МИНИСТЕРСТВО ОБРАЗОВАНИЯ И НАУКИ
ДОНЕЦКОЙ НАРОДНОЙ РЕСПУБЛИКИ
ГОСУДАРСТВЕННОЕ ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ
ВЫСШЕГО ПРОФЕССИОНАЛЬНОГО ОБРАЗОВАНИЯ
«ДОНЕЦКИЙ НАЦИОНАЛЬНЫЙ ТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ»

КОНСПЕКТ ЛЕКЦИЙ
по дисциплине «Объектно-ориентированное программирование» (часть 1)

Донецк
2021

МИНИСТЕРСТВО ОБРАЗОВАНИЯ И НАУКИ
ДОНЕЦКОЙ НАРОДНОЙ РЕСПУБЛИКИ
ГОСУДАРСТВЕННОЕ ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ
ВЫСШЕГО ПРОФЕССИОНАЛЬНОГО ОБРАЗОВАНИЯ
«ДОНЕЦКИЙ НАЦИОНАЛЬНЫЙ ТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ»

КАФЕДРА КОМПЬЮТЕРНОГО МОДЕЛИРОВАНИЯ И ДИЗАЙНА

КОНСПЕКТ ЛЕКЦИЙ
по дисциплине «Объектно-ориентированное программирование» (часть 1)

для обучающихся по направлениям подготовки
02.03.01 «Математика и компьютерные науки»,
09.03.02 «Информационные системы и технологии»
всех форм обучения

РАССМОТРЕНО
на заседании кафедры
компьютерного моделирования и дизайна
Протокол № __ от _____

УТВЕРЖДЕНО
на заседании учебно-издательского
совета ДОННТУ
Протокол № ___от _____

Донецк
2021

УДК 004.432

Составитель:
Павлий Виталий Александрович – кандидат технических наук, доцент кафедры компьютерного моделирования и дизайна ГОУВПО «ДОННТУ»

М54 Конспект лекций по дисциплине «Объектно-ориентированное программирование» (часть 1) [Электронный ресурс]: для обучающихся по направлениям подготовки 02.03.01 «Математика и компьютерные науки», 09.03.02 «Информационные системы и технологии» всех форм обучения / ГОУВПО «ДОННТУ», каф. компьютерного моделирования и дизайна; сост. В. А. Павлий. — Донецк : ДОННТУ, 2021. – Систем. требования: ZIP-архиватор. – Загл. с титул. экрана.

Конспект лекций разработан с целью оказания помощи обучающимся в усвоении и обобщении теоретического материала по дисциплине «Объектно-ориентированное программирование» и содержит основной теоретический материал, изучаемый в первой части курса.

УДК 004.432

Содержание

ЛЕКЦИЯ 1.
ПАРАДИГМЫ ПРОГРАММИРОВАНИЯ. ОСНОВНЫЕ ПОНЯТИЯ ООП

Парадигмы программирования

Парадигма программирования - это совокупность идей и понятий, определяющих стиль написания программ.

Не определяется однозначно языком программирования. Практически все современные языки программирования в той или иной мере допускают использование различных парадигм.

Существуют основные модели (виды) программирования:

- *функциональное программирование* - это способ программирования, основанный на использовании подпрограмм (функций и процедур);
- *структурное программирование* - это способ программирования, основанный на представлении программы в виде иерархической структуры блоков;
- *декларативное программирование* - это способ программирования, в котором описывается, что представляет собой проблема и что должно получиться в результате;
- *императивное программирование* - это способ программирования, в котором программа состоит из набора инструкций, выполняемых последовательно;
- *логическое программирование* - это способ программирования, основанный на автоматическом доказательстве теорем;
- *объектно-ориентированное программирование*

Объектно-ориентированное программирование бывает:

- основанное на классах;
- основанное на прототипах;
- основанное на субъектах.

Объектно-ориентированное программирование, основанное на прототипах - это парадигма, которая позволяет переопределять и добавлять новые функции уже существующим объекта.

- *субъектно-ориентированное программирование* - это способ программирования, основанный на построении ООП-систем, как композиции субъектов.

Объектно-ориентированное программирование, основанное на классах

Ключевыми понятиями ООП, основанного на классах, является класс и объект. *Класс* - это программная единица, представляющая собой совокупность характеристик какой-либо сущности и действий, которые эта сущность может выполнять.

Характеристики называются свойствами, действия - методами, сущность - объектом.

Фактически, *класс* - тип данных, определяемый пользователем и описывающий сущность.

Объект - это переменная типа «класс» или экземпляр «класса»

Свойства объектно-ориентированного программирования

Принято считать, что ООП рядом свойств:

1. *Инкапсуляция* - это объединение в единое целое свойств и методов с одновременным скрытием информации, которая не нужна для внешнего использования.
2. *Наследование* - это свойство ООП, позволяющие в одном классе использовать полностью или частично функционал другого класса. Класс, от которого берется функционал или его часть, называется родительский класс, а который берет функционал - дочерний
3. *Полиморфизм* - это свойство ООП, при котором одно и тоже имя может вызывать различные действия на этапе выполнения. В практике программирования проявляется в двух случаях:

- при создании объектов;
- при передаче объектов в виде аргументов в методы других классов

В практике программирования выделяют также четвертое свойство ООП:

4. *Абстрагирование* - это свойство ООП, при котором класс может содержать только описание (декларации) тех или иных методов без их реализации. Такие классы используются в основном в тех случаях, когда точно известно, что класс должен что-то делать, но неизвестно, как он должен это делать.

Модификаторы доступа в классы

Принцип инкапсуляции предписывает скрывать как можно больше ненужной для использования извне информации, используя модификаторы доступа.

Существует 3 модификатора доступа:

1. *Private* - модификатор доступа, дающий доступ к полям и методам класса только для других методов этого же класса. Приватный уровень доступа считается самым низким, т.к. дает доступ к минимальному количеству программных конструкций;
2. *Protected* - модификатор доступа, дающий доступ для методов текущего класса и всех его наследников(средний уровень доступа);

3. *Public* - модификатор доступа, дающий доступ для любого метода или любой функции в программе без ограничений.

Если модификатор доступа не указан, то поле или метод автоматически становится приватным.

Описание класса

Описание класса приблизительно выглядит так:

```
class <имя> {
[private:]
<описание скрытых элементов>
[protected:]
<описание защищенных элементов>
[public:]
<описание доступных элементов>
}; // описание заканчивается точкой с запятой обязательно!!!
```

Модификаторы доступа могут быть назначены как свойствами класса, так и методами. В практике программирования принято свойства класса делать приватными, а методы класса - публичными. Такое ограничение связано с тем, что метод класса в состоянии проверить и не допустить некорректного использования данных.

Виды классов

Классы бывают:

– *глобальные* (объявленные вне любого блока);

– *локальные* (объявленные внутри блока, например, функции или другого класса).

Свойства полей класса

Поля класса:

– могут иметь любой тип, кроме типа этого же класса, но могут быть указателями или ссылками на этот класс (данное утверждение только для C++);

– могут быть описаны с модификатором const в том случае, если поле представляет собой константу, при этом они инициализируются только один раз (с помощью конструктора) и не могут изменяться;

```
class test
{
   private:
      const double pi = 3.14;
}
class test
{
   private:
      const double pi = 3.14;
   void modify ()
   {
      pi = 5; // error
   }
} //  приведет к ошибке
```

– поля могут быть описаны с модификатором static, но не с auto, extern и register.

Инициализация полей при описании не допускается, так как память под класс не выделяется, пока не будет создан экземпляр класса.

ООП и раздельная компиляция кода

В практике программирования на C++ принято разделять декларативную и функциональную часть программного кода. Именно поэтому описание любого класса физически находится в 2-х файлах:

1. файл с расширением «.h» содержит декларативную (описательную) часть;
2. файл с расширением «.cpp» содержит функциональную часть или реализацию класса в виде множества методов.

Оба файла должны иметь одно и то же имя.

При использовании класса программист подключает декларативную часть как обычную библиотеку, функциональная — подключается автоматически.

Единственное отличие при подключении класса заключается в использовании двойных кавычек (″″) вместо угловых скобок (< >).

Данный механизм называется *раздельной компиляцией кода.*

Для описания реализации методов класса используется в файлах «.cpp» необходимо указывать сначала имя класса, затем имя метода, между которыми ставится оператор «::». Этот оператор называют *оператором расширения области видимости*.

Пример: вычисляем площадь круга - S

```
// a.h
class test
{
   private:
      const double pi = 3.14;
   public:
      double getArea(double radius);
};
// a.cpp
#include ″a.h″
double test::getArea(double radius)
{
   return radius*radius*pi;
}
// main.cpp
#include <stdlib.h>
#include ″a.h″
int main()
{
   test t; double rad;
   scanf(″%f″, &rad);
   double area = t.getArea(rad);
   printf(″%5.2f″, area);
   return 0;
}
```

ЛЕКЦИЯ 2.
СОЗДАНИЕ И УДАЛЕНИЕ ОБЪЕКТОВ.
КОНСТРУКТОР И ДЕСТРУКТОР КЛАССА

Создание объектов

Создание объекта в зависимости от места его расположения в памяти компьютера бывают двух видов:

- *статическое;*
- *динамическое.*

Соответственно, статические объекты располагаются в статической памяти, а динамические - в куче (динамической памяти).

