sum = C (this->re+c2.re,this->im+c2.im);
 return sum;
}

C& C::operator + (void) {
 //реализация УНАРНОГО "+" -смены знака у мнимой части
 im=-im; return *this;
}

C& C::operator ++ (void) {
 //переопределяем префиксный ++
 ++re; return *this;
}

C C::operator ++ (int) {
 //переопределяем постфиксный ++ - нужен параметр int
 C z(this->re,this->im); ++re; return z;
}

void C::show(void) {
 //вывод числа - не печатаем лишний "-" перед мнимой частью
 if (im<0) printf ("\n%.2f%.2fi",re,im);
 else printf ("\n%.2f+%.2fi",re,im);
}

int main (void) { //Демо
 setlocale (LC_ALL,"Russian");
 C c1(1,-1);
 C c2(2,2);
 C c3=c1+c2; //сложили 2 комплексных числа
 printf ("\n\nПервое число c1"); c1.show();
 printf ("\nВторое число c2"); c2.show();
 printf ("\nСумма c3=c1+c2"); c3.show();
 c2=+c2; //сменили знак у мнимой части
 printf ("\nСмена знака мнимой части числа c2"); c2.show();
 printf ("\nВыполнили c0=c1++");
 C c0=c1++;
```

```
 printf ("\nПосле этого c0"); c0.show();
 printf ("\nПосле этого c1"); c1.show();
 printf ("\nВыполнили c0=++c2");
 c0=++c2;
 printf ("\nПосле этого c0"); c0.show();
 printf ("\nПосле этого c2"); c2.show();

 printf ("\n"); system("pause");  return 0;
}
```

Пример 3. В классе `Class`, представляющем собой "обёртку" для обычного целочисленного значения, проиллюстрирована перегрузка таких операторов, как присваивание и круглые скобки, а также показана перегрузка операторов функциями-друзьями класса (на практике не рекомендуется перегружать "друзьями" любые операторы, меняющие состояние объекта).

```
#include <iostream>
#include <stdlib.h>
using namespace std;

class Class {
 int n; //приватный член класса
 public:
 Class (int n=0) { this->n=n; }
 inline void set (int n) { this->n=n; }
 void show ();
 void show (char *);
 Class & operator + (Class op2);
/* Если ф-я перегрузки оператора - член класса, то при перегрузке
 бинарного оператора 1-й операнд (до знака операции) передается
 неявно (через this), 2-й - через параметр функции.
 Здесь мы перегрузили бинарное сложение */
 Class & operator ++ ();
/* Для унарного оператора операнд передается неявно, явный параметр
 не нужен. Перегрузили префиксный инкремент */
 Class & operator ++ (int);
/* Для постфиксного оператора функции передается доп. неиспользуемое
 целочисл. значение, чтобы комплятор мог отличить постфиксный
 оператор от префиксного. Перегрузили постфиксный ++ */
 friend Class & operator * (Class op1, Class op2);
/* Если ф-я перегрузки оператора - не член, а друг класса (friend),
 то бинарному оператору явно передаются оба операнда, а унарному
 - явно передается ссылка на его единственный операнд).
 Перегрузили умножением "другом" класса */
 friend Class & operator + (Class op1, int n);
 friend Class & operator + (int n, Class op2);
/* Перегрузили "друзьями" сложение объекта класса с числом */
 Class & operator -- ();
 Class & operator -- (int);
```

```
/* Перегрузили унарные -- */
 void operator () (int n, ... );
/* Перегрузили оператор вызова функций (), он будет прибавлять к
 полю n объекта класса любое кол-во целых чисел */
 Class & operator = (Class);
/* Перегрузили присваивание в классе Class */
};

void Class::show (void) { cout << this->n << endl; }
void Class::show (char *hdr) { cout << hdr << " "; this->show(); }

Class & Class::operator + (Class op2) {
 Class *op = new Class (this->n + op2.n);
 return *op;
}

Class & operator + (Class op1, int n) {
 Class *op = new Class (op1.n + n);
 return *op;
}

Class & operator + (int n, Class op2) {
 Class *op = new Class (n + op2.n);
 return *op;
}

Class & operator * (Class op1, Class op2) {
 Class *op = new Class (op1.n * op2.n);
 return *op;
}

Class & Class::operator = (Class op2) {
 this->n = op2.n; return *this;
 //Возвращаем текущий объект!
}

Class & Class::operator ++ () { //Префиксный ++
 (this->n)++; return *this;
}

Class & Class::operator ++ (int unused) { //Постфиксный ++
 Class *op = this;
 this->n++;
 return *op;
}

Class & Class::operator -- () {
 this->n--; return *this;
```

```
}

Class & Class::operator -- (int unused) {
 Class op(*this);
 this->n--;
 return op;
}

void Class::operator () (int n, ...) {
/* Переопределенные () создают не новый вид вызова функций,
 но операторную функцию, к-рой можно передать любое число
 параметров: n обозначает число остальных параметров, например,
 Class z; z.set (0); z(3,1,2,3); //К z прибавил 1+2+3
*/
 int *item = &n;
 for (;n;n--) this->n += *++item;
}

int main () {
 Class a,b,c;
 a=1; b=a;
 b=c=a;
 a.show ("A"); b.show ("B"); c.show ("C");
 b=a++; b.show ("b=a++");
 c=++b; c.show ("c=++b");
 a.show ("A"); b.show ("B"); c.show ("C");
 Class e=a*b*c; e.show("E=A*B*C");
 Class f=e+2; f.show ("F=E+2");
 Class g=1+e; g.show ("G=1+E");
 Class h; h(5,1,2,3,4,5); h.show ("H(...)");
 system("pause>nul");
 return 0;
}
```

Пример 4, совсем простой. Шаблон для класса-массива из элементов любого скалярного типа (например, int, double, char)

```
#include <iostream>
using namespace std;

template <class T> class C {
private:
 T *data;
 int n;
public:
 C(int n) {
  this->n = 0;
  if (n > 0) {
   this->n = n;
```

```
   data = new T [n];
   if (!data) { data = nullptr; this->n = 0; return; }
   for (int i = 0; i < n; i++) data[i] = 0;
  }
 }
 void show(char *hdr) {
  cout << endl << hdr << ": ";
  for (int i=0; i<n; i++) cout << data[i] << " ";
 }
};

int main() {
 C <int> a(10); a.show("A");
 C <double> b(5); b.show("B");
 C <char> c(5); b.show("C");
 cin.get(); return 0;
}
```

Примеры проверены в бесплатной сборке Visual Studio 2010 Express или в Studio 2015.

Все лекции по C/C++

теги: c++ учебное

Здесь можно оставить комментарий, обязательны только выделенные цветом поля. Не пишите лишнего, и всё будет хорошо

Ваше имя:

Пароль (если желаете запомнить имя):

Любимый URL (если указываете, то вставьте полностью):

Текст сообщения (до 1024 символов):

Введите код сообщения: 255_5

Добавить Сброс

24.03.2016, 17:56; рейтинг: 10650