Создание статических объектов имеет определенные ограничения, связанные с ограничением объема статической памяти программы. В современных языках программирования объекты в основном принято располагать в динамической памяти (куче), которая имеет объем до 80-ти процентов от общего объема памяти, выделяемого программе.

```
point p1; // создание объекта в статистической памяти;
point *p2 = new point(20,20);// создание объекта в динамической памяти.
```

Создание объектов в динамической памяти состоит из 2-х этапов:

1. Объявление указателя на объект в статистической памяти;
2. Создание объекта в куче и присвоение указателю его адреса.

Создание объектов в динамической памяти происходит при помощи конструкции new, которая выступает аналогом malloc из языка С.

При этом new является более производительной и легкой в исполнении, т.к. не требует вычисления необходимого для создания объекта объема памяти.

По способу конструирования объекта выделяют 3 способа создания объекта:

1) создание объектов с инициализацией по умолчанию;

```
point p;
point *p = new Point();
```

2) создание объектов со специальной инициализацией;

```
point p(20,20);
point *p = new Point(20,20);
```

Количество параметров в данном способе зависит от способов конструирования объектов, поддерживаемых классом.

3) создание объектов путем копирования других объектов. Копирование объекта осуществляется на основе объекта-образца, такой объект называют *прототипом*, а сам процесс копирования – *прототипированием*. При прототипировании создаваемый объект получает те же самые характеристики, которые имеет и прототип. Для копирования внутри класса должен быть реализован специальный конструктор – конструктор копирования (конструкторы классов рассматриваются далее).

```
point p1(20,20); // Прототип
point p2 = p1; // Создание объекта путем копирования в статической памяти
point *p1 = new point(20,20); // Прототип
point *p2 = new point(&p1);   // Создание объекта путем копирования
                              // в динамической памяти
```

Обращение к свойствам и методам класса

После создания объектов программист получает возможность обращаться к их полям и методам. При этом в зависимости от того, в какой области памяти был создан класс, обращение к полям и методам осуществляется по-разному.

Если класс был создан в статической области памяти, то обращение к его методам и свойствам осуществляется через оператор «.», например:

```
point p1;
int x = p1.getX();
```

Если класс был создан в динамической области памяти, то обращение к его методам и свойствам осуществляется через оператор «->», например:

```
point* p2 = new Point();
int x = p2->getX();
```

Удаление объектов

Объекты, создаваемые в статистической памяти, в удалении не нуждаются. Такие объекты удаляются автоматически во время завершения работы программы. Однако, объекты, создаваемые в динамической памяти при помощи *new*, требуют удаления. Если удаление не выполнить, то отведенная область памяти будет считаться зарезервированной за данным приложением и не освободится даже после его закрытия. Такая ситуация называется *утечкой памяти*.

Для удаления используется конструкция *delete*. Она принимает указатель на область памяти, которую необходимо очистить.

```
point *p = new Point(10,10);
delete p;
```

Getter и Setter

В практике программирования приходится очень часто манипулировать значениями свойств, объявленными в классе, при этом из соображений безопасности эти свойства объявляются приватными. Для корректного чтения и записи этих свойств, программист должен разработать специальные методы называемые *getter* и *setter*.

Чаще всего имена методов выбираются по имени свойства с добавлением префикса *get* и *set.*

Setter может содержать все необходимые проверки на корректность значения свойства.

```
class Student
{
   private:
      char *Name;
      double avgBall;
   public:
      char* getName ()
      {
         return Name;
      }
      void setName(char *_Name)
      {
```

```
            if (strlen(_Name) == 0)
               printf ("Error");
            else
               strcpy(_Name, Name)
         }
         double getAvgBall()
         {
            return avgBall;
         }
         void setAvgBall (double _avgBall)
         {
            if (_avgBall < 2 && _avgBall > 5)
               printf("Error");
            else
               avgBall = _avgBall;
         }
};
```

Указатель this

Обычно в указателе находится адрес объекта (номер ячейки оперативной памяти, с которой начинается объект). При этом объект может занимать несколько смежных ячеек памяти. *Указатель this* - это указатель на текущий (этот) объект, доступный только внутри класса.

Указатель *this* в основном используется для решения конфликтных ситуаций, при обращении к одноименным свойствам класса, для возврата в вызывающий метод ссылки на текущий. Это необходимо для организации внешнего управления объектом, например удаления. К примеру, конструкция *delete this* приведет к ошибке компиляции, т.к. объект сам себя удалять не может.

Правилом хорошего тона считается использовать *this* всякий раз, когда идет обращение к методам и свойствам класса внутри этого класса.

Так как this по определению указатель, то работа с ним возможна только при помощи оператора «->».

Пример 1: Использование указателя *this* для решения конфликтной ситуации при обращении к одноименным свойствам класса:

```
class Student
{
   private:
      double avgBall;
   public:
      void setAvgBall(double avgBall)
      {
         if (avgBall < 2 && avgBall > 5)
            printf("Error");
         else
            this->avgBall = avgBall;
      }
};
```

Пример 2: Добавление / удаление (отчисление) студентов

```
class Student
{
   private:
      double avgBall;
   public:
      void setAvgBall(double avgBall)
```

```
        {
           if (avgBall < 2 && avgBall > 5)
              printf("Error");
           else
              this->avgBall = avgBall;
        }
        Student* getLink()
        {
           if (this -> avgBall < 2.5)
              return this;
           else
              return NULL;
        }
};
class Faculty
{
   private:
      Student* st[10];
   public:
      void Add(Student* s)
      {
         // Добавляет студента в массив
         int flag=0;
         for (int i=0; i<10; i++)
         {
            if (this->st[i] == NULL)
            {
                flag=1;
                this->st[i] = s;
                break;
            }
         }
         if (flag==0)
            printf("Error");
      }
      void Otchislenie()
      {
         // Отчислить всех студентов у которых средний балл менее 2.5
         for (int i=0; i<10; i++)
         {
            Student* s = this->st[i]->getLink();
            if (s != NULL) {
               delete s; this->st[i] = NULL;
         } }
      }
};
```

Перегрузка (Overloading) и переопределение (Overriding) методов класса

Перегрузка методов - это возможность ООП объявлять внутри класса более одного метода с одинаковым именем, но разными аргументами. При этом тип возвращаемого значения не учитывается.

Фактически, *перегрузка - типичное проявление полиморфизма.*

Пример: Вывод информации о студенте (всей или какой-то определенной)

```
class Student
{
   private:
      char *Name;
      double avgBall;
   public:
```

```
        void printInfo(int typeInfo)
        {
           if (typeInfo == 1)
              printf("%s", this->Name);
           if (typeInfo == 2)
              printf("%5.2f", this->avgBall);
        }
        void printInfo()
        {
           printf ("%s %5.2f", this->Name, this->avgBall);
        }
};
```

Переопределение метода - это замена метода, функционал которого был определен ранее. Наиболее часто переопределение проявляется при наследовании. Наследование классов рассматривается далее.

Конструкторы класса

Конструктор класса – это метод, который вызывается при создании объекта и производит инициализацию переменных членов.

Свойства конструктора:

– конструктор имеет такое же имя, как и класс.

– конструктор не возвращает значения, даже типа void; нельзя получить указатель на конструктор;

– класс может иметь несколько конструкторов с разными параметрами для разных видов инициализации (при этом используется перегрузка);

– конструктор, вызываемый без параметров, называется конструктором по умолчанию;

– параметры конструктора могут иметь любой тип, кроме этого же класса, однако допускается указатель на этот класс;

– если в классе не указано ни одного конструктора, то компилятор создаст его автоматически (с пустым телом);

– конструкторы не наследуются;

– конструкторы нельзя описывать с модификаторами const, virtual и static (статические конструкторы есть в Java, C# и других современных языках программирования, но не в C++);

– конструкторы глобальных объектов вызываются до вызова функции main; локальные объекты создаются, как только становится активной область их видимости.

Конструктор автоматически вызывается, если в программе встретилась какая-либо из синтаксических конструкций:

```
имя_класса имя_объекта[(список параметров)];
//список параметров может быть пустым
имя_класса(список параметров);
//создается объект без имени (список параметров может быть пустым)
имя_класса имя_объекта = выражение;
//создается объект посредством копирования
```

Например:

```
point p1(10,10), p2(20), p3;
point p4 = point(100);
point p5 = 200;
```

В первом операторе создаются три объекта. Значения не указанных параметров устанавливаются по умолчанию. Во втором операторе создается безымянный объект со значением координаты x=100 (значение координаты y устанавливается по умолчанию). Выделяется память под объект p4, в которую копируется безымянный объект.

В третьем операторе создается безымянный объект со значением координаты x=200 (значение координаты y устанавливается по умолчанию). Выделяется память под объект p5, в которую копируется безымянный объект. Такая форма создания объекта возможна в том случае, если для инициализации объекта допускается задать один параметр.

Пример: Использование перегруженных конструкторов.

```
enum color {red, green, blue};
class point
{
   private:
      double x, y;
      color c;
   public:
      point(double x, double y);
      point(color c);
};
point::point(double x, double y)
{
   this->x=x;
   this->y=y;
   this->c=red;
}
point::point(color c)
{
   switch (c)
   {
      case red:
         this->x=100;
         this->y=100;
         this->c=red;
      break;
      case green:
         this->x=200;
         this->y=200;
         this->c=green;
      break;
      case blue:
         this->x=300;
         this->y=300;
         this->c=blue;
      break;
   }
}
```

Есть еще один способ инициализации полей в конструкторе – с помощью *списка инициализаторов*, расположенных после двоеточия между заголовком и телом конструктора, например:

```
point::point(double x):y(100),c(green)
{
   this->x=x;
}
```

Такая запись обозначает, что поля y и c будут инициализированы значениями «100» и «green» соответственно, в то время как поле x будет инициализировано

при помощи передаваемого параметра. Подобный способ инициализации сохранился в C++ и отсутствует во многих современных языках программирования.

После описания класса можно создавать объекты тремя разными способами (так как в классе выше три конструктора):

```
point p1(10,10);
point p2(green);
point p3(20);
```

Деструктор класса

Деструктор класса - это метод, который используется для выполнения определенных операций при удалении объекта. Обычно деструктор выполняет операции, обратные тем, которые выполняли конструкторы. Например, если конструктор выделяет динамическую память для членов класса, то деструктор ее освобождает.

Деструктор вызывается автоматически:

– для глобальных объектов как часть процедуры выхода из main;

– для локальных объектов – при выходе из блока, в котором они объявлены;

– для объектов, заданных через указатели, деструктор вызывается при использовании операции delete.

Свойства деструктора:

– деструкторы (в отличие от конструкторов) могут быть виртуальными.

– деструкторам нельзя передавать аргументы.

– в каждом классе может быть объявлен только один деструктор.

– имя деструктора состоит из имени класса, перед которым стоит символ «~» (тильда).

Пример: деструктор класса

```
class point
{
   private:
      int x, y;
   public:
      point()
      {
         this->x=0;
         this->y=0;
      }
      ~point()
      {
      }
};
```

ЛЕКЦИЯ 3.
СТАТИЧЕСКИЕ ЭЛЕМЕНТЫ КЛАССА И КОНСТАНТНЫЕ ОБЪЕКТЫ. ДРУЖЕСТВЕННЫЕ ФУНКЦИИ И КЛАССЫ

Статистические элементы класса

С помощью модификатора static можно описать статические поля и методы класса. *Статические элементы класса принадлежат самому классу, а не объектам, порожденным от него*, и, как следствие, всегда существуют в единственном экземпляре.

Статические поля

Статические поля применяются для хранения данных, общих для всех объектов класса, например, количества объектов или ссылки на разделяемый всеми объектами ресурс. Эти поля существуют для всех объектов класса в единственном экземпляре, то есть не дублируются.

Память под статическое поле выделяется один раз при его инициализации независимо от числа созданных объектов (и даже при их отсутствии) и инициализируется с помощью оператора расширения области видимости «::». В дальнейшем статические поля будут доступны как через имя класса, так и через имя объекта.

Пример: Подсчет количества созданных объектов класса.

```
#include<stdio.h>
class A
{
   public:
      static int count;
      A()
      {
         A::count++; // в языке C++ допускается также this->count++;
      }
      ~A()
      {
         A::count--; // в языке C++ допускается также this->count--;
      }
};
int A::count = 0;
int main()
{
   A *a;                    // (1)
   A b;                     // (2)
   A *c = new A();          // (3)
   printf("%5d", A::count); // выведется 2
   delete c;                // (4)
   printf("%5d", A::count); // выведется 1
   return 0;
}
```

В примере выше в строке (1) объект не создается, а лишь объявляется указатель на него, поэтому конструктор класса вызван не будет. В строках (2) и (3) объекты создаются – в динамической и статической памяти соответственно. Здесь в обоих случаях вызовется конструктор, который увеличит счетчик созданных объектов. После выполнения операции *delete* в строке (4) объект в динамической памяти будет уничтожен, что приведет к вызову деструктора и уменьшению

счетчика. Объект, созданный в статической памяти, будет существовать до завершения функции *main*.

Свойства статических полей:

– на статические поля распространяется действие спецификаторов доступа, поэтому статические поля, описанные как *private*, нельзя инициализировать с помощью оператора расширения области видимости «::». Им можно присвоить значения только с помощью статических методов (см. ниже).

– память, занимаемая статическим полем, не учитывается при определении размера объекта операцией *sizeof*.

– статические поля нельзя инициализировать в обычном конструкторе, так как они создаются до создания любого объекта, однако в современных языках программирования существуют также статические конструкторы, позволяющие выполнить указанную задачу.

Статические методы

Подобно статическим полям, статические методы существуют в единственном экземпляре и способны работать только со статическими полями. Статические методы играют важную роль в современных языках программирования. Так например, *точкой входа в программу* (местом откуда начинает исполняться программа) в языках Java, C# считается статический метод *main* объявленный и реализованный в одном из классов.

Свойства статических методов:

– статические методы могут обращаться только к статическим полям и вызывать другие статические методы класса.

– внутри статического метода указатель this недоступен.

– обращение к статическим методам производится так же, как к статическим полям – либо через имя класса, либо, если хотя бы один объект класса уже создан, через имя объекта.

– статические методы не могут быть константными (const) и виртуальными (virtual).

Константные объекты

Если создать объект с модификатором const, то компилятор будет проинформирован, что содержимое объекта не должно изменяться после инициализации. Чтобы предотвратить изменение значений элементов константного объекта, компилятор генерирует сообщение об ошибке, если объект используется с неконстантным методом класса.

Константный метод класса, объявляемый с ключевым словом *const* после списка параметров, должен удовлетворять следующим правилам:

– не может изменять значение элементов данных класса;

– не может вызывать неконстантные методы класса;

– может вызываться как для константных, так и неконстантных объектов класса.

Для того чтобы сделать метод константным, необходимо указать ключевое слово *const* после декларации метода, но до начала тела. Если объявление и

определение метода разделены, то модификатор *const* необходимо указать дважды – как при объявлении, так и при его определении. Те методы, которые только лишь считывают данные из поля класса, имеет смысл делать константными, поскольку у них нет необходимости изменять значения полей объектов класса. Например:

```
class CCoord
{
   private:
      int iX, iY;
   public:
      void SetCoord(int iX, int iY);
      void GetCoord(int &iX, int &iY) const;
};
void CCoord::SetCoord(int iX, int iY)
{
   this->iX = iX;
   this->iY = iY;
}

void CCoord::GetCoord(int &iX, int &iY) const
{
   iX = this->iX;
   iY = this->iY;
}
int main(void)
{
   CCoord c1;       // создаем неконстантный объект
   const CCoord c2; // создаем константный объект
   int x, y;
   c1.SetCoord(10,10);
/* - разрешено – вызов неконстантного метода для неконстантного объекта */
   c2.SetCoord(10,10);
/* - запрещено – вызов неконстантного метода для константного объекта */
   c1.GetCoord(x, y);
/* - разрешено: вызов константного метода для неконстантного объекта */
   c2.GetCoord(x, y);
/* - разрешено: вызов константного метода для константного объекта */
   return 0;
}
```

Обычно константные объекты создают для того, чтобы получить гарантии того, что данные объекта невозможно будет изменить. Однако иногда случаются ситуации, когда необходимо создать объект-константу, имеющий определенное поле, которое нужно будет изменять, несмотря на то, что сам объект является константой. Для этих целей необходимо соответствующее поле объявить со спецификатором *mutable*. Например, объявив в предыдущем примере поля iX, iY следующим образом:

```
mutable int iX, iY;
```

можно создать константный метод *ResetCoord,* разрешив тем самым модификацию полей константного объекта:

```
class CCoord
{
   private:
      mutable int iX, iY;
   public:
      void SetCoord(int iX, int iY);
      void ResetCoord(int iX, int iY) const;
      void GetCoord(int &iX, int &iY) const;
};
void CCoord::SetCoord(int iX, int iY)
```

```
{
   this->iX = iX;
   this->iY = iY;
}
void CCoord::ResetCoord(int iX, int iY) const
{
   this->iX = iX;
   this->iY = iY;
}
void CCoord::GetCoord(int &_iX, int &_iY) const
{
   iX = this->iX;
   iY = this->iY;
}
int main(void)
{
   CCoord c1;
   const CCoord c2;
   int x, y;
   c1.SetCoord(10,10);
/* - разрешено – вызов неконстантного метода для неконстантного объекта */
   c2.SetCoord(10,10);
/* - запрещено – вызов неконстантного метода для константного объекта */
   c1.GetCoord(x, y);
/* - разрешено: вызов константного метода для неконстантного объекта */
   c2.GetCoord(x, y);
/* - разрешено: вызов константного метода для константного объекта */
   c2.ResetCoord(10, 10);
/* - разрешено, т.к. iX, iY объявлены со спецификатором mutable */
   return 0;
}
```

Дружественные функции и дружественные классы

Иногда возникает необходимость иметь непосредственный доступ извне к скрытым полям класса, то есть расширить интерфейс класса. Для этого служат *дружественные функции и дружественные классы*.

Дружественная функция объявляется внутри класса, к элементам которого ей нужен доступ, с ключевым словом *friend.* В качестве параметра ей должен передаваться объект или ссылка на объект класса, поскольку указатель *this* ей не передается. Одна функция может дружественной сразу с несколькими классами.

Дружественная функция может быть обычной функцией или методом другого ранее определенного класса. На нее не распространяется действие спецификаторов доступа.

В качестве примера ниже приведено описание двух функций, дружественных классу monster. Функция kill является методом класса hero, а функция steal_ammo не принадлежит ни одному классу. Обеим функциям в качестве параметра передается ссылка на объект класса monster.

```
class monster; // Предварительное объявление класса
class hero
{
   public: void kill(monster);
};
class monster
{  private: int health, ammo;
   friend int steal_ammo(monster);
   /* Класс hero должен быть определен ранее */
```

```
   friend void hero::kill(monster);
};
int steal_ammo(monster M)
{
   return --M.ammo;
}
void hero::kill(monster M)
{
   M.health = 0;
   M.ammo = 0;
}
```

Дружественный класс. Если все методы какого-либо класса должны иметь доступ к скрытым полям другого класса, весь этот класс можно объявить дружественным с помощью ключевого слова *friend*. В приведенном ниже примере класс mistress объявляется дружественным классу hero:

```
class hero
{
   friend class mistress;
}
class mistress
{
   void f1();
   void f2();
}
```

Методы f1 и f2 являются дружественными по отношению к классу hero (хотя и описаны без ключевого слова friend) и имеют доступ ко всем его полям.

Объявление *friend* не является спецификатором доступа и не наследуется.

Свойства дружественных классов:

– отношение дружественности не наследуются, то есть, если A дружественен B, а C порожден от A, то это не означает, что C становится автоматически дружественным B;

– свойство дружественности не транзитивно, то есть, если класс A дружественен классу B, а класс B – классу C, то A не становится автоматически дружественным классу C;

– свойство дружественности не коммутативно, то есть, если A дружественен B, то это не означает, что B дружественен A. Но при этом A можно объявить дружественным B. Такие классы называют взаимодружественными.

Взаимодружественные классы

Пример описания взаимодружественного класса показан ниже:

```
class A; // неполное объявление класса class A
class B
{
   private:
      friend class A;
   public:
      void f(A* c1)
      {
      };
};
class A
{
   private:
      friend class B;
```

```
};
```

Секция доступа, в которой помещено объявление *friend*, не имеет значения. Обычно для наглядности объявление *friend* для класса помещается в первой секции (которая, как правило, имеет модификатор *private*). Единственное ограничение, налагаемое на объявление *friend*, заключается в том, что это объявление должно находиться внутри объявления класса.

Контейнерные классы

Контейнерные классы – это классы, которые содержат в своем описании один или несколько объектов или указателей на объекты. Например:

```
// fl.h
#include <stdio.h>
class Tail
{
   private:
      int length;
   public:
      Tail(int n)
      {
         this->length=n;
      }
      int GetTail()
      {
         return this->length;
      }
};
class Dog {
   private:
      Tail tail;
   public:
      Dog(int n) : tail(n)
      {
      };
      void DisplayPar()
      {
         printf("%d\n", tail.GetTail());
         return;
      }
};
//f1.cpp
#include "f1.h"
void main()
{
   Dog d(20);
   d.DisplayPar();
}
```

Сначала инициализируются все поля-объекты, которые содержатся в описании класса, причем в том порядке, в котором они объявлены. Деструкторы вызываются в порядке, обратном инициализации.

ЛЕКЦИЯ 4.
НАСЛЕДОВАНИЕ И ПОЛИМОРФИЗМ

Наследование классов

Как было отмечено ранее, *наследование* - это возможность использования в наследуемом классе функционала родительского класса. В C++ существует два вида наследования:

– *простое*;

– *множественное*.

Простое наследование – это такое наследование, при котором порождаемые классы наследуют методы и свойства одного родительского класса.

Множественное наследование – это такое наследование, при котором порождаемые классы наследуют методы и свойства нескольких родительских классов. В современных языках программирования множественное наследование не используется ввиду возможных конфликтов. Например, если в двух родительских классах существуют методы с одинаковыми именами и параметрами, то возникает конфликт наследования, при котором дочерний класс не знает, какой из методов ему унаследовать. В языке C++ в случае возникновения подобного конфликта методы унаследованы не будут, но программист, разрабатывающий дочерний класс, может вызвать любой из методов родительских классов вручную, через оператор «::».

При наследовании не существует ограничений на:

– количество производных классов;

– длину цепочки наследования.

Синтаксис наследования

Для того, чтобы унаследовать дочерний класс от родительского класса, необходимо указать имя родительского класса, отделив его символом «:» от имени дочернего класса. Также в языке C++ перед именем родительского класса можно указать модификатор доступа. Например:

```
#include <iostream>
using namespace std;
class FirstClass    // базовый класс
{
   protected:          // спецификатор доступа к элементу value
      int value;
   public:
      FirstClass()
      {
         this->value = 0;
      }
      FirstClass(int input)
      {
         this->value = input;
      }
      void show_value()
      {
        printf("%d\n", this->value);
```

```
      }
};
class SecondClass:public FirstClass   // дочерний класс
{
   public:
      // конструктор класса SecondClass вызывает
      // конструктор класса FirstClass
      SecondClass():FirstClass()
      {
      }
      // inputS передается в конструктор с параметром класса FirstClass
      SecondClass(int inputS):FirstClass (inputS)
      {
      }
      // возводит value в квадрат. Без спецификатора доступа protected
      // эта функция не могла бы изменить значение value
      void ValueSqr()
      {
         this->value *= this->value;
      }
};
int main()
{
    setlocale(LC_ALL, "rus");

    FirstClass F_object(3);       // объект базового класса
    printf("value F_object = ");
    F_object.show_value();

    SecondClass S_object(4);      // объект производного класса
    printf("value S_object = ");
    S_object.show_value();        // вызов метода базового класса
    S_object.ValueSqr();          // возводим value в квадрат
    printf("квадрат value S_object = ");
    S_object.show_value();

    // ошибка, так как базовый класс не имеет доступа
    // к методам производного класса
    // F_object.ValueSqr();
    return 0;
}
```

Спецификаторы доступа базовых классов

Спецификатор доступа базового класса позволяет при наследовании понижать или оставлять неизменным уровень доступа для всех наследуемых полей класса. Данный механизм действует только в языке C++. Данный механизм не позволяет повышать уровень доступа.

Результирующий спецификатор поля		**Спецификатор доступа базового класса**		
		private	**protected**	**public**
Спецификатор доступа поля	**private**	—	—	—
	protected	private	protected	protected
	public	private	protected	public

Порядок вызова конструкторов

При создании экземпляра класса вызывается его конструктор. Если класс является дочерним, то должен быть вызван конструктор родительского класса. Порядок вызова конструкторов в C++ фиксирован. Если родительский класс, в свою очередь, является производным, то процесс рекурсивно повторяется до тех пор, пока не будет достигнут корневой класс.

Пример вызова конструктора родительского класса:

```
class Man
{
   protected:
      char* fio;
   public:
      Man() {}
      Man(char* fio)
      { this->fio = fio; }
};
class Student: public Man
{
   private:
     double avg; // средний балл обучения
   public:
      Student(char* fio, double avg)
      {
         // Вызвать конструктор родительского класса. Если конструктор
         // не вызвать, то он будет вызван автоматически, однако
         // компилятор способен вызывать только конструктор по
         // умолчанию (без параметров)
         Man::Man(fio);

         // Далее идет инициализирующая часть дочернего класса
         this->avg = avg;
      }
};
```

Теперь при создании объекта *Student* будет вызван его конструктор, который в свою очередь вызовет конструктор родительского класса *Man*. Этот конструктор проинициализирует поле *fio*. После этого конструктор класса *Student* продолжит свое выполнение и проинициализирует поле *avg*.

Порядок вызова деструкторов

Деструктор базового класса при необходимости вызывается в порядке, обратном вызову конструкторов, т.е. в самом конце. Например:

```
class Man
{
   public:
      ~Man()
      {
         // Какие-то действия
      }
};
class Student: public Man
{
   public:
      ~Student()
      {
         // Какие-то действия
```

```
            // Здесь автоматически будет вызван деструктор родительского
            // класса. Вызывать вручную необязательно, т.к. декстуктор один
        }
};
```

Полиморфизм, раннее и позднее связывание

Ранее отмечалось, что полиморфизм - это способность ООП, при которой одно и тоже имя может вызывать различные действия на этапе выполнения. Полиморфизм – это, в первую очередь, характеристика методов, а не объектов. Несмотря на то, что полиморфизм реализуется через архитектуру класса, полиморфными могут быть только методы класса, а не весь класс целиком.

Для вызова тех или иных методов класса на этапе компиляции создается специальная таблица соответствий имен методов их адресам в оперативной памяти. В дальнейшем при вызове метода эта таблица используется для поиска адреса этого метода и передачи ему управления. Этот процесс называют *связыванием*. Однако на этапе компиляции не всегда возможно предсказать какому методу какого класса соответствует какой адрес, например для следующего кода:

```
class Man
{
   protected:
      char* fio;

   public:
      void setParam(char* fio)
      {
         this->fio = fio;
      }
};
class Student: public Man
{
   private:
      double avg;
   public:
      void setParam(char* fio)
      {
         this->fio = fio;
         this->avg = 5.0;
      }
};
void Func(Man* m)
{
   m->setParam("Ivanov"); // (1)
}
***
```

Благодаря полиморфизму, под видом объекта класса *Man* в функцию *Func* можно передать объект любого дочернего класса, например *Student*. В строке (1) вызывается метод *setParam*, однако для какого объекта это произойдет - *Man* или *Student* - предсказать на этапе компиляции невозможно. Это станет известно только на этапе выполнения программы. Поэтому в таких случаях вызов функции в исходном коде C++ только обозначается, без точного указания на то, какая именно функция вызывается. Этот процесс называется *позднее связывание*. Позднее связывание всегда выполняется компоновщиком (не компилятором).

Если же подобной неоднозначности не возникает, т.е. полиморфизма нет, и компилятор в состоянии предсказать точно, какой метод будет вызван, он выполнит связывание сразу. Этот процесс соответственно называется *ранним связыванием*.

При *раннем связывании* программы выполняются быстрее, но существенно ограничены возможности разработчика. При *позднем связывании* остро встает вопрос об эффективности исполняемой программы.

Способность объектно-ориентированных языков автоматически определять тип объекта на этапе выполнения программы называется RTTI (Run-Time Type Identification – идентификация во время выполнения).

Виртуальные методы

В C++ позднее связывание для метода определяется при его объявлении с помощью ключевого слова *virtual*. Позднее связывание имеет смысл только для методов, являющихся частью иерархии классов. Объявление метода виртуальным для класса, не используемого в качестве базового, синтаксически корректно, но приведет только к потере времени в момент выполнения.

Виртуальные методы – методы, вызов которых зависит от типа объектов. С помощью виртуальных методов объект определяет свои действия.

Чистый виртуальный метод

Когда виртуальный метод не переопределен в дочернем классе, то при вызове его в объекте дочернего класса вызывается версия из родительского класса. Однако в некоторых случаях реализовать виртуальный метод в родительском классе невозможно. Например, при объявлении родительского класса *Character* (игровой персонаж) точно известно, что он должен двигаться, атаковать, перерисовываться на экране. Но как он должен это делать - на данном этапе неизвестно. Если унаследовать в дальнейшем от этого класса дочерний класс *Archer* (лучник), то необходимая для разработки этого класса информация (скорость, внешний вид персонажа, число патронов и т.п.) становится очевидной и конкретной. При разработке родительского класса *Character* в этом случае можно определить методы-пустышки (с пустым телом), однако в этом случае не будет гарантии, что все эти методы будут переопределены в дочернем классе *Archer*. Язык C++ предлагает в качестве решения этой проблемы чистые виртуальные методы.

Чистый виртуальный метод - это метод, который объявляется в родительском классе, но при этом не имеет тела. Поскольку такой метод не имеет реализации, то всякий дочерний класс должен переопределить этот метод. Для объявления чистого виртуального метода используется следующая общая форма:

```
virtual тип имя метода(список параметров) = 0;
```

Здесь тип обозначает тип возвращаемого значения, а имя_метода является именем метода. В следующем примере показано, как описать чистый виртуальный метод.

Пример: класс с чистым виртуальным методом.

```
class figure {
   private:
      double x, y;
   public:
      void setCoord(double i, double j) {
         this->x = i;
         this->y = j;
      }
      virtual void show() = 0; // чистый виртуальный метод
};
```

При использовании чистых виртуальных методов в дочернем классе обязательно необходимо переопределить эти методы. Если дочерний класс не будет содержать переопределения этих методов, то компилятор выдаст ошибку. Это и является гарантией того, что все необходимые методы будут реализованы в дочернем классе.

Таким образом, *чистые виртуальные методы следует применять в тех случаях, когда точно известно, что класс должен что-то делать, но не известно как именно он должен это делать.*

Абстрактные классы

Если какой-либо класс имеет хотя бы один чистый виртуальный метод, то такой класс называется *абстрактным* (*abstract*). Абстрактный класс всегда служит в качестве родительского для других дочерних классов. Также в нем могут быть реализованы некоторые методы.

Свойства абстрактных классов:

– нельзя создавать объекты от абстрактного класса ввиду того, что некоторые методы в нем не имеют реализации;

– дочерний класс, наследуемый от абстрактного, должен переопределить все его чистые виртуальные методы. Если дочерний класс не переопределяет или переопределяет не все чистые виртуальные методы родительского абстрактного класса, то он сам становится абстрактным

– на абстрактный класс можно объявлять указатели или ссылки, с помощью которых затем поддерживается полиморфизм во время исполнения.

Множественное наследование

В языке C++ один класс может наследовать поля и методы двух и более классов одновременно. Для этого после описания класса необходимо перечислить все родительские классы через запятую. Общая форма множественного наследования имеет вид:

```
class дочерний_класс: [модификатор] родительский_класс_1,
                      [модификатор] родительский_класс_2
{
};
```

В следующем примере класс Z наследуется от двух классов X и Y:

Пример: множественное наследование 1

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

```
class X
{
   protected:
      int a;
   public:
      void make_a(int i)
      {
         this->a = i;
      }
};
class Y
{
   protected:
      int b;
   public:
      void make_b(int i)
      {
         this->b = i;
      }
};
// Z наследует как от X, так и от Y
class Z: public X, public Y
{
   public:
      int make_ab()
      {
         return this->a*this->b;
      }
};
int main()
{
   Z i;
   i.make_a(10);
   i.make_b(12);
   printf("%d", i .make_ab());
   return 0;
}
```

Поскольку класс Z наследует оба класса X и Y, то он имеет доступ к публичным и защищенным членам обоих классов X и Y.

В предыдущем примере ни один из классов не содержал конструкторов. Однако ситуация становится более сложной, когда базовый класс содержит конструктор. Изменим предыдущий пример таким образом, чтобы классы X, Y и Z содержали конструкторы.

Пример: множественное наследование 2

```
#include <iostream.h>
class X
{
   protected:
      int a;
   public:
   X()
   {
      this->a = 10;
      printf("Initializing X\n");
   }
};
class Y
{
   protected:
      int b;
   public:
```

```
   Y()
   {
      printf("Initializing Y\n");
      this->b = 20;
   }
};
// Z наследует как от X, так и от Y
class Z: public X, public Y
{
   public:
      Z()
      {
         printf("Initializing Z\n");
      }
      int make_ab()
      {
         return this->a*this->b;
      }
};
int main()
{
   Z i;
   printf("%d", i.make_ab());
   return 0;
}
```

Программа выдаст на экран следующий результат:

```
Initializing X
Initializing Y
Initializing Z
200
```

Следует обратить внимание, что конструкторы базовых классов вызываются в том порядке, в котором они указаны в списке при объявлении класса Z.

В общем случае, когда используется список родительских классов, их конструкторы вызываются слева направо. Деструкторы вызываются в обратном порядке - справа налево.

Следует отметить, что множественное наследование в языке C++ можно использовать, только если это не создает конфликтных ситуаций, описанных в разделе «Наследование классов».

ЛЕКЦИЯ 5.
ПЕРЕГРУЗКА ОПЕРАЦИЙ

Основные понятия перегрузки операций

Перегрузка операций позволяет определить действия для объектов в выражениях. Для этого разработчик определяет метод в классе, который будет вызван во время выполнения той или иной операции. Общая форма записи при перегрузке операций выглядит следующим образом:

```
<имя_класса> operator op(<имя_класса> a, < имя_класса> b)
```

В C++ существует 40 операций, которые можно перегрузить.

Нельзя перегрузить следующие операции: «.», «.*», «?:», «::», «sizeof», «#», «##».

Как известно, операции бывают *унарные* и *бинарные*. Унарные операции требуют одного аргумента, бинарные - двух. Например, операция «++» является унарной, в то время как операция «+» является бинарной.

Примечание: тернарная операция «?:» требует трех аргументов, однако она неперегружаема.

Существует две возможности определения операций:

– определение операции как метода класса;

– определение операции как дружественной функции.

Если перегруженная операция реализована как метод класса, то ее первым операндом является текущий объект, доступ к которому осуществляется через *this*. Поэтому такие операции при перегрузке в общем случае имеют на один аргумент меньше, чем это предусмотрено для соответствующей операции. Таблица ниже показывает общее число аргументов, необходимых для реализации перегрузки операций.

Число аргументов		**Операция**	
		унарная	**бинарная**
Реализация в виде	**метода класса**	0	1
	дружественной функции	1	2

C++ «не понимает» семантики перегруженного оператора. C++ не может выводить сложные операторы из простых. Нельзя изменять синтаксис перегруженных операций. Нельзя вводить новые операторы, а можно использовать только разрешенные 40.

Пример: Класс для работы с комплексными числами (определение операции как дружественной функции)

```
//f1.h
class Complex
{
   private:
      float real, imag;
   public:
      Complex(float aReal, float aImag): real(aReal), imag(aImag)
      {
      }
      Complex()
      {
         this->real = this->imag = 0.0;
      }
      Complex(float aReal): real(aReal), imag(0.0)
      {
      }
      float GetReal()
      {
         return this->real;
      }
      float GetImag()
      {
         return this->imag;
      }
      friend Complex operator+(Complex &c1, Complex &c2);
      friend Complex operator-(Complex &c1, Complex &c2);
```

```
};
Complex operator+(Complex &c1, Complex &c2)
{
   return Complex(c1.real + c2.real, c1.imag + c2.imag);
}
Complex operator-(Complex &c1, Complex &c2)
{
   return Complex(c1.real - c2.real, c1.imag - c2.imag);
}

//f1.cpp
#include <iostream.h>
#include "f1.h"
void main()
{
   Complex a(1.0,2.0), b(3.0,4.0);
   a = a+b;
   printf("%5.2f+i%5.2f\n", a.GetReal(), a.GetImag());
}
```

Пример: Класс для работы с комплексными числами (определение операции как метода класса)

```
//f1.h
class Complex
{
   private:
      float real, imag;
   public:
      Complex(float aReal, float aImag): real(aReal), imag(aImag)
      {
      }
      Complex()
      {
         this->real = this->imag = 0.0;
      }
      Complex(float aReal): real(aReal), imag(0.0)
      {
      }
      float GetReal()
      {
         return this->real;
      }
      float GetImag()
      {
         return this->imag;
      }
      void operator+(Complex &c1);
      void operator-(Complex &c1);
};
//f1.cpp
void Complex::operator +(Complex &c1)
{
   this->real+=c1.real;
   this->imag+=c1.imag;
}
void Complex::operator -(Complex &c1)
{
   this->real-=c1.real;
   this->imag-=c1.imag;
}
// main.cpp
#include <iostream.h>
#include "f1.h"
void main()
```

```
{
   Complex a(1.0,2.0), b(3.0,4.0);
   a = a+b;
   printf("%5.2f+i%5.2f\n", a.GetReal(), a.GetImag());
}
```

ЛЕКЦИЯ 6.
ПОТОКИ ДАННЫХ

Общее представление о потоках данных

В языке C++ нет средств для ввода-вывода. Эти средства можно просто воссоздать на самом языке.

В языке C++ производится ввод-вывод потоков (*streams*) данных. Поток данных представляет собой последовательность байтов. В операциях ввода байты пересылаются от устройства (например, от клавиатуры, дисковода или соединения сети) в оперативную память. При выводе байты пересылаются из оперативной памяти на устройство.

Чтение данных из потока называется *извлечением*, вывод в поток – *помещением*, или *включением*.

Основная задача потоковых средств ввода-вывода - это процесс преобразования объектов определенного типа в последовательность символов в битовом их представлении и наоборот. Такая схема ввода-вывода является универсальной и с одинаковой легкостью позволяет работать с любым периферийным устройством.

Обмен с потоком для увеличения скорости передачи данных производится через специальную область оперативной памяти – буфер. Фактическая передача данных выполняется при выводе после заполнения буфера, а при вводе – если буфер исчерпан.

По направлению обмена потоки можно разделить на *входные* (данные вводятся в память), *выходные* (данные выводятся из памяти) и *двунаправленные* (допускающие как извлечение, так и включение).

По виду устройств, с которыми работает поток, можно разделить потоки на *стандартные*, *файловые* и *строковые*.

Стандартные потоки предназначены для передачи данных от клавиатуры и на экран дисплея, *файловые потоки* – для обмена информацией с файлами на внешних носителях данных, *строковые потоки* – для работы с массивами символов в оперативной памяти.

Язык C++ предоставляет возможности для ввода-вывода как *неформатированных* так и *форматированных* данных. При вводе-выводе неформатированных данных каждый байт является самостоятельным элементом данных. Передача неформатированных данных позволяет осуществлять пересылку больших по объему потоков ввода-вывода с высокой скоростью. Недостаток программирования неформатированного потокового ввода-вывода состоит в его трудоемкости.

При форматированном потоковом вводе-выводе байты группируются в значащие элементы данных, например, целые числа, символы, строки,

вещественные числа, а также данные типов, определенных пользователем. Такой способ потокового ввода-вывода неэффективен только для файлов очень большого объема, однако является более удобным для работы.

Заголовочные файлы библиотеки iostream

Заголовочный файл <ios.h> содержит описание базового класса потоков ввода-вывода.

Заголовочный файл <iosfwd.h> содержит предварительные объявления средств ввода-вывода.

Библиотека <iostream.h> потокового ввода-вывода реализует строгий типовой и эффективный способ символьного ввода и вывода целых, вещественных чисел и символьных строк. Также библиотека является базой для расширения, рассчитанного на работу с пользовательскими типами данных.

Заголовочный файл <iostream.h> определяет интерфейс потоковой библиотеки. <iostream.h> определяет полный набор средств.

Заголовочный файл <iostream.h> включает объекты cin, cout, cerr и clog. cin соответствует стандартному потоку ввода, cout – стандартному потоку вывода, cerr – небуферизованному стандартному потоку вывода сообщений об ошибках, clog – буферизованному стандартному потоку вывода сообщений о работе программы. Для этих объектов предусмотрены возможности для форматированного и для неформатированного ввода-вывода.

Заголовочный файл <istream.h> содержит описание шаблона потока ввода.

Заголовочный файл <ostream.h> содержит описание шаблона потока вывода.

Заголовочный файл <iomanip.h> содержит информацию, необходимую для обработки форматированного ввода-вывода при помощи параметризованных манипуляторов потока.

Заголовочный файл <fstream.h> содержит информацию для проведения операций с файлами.

Заголовочный файл <sstream.h> содержит описание потоков ввода-вывода в строки.

Заголовочный файл <streambuf.h> содержит описание функций буферизации ввода-вывода.

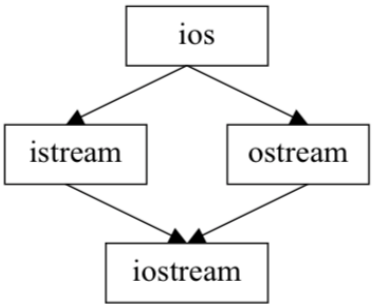
Также в составе библиотеки есть заголовочные файлы <amstream.h>, <austream.h>, <ddstream.h> и <mmstream.h>, который содержат информацию, необходимую для управления специализированными устройствами, предназначенными, например, для ввода-вывода аудио- и видеоданных.

Классы и объекты потоков ввода-вывода

Библиотека iostream содержит много классов для обработки операций ввода-вывода. Например, класс istream поддерживает операции по вводу потоков, класс ostream поддерживает операции по выводу объектов, класс iostream поддерживает оба типа операций: и ввод, и вывод потоков.

Класс istream и класс ostream являются прямыми потомками класса ios. Класс

iostream является производным классом множественного наследования классов istream и ostream. Иерархию наследования можно представить так:



Операции «поместить в поток» и «взять из потока» – это перегруженные операции сдвига влево (<<) и вправо (>>). Эти операции применяются к объектам стандартных потоков cin, cout, cerr и clog.

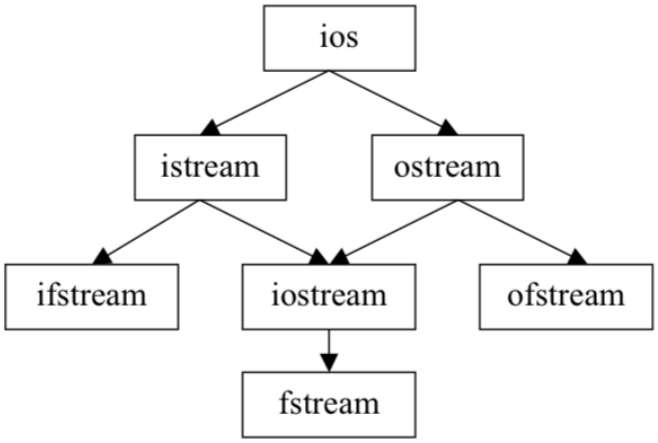
При обработке файлов C++ используются следующие классы:

– класс ifstream, который выполняет операции ввода из файлов;

– класс ofstream, который выполняет операции вывода в файлы;

– класс fstream, который выполняет операции ввода-вывода файлов.

Класс ifstream наследует класс istream, класс ofstream наследует класс ostream, а класс fstream - класс iostream. Иерархия классов потоков ввода-вывода с ключевыми классами обработки файлов выглядит так:



Пример использования объекта cerr:

```
cerr << "x = " << x << endl;
```

Здесь cerr обозначает стандартный поток ошибок. Так, если переменная x имеет тип int и значением 123, то приведенный оператор выдаст

```
x = 123
```

Вывод встроенных типов

Для управления выводом встроенных типов определяется класс ostream с операцией << (вывести):

```
class ostream:public virtual ios
{
   public:
   ***
   ostream& operator<<(const char*);
   ostream& operator<<(const char);
```

```
   ostream& operator<<(short i);
   ostream& operator<<(int);
   ostream& operator<<(long);
   ostream& operator<<(double);
   ostream& operator<<(const void*);
   ***
};
```

Функция operator<< возвращает ссылку на класс ostream, из которого она вызывалась, чтобы к ней можно было применить еще раз operator<<. Это дает возможность составлять цепочки подобных операторов. Данная идея используется во многих популярных фреймворках, например JQuery.

Если несколько объектов выводятся с помощью одного оператора вывода, то они будут выдаваться в естественном порядке: слева-направо.

Функция `ostream::operator<<(const void*)` напечатает значение указателя в такой записи, которая более подходит для используемой системы адресации. Так, например, программа:

```
main()
{
   int i = 0;
   int* p = new int(1);
   cout << "local " << &i<< ", free store " << p << '\n';
}
```

выведет, например,

```
local 0x0066FDF4, free store 0x00790DA0
```

Ввод встроенных типов

Для управления вводом встроенных типов определяется класс istream с операцией >> (ввести):

```
class istream:public virtual ios
{
   public:
   ***
   istream& operator>>(char*);
   istream& operator>>(char);
   istream& operator>>(short&);
   istream& operator>>(int&);
   istream& operator>>(long&);
   istream& operator>>(float&);
   istream& operator>>(double&);
   ***
};
```

Как и в предыдущем случае, функция operator>> возвращает ссылку на класс istream, из которого она вызывалась, чтобы к ней можно было применить еще раз operator>>, т.е. возможность конструирования цепочек сохраняется.

Для контроля соответствия количества введенных символов и объема зарезервированной памяти можно использовать функцию *get*.

```
class istream:public virtual ios
{
   public:
   ***
   istream& get(char& c);
   istream& get(char* p, int n, char ='n');
   istream& getline(char *, int, char ='\n');
};
```

Эти функции предназначены для таких операций ввода, когда не делается никаких предположений о вводимых символах.

Функция `istream::get(char&)` вводит один символ в свой параметр.

Поэтому программу посимвольного копирования можно написать так:

```
int main()
{
   char c;
   while (cin.get(c)) cout.put(c);
}
```

Функция с тремя параметрами `istream::get()` вводит в символьный вектор не менее *n* символов, начиная с адреса *p*. При всяком обращении к `get()` все символы, помещенные в буфер (если они были), завершаются 0, поэтому если второй параметр равен *n*, то введено не более *n-1* символов. Третий параметр определяет символ, завершающий ввод. Типичное использование функции `get()` с тремя параметрами сводится к чтению строки в буфер заданного размера для ее дальнейшего разбора, например, так:

```
void f()
{
   char buf[100];
   cin >> buf;               // подозрительно
   cin.get(buf,100,'\n');    // надежно
   ***
}
```

Операция `cin>>buf` может работать некорректно, поскольку строка из более чем 99 символов переполнит буфер.

Функция *get* без аргументов вводит одиночный символ из указанного потока и возвращает это символ в качестве значения вызова функции. Этот вариант функции *get* возвращает EOF, когда в потоке встречается признак конца файла.

Пример: использование функций *get*, *put* и *eof*.

```
int main()
{
   char c;
   cout<<"Перед вводом cin.eof равен: "<<cin.eof()<<endl;
   while ((c=cin.get())!=EOF)
      cout.put(c);
   cout<<"После ввода cin.eof равен: "<<cin.eof()<<endl;
   return 0;
}
```

Функция *getline* действует подобно функции *get*, вводящей строку, но, в отличие от функции *get*, функция *getline* удаляет символ-ограничитель из потока и не сохраняет его в символьном массиве.

Пример: символьный ввод функцией getline.

```
int main()
{
   const int SIZE=80;
   char buff[SIZE];
   cout<<"Vvedite predlozhenie\n";
   cin.getline(buff,SIZE);
   cout<<"Vvedennoye predlozhenie: \n"<<buff<<endl;
   return 0;
}
```

Функция *ignore* пропускает заданное число символов (по умолчанию один символ) или завершает свою работу при обнаружении заданного ограничителя. По

умолчанию символом-ограничителем является *EOF*, который заставляет функцию *ignore* пропускать символы до конца файла.

Функция *putback* возвращает обратно в этот поток предыдущий символ, полученный из входного потока с помощью *get*. Функция полезна для приложений, которые просматривают входной поток с целью поиска записи, начинающейся с заданного символа. Когда этот символ введен, приложение возвращает его в поток, так что он может быть включен в те данные, которые будут вводиться.

Функция *peek* возвращает очередной символ из входного потока, но не удаляет этот символ из него.

Неформатированный ввод-вывод

Неформатированный ввод-вывод выполняется с помощью функций *read* и *write*. Каждая из них вводит или выводит некоторое число байтов в символьный массив в памяти или из него. Эти байты не подвергаются какому-либо форматированию.

Например, вызов

```
char buff[25]="ABCDEFGHIJKLMNOPRSTUVWXYZ";
cout.write(buff,10);
```

или

```
cout.write("ABCDEFGHIJKLMNOPRSTUVWXYZ",10);
```

выводит первые 10 байтов символьного массива.

Функция *read* вводит в символьный массив указанное число символов. Если считывается меньшее количество символов, то устанавливается флаг *failbit*.

Функция *gcount* сообщает о количестве символов, прочитанных последней операцией ввода.

Пример: работа функций неформатированного ввода-вывода

```
int main()
{
   const int SIZE=80;
   char buff[SIZE];
   cout<<"Vvedite predlozhenie:";
   cin.read(buff,20);
   cout<<"Vvedennoye predlozhenie:";
   cout.write(buff,cin.gcount());
   return 0;
}
```

Форматирование данных

В потоковых классах форматирование выполняется тремя способами – с помощью манипуляторов, форматирующих методов и флагов.

Манипуляторы потоков

В языке C++ есть возможность использовать манипуляторы потоков, которые решают задачи форматирования. Манипуляторы потоков позволяют выполнить следующие операции: задание ширины полей, задание точности, установку и сброс

флагов формата, задание заполняющего символа полей, сброс потоков, вставку в выходной поток символа новой строки и сброс потока, вставку нулевого символа в выходной поток и пропуск символов разделителей во входном потоке.

Манипуляторы потоков, задающие основание чисел и вывод спецсимволов.

Для указания основания вывода чисел используются следующие манипуляторы без параметров: *dec* – устанавливает вывод десятичных чисел; *oct* – устанавливает вывод чисел в восьмеричной системе счисления; *hex* –устанавливает вывод чисел в шестнадцатеричной системе счисления.

Основание выводимых чисел можно также изменить с помощью манипулятора *setbase*. Этот манипулятор принимает целый параметр со значениями 10, 16 или 8. Так как манипулятор *setbase* принимает параметр, он называется *параметризованным манипулятором*. *Применение параметризованных манипуляторов требует подключения заголовочного файла iomanip.h.*

Основание потока является установленным до тех пор, пока оно не будет изменено явным образом.

Пример: использование манипуляторов *hex*, *dec*, *oct* и *setbase* для задания основания выводимых чисел.

```
#include <iostream.h>
#include <iomanip.h>
int main()
{
   int n;
   cout<<"Vvedite chislo: ";
   cin>>n;                                    // 60
   cout<<"16 format: "<<hex<<n<<endl;         // 3c
   cout<<"10 format: "<<dec<<n<<endl;         // 60
   cout<<"8 format: "<<oct<<n<<endl;          // 74
   cout<<"10 format: "<<setbase(10)<<n<<endl; // 60
   return 0;
}
```

Для вывода специальных символов используют следующие манипуляторы:

– *ws* – извлечение пробельных символов;

– *endl* – вставка символа новой строки и очистка потока;

– *ends* – вставка оконечного пустого символа в строку;

– *flush* – сброс на диск и очистка ostream;

– *setfill*(int c) – установка символа-заполнителя в с.

Пример: использование манипуляторов setfill, endl для управления форматированием и вывода спецсимволов.

```
#include <iostream.h>
#include <iomanip.h>
int main()
{
   char str[50];
   cin>>str;                                  // 0123
   cout<<setfill('#')<<setw(10)<<str<<endl; // ######0123
   return 0;
}
```

Примечание: манипулятор setw устанавливает ширину поля для вывода. Он разбирается далее.

Манипуляторы потоков, задающие формат вывода вещественного числа

Число печатаемых разрядов справа от десятичной точки для вещественного

числа можно задать с помощью манипулятора потока *setprecision*.

Пример: использование манипулятора *setprecision* для задания точности вывода вещественных чисел.

```
#include <iostream.h>
#include <iomanip.h>
int main()
{
   cout<<setprecision(4)<<sqrt(3.0)<<endl; // 1.7321
   return 0;
}
```

Манипулятор *setw* устанавливают ширину поля. Под шириной поля понимается число символьных позиций, в которые значение будет выведено, или число символов, которые будут выведены. Если обрабатываемые значения имеют меньше символов, чем заданная ширина поля, то для заполнения лишних позиций используются заполняющие символы. Если число символов в обрабатываемом значении больше, чем заданная ширина поля, то лишние символы не отсекаются.

Установка ширины поля применяется только к текущей операции «поместить в поток» или «взять из потока». После выполнения этих операций ширина поля устанавливается неявным образом на 0. Такая установка означает, что поле для представления выходных данных будет необходимой ширины.

Пример: использование манипулятора *setw* для задания ширины поля.

```
#include <iostream.h>
#include <iomanip.h>
int main()
{
   char str[50];
   cin>>setw(21)>>str;  // 012345678901234567890
   cout<<setw(10)<<str; // 0123456789
   return 0;
}
```

Замечание: для того чтобы обеспечить размещение во входной строке нулевого символа, считывается на один символ меньше, чем установленная ширина поля.

Создание собственных манипуляторов потока

Программист может также создавать собственные манипуляторы потоков так, так показано в примере.

Пример: создание и использование собственного манипулятора потока (tab).

```
#include <iostream.h>
#include <iomanip.h>
ostream & tab(ostream &output)
{
   return output<<'\t';
}
int main()
{
   char *str = 'ABCD';
   cout<<tab<<tab<<str<<endl;
   return 0;
}
```

Форматирующие методы потоков

Устанавливать точность вывода дробных чисел можно также при помощи

метода *precision*. Вызов этого метода действует для последующих операций вывода до тех пор, пока не будет произведена следующая установка точности.

В классе ios объявлено два перегруженных метода *precision*.

– *int ios::precision()* – возвращает значение точности представления при выводе вещественных чисел;

– *int ios::precision(int)* – устанавливает значение точности представления при выводе вещественных чисел, возвращает старое значение точности.

Пример: использование метода *precision* для задания точности вывода вещественных чисел.

```
#include <iostream.h>
#include <iomanip.h>
int main()
{
   cout.precision(10);
   cout<<sqrt(3.0)<<endl; // 1.732050808
   return 0;
}
```

Для установки ширины поля также может использоваться метод *width* класса ios. Метод *width* имеет две реализации:

– *int ios::width()* – возвращает значение ширины поля ввода/вывода;

– *int ios::width(int)* – возвращает значение ширины поля ввода/вывода, возвращает старое значение ширины.

Пример: использование метода *width* для задания ширины поля.

```
#include <iostream.h>
#include <iomanip.h>
int main()
{
   char str[50];
   cin.width(15);
   cin>>str;          // 012345678901234567890
   cout.width(10);
   cout<<str<<endl;   // 01234567890123
   return 0;
}
```

Замечание: для того чтобы обеспечить размещение во входной строке нулевого символа, считывается на один символ меньше, чем установленная ширина поля.

Установить символ-заполнитель перед выводом данных в поток можно с помощью двух перегруженных методов класса ios:

– *char ios::fill()* – возвращает текущий символ заполнения;

– *char ios::fill(char)* – устанавливает значение текущего символа заполнения; возвращает значение старого.

Пример: использование метода *fill.*

```
#include <iostream.h>
#include <iomanip.h>
int main()
{
   cout.fill('#');
   cout.width(10);
   int n=12345678;
   cout<<n<<endl;    // ##12345678
   return 0;
}
```

Флаги форматирования в потоках

Различные флаги формата (*format flags*) задают виды форматирования, которые выполняются во время операций ввода–вывода. В таблице приведены различные флаги формата.

Для установки и считывания значений флагов можно использовать методы класса ios:

– *long ios::flags()* – возвращает текущие флаги.

– *long ios::flags(long)* – присваивает новые значения флагам и возвращает старые.

– *long ios::setf(long, long)* – перед установкой некоторых флагов позволяет сбросить флаги, которые не могут быть установлены одновременно с ними. В качестве первого параметра передается устанавливаемый флаг, а качестве второго параметра передается одна из следующих статических констант класса ios:

– *adjustfield* (*left* | *right* | *internal*)

– *basefield* (*dec* | *oct* | *hex*)

– *floatfield* (*scientific* | *fixed*)

– *long ios::setf(long)* – устанавливает флаги формата, заданные в качестве аргумента и возвращает предыдущие установки этих флагов.

– *long ios::unsetf(long)* – сбрасывает указанные флаги формата и возвращает предыдущие их значения.

Для работы с флагами можно также воспользоваться манипуляторами с параметрами:

– *setiosflags(long)* – устанавливает флаги формата, указанные в аргументе.

– *resetiosflags(long)* – сбрасывает флаги формата, указанные в аргументе.

Описание основных флагов приводится в следующей таблице:

Флаг	**Описание**
ios::skipws	При извлечении пробельные символы игнорируются
ios::left	Выравнивание по левому краю поля
ios::right	Выравнивание по правому краю поля
ios::internal	Знак числа выводится по левому краю, число – по правому
ios::dec	Десятичная система счисления
ios::oct	Восьмеричная система счисления
ios::hex	Шестнадцатеричная система счисления
ios::showbase;	Выводится основание системы счисления (0x – шестнадцатеричная, 0 – восьмеричная)
ios::showpoint	Определяет, что вещественные числа должны выводиться с десятичной точкой. Этот флаг обычно используется для обеспечения определенного числа цифр справа от десятичной точки

ios::uppercase	Предписывает использовать прописные буквы в шестнадцатеричных числах и экспоненциальной форме вещественных чисел.
ios::showpos	Определяет, что числа должны выводиться со знаком
ios::scientific	Определяет вывод вещественных чисел в экспоненциальной форме
ios::fixed	Определяет вывод вещественных чисел в форме с фиксированной точкой
ios::showpoint	Определяет, что вещественные числа должны выводиться с десятичной точкой

Пример: использование флагов состояния формата.

```
#include <iostream.h>
#include <iomanip.h>
int main()
{
   // устанавливаем флаг вывода вещественных чисел
   cout.setf(ios::showpoint);
   cout<<9.0000<<endl;                                  // 9.00000
   cout<<9.9000<<endl;                                  // 9.90000
   // снимаем флаг вывода вещественных чисел
   cout.unsetf(ios::showpoint);
   cout<<9.0000<<endl;                                  // 9
   cout<<9.9000<<endl;                                  // 9.9
   // устанавливаем флаг вывода целого числа в шестнадцатеричном виде
   cout.setf(ios::hex, ios::basefield);
   int x=12345;
   cout<<x;                                             // 3039
   // снимаем флаг вывода целого числа в шестнадцатеричном виде
   cout.unsetf(ios::hex);
   // устанавливаем флаг вывода символов по левому краю поля
   cout<<endl<<setw(30)<<setiosflags(ios::left)<<x;  // 12345
   // снимаем флаг вывода символов по левому краю поля
   cout<<resetiosflags(ios::left)<<endl;
   // устанавливаем флаг вывода знака числа и флаг вывода знака числа,
   // прижатого к левому краю, и самого числа, прижатого к правому краю
   cout.flags(ios::showpos | ios::internal);
   cout<<setw(10)<<setfill('#')<<x<<endl;            // +####12345
   // устанавливаем флаг вывода символов по левому краю поля
   cout.flags(ios::left);
   cout<<setw(10)<<setfill('%')<<x<<endl;            // +12345%%%%
   // устанавливаем флаг специфического отображения шестнадцатеричных
   // и восьмеричных чисел
   cout.setf(ios::showbase);
   cout<<oct<<x<<endl;                                  //030071
   cout<<hex<<x<<endl;                                  // 0x3039
   cout<<dec<<x<<endl;                                  // +12345
   double y=9000, z=0.0009;
   // снимаем флаг вывода знака числа
   cout.unsetf(ios::showpos);
   // устанавливаем флаг вывода вещественных чисел в экспоненциальной
   // форме и флаг вывода прописных в представлении чисел
   cout.setf(ios::scientific | ios::uppercase);
   cout<<y<<" "<<z<<endl;                    // 9.00000E+003 9.000000E-004
   // снимаем флаг вывода прописных в представлении чисел
   cout.unsetf(ios::uppercase);
   cout<<y<<" "<<z<<endl;                    // 9.00000e+003 9.000000e-004
   // устанавливаем флаг вывода вещественных чисел в виде
   // числа с десятичной точкой
   cout.setf(ios::fixed, ios::floatfield);
```

```
    cout<<y<<" "<<z<<endl;                    // 9000.000000 0.000900
    // устанавливаем флаг вывода целого числа в шестнадцатеричном виде
    // и флаг вывода прописных в представлении чисел
    cout.setf(ios::hex | ios::uppercase);
    cout<<31<<endl;                           // 0X1F
    return 0;
}
```

Связывание выходного и входного потоков

Интерактивные приложения обычно включают класс *istream* для ввода и класс *ostream* для вывода. Очевидно, что в таких приложениях приглашение на ввод должно появляться до осуществления операции ввода. В языке C++ есть метод *tie*, который выполняет синхронизацию (связывание) выполнения операций над потоками *istream* и *ostream*. Этот метод гарантирует, что вывод появится раньше последующего ввода. В C++ происходит автоматическое связывание объектов *cin* и *cout*:

```
cin.tie(&cout);
```

При программировании следует связывать другие пары потоков классов *istream* и *ostream*.

Например, для того чтобы отвязать входной поток InputStream от выходного:

```
InputStream.tie(0);
```

Файлы и потоки

В C++ каждый файл рассматривается как последовательность байтов. Каждый файл завершается маркером конца файла (*end-of-file marker*). Когда файл открывается, то создается объект и с этим объектом связывается поток. Для обработки файлов в C++ должны быть включены в программу заголовочные файлы <*iostream.h*> и <*fstream.h*>. Файл <*fstream.h*> включает определения классов потоков *ifstream* (для ввода из файла), *ofstream* (для вывода в файл) и *fstream* (для ввода-вывода файлов). Файлы открываются путем создания объектов этих классов потоков. Так эти классы потоков являются дочерними от классов *istream*, *ostream* и *iostream*, то они могут использовать методы, операции и манипуляторы, определенные в их базовых классах.

Создание файла последовательного доступа

Так как C++ не предписывает файлу никакой структуры, то программист должен задавать структуру файлов в соответствии с требованиями прикладных программ.

Пример: занесение данных о студенте, его группе и среднем балле в текстовый файл.

```
#include <iostream.h>
#include <fstream.h>
int main()
{
   /* Так как файл открывается для ввода, то создается объект
      класса ofstream. Конструктору этого класса передается
      два параметра – имя файла и режим открытия файла. */
```

```
    ofstream outGroupFile("student.txt",ios::out);
    /* Для проверки того, успешно ли открылся файл,
       используется перегруженный метод opearator ! класса ios.
       Если файл открыт, он возвращает ненулевое значение. */
    if(!outGroupFile)
    {
       cerr<<"File not open"<<endl; return 0;
    }
    char name[30], group[10]; double AverMark;
    cout<<"?: ";
    while(cin>>name>>group>>AverMark)
    {
       /* Условие в заголовке оператора while автоматически
          вызывает перегруженный метод operator void*
          класса ios. Эта функция превращает поток в
          указатель, который можно проверить на равенство 0.
          Ввод признака конца файла (Ctrl+Z) установит бит
          failbit для cin, после проверки которого метод operator
          void* вернет 0, и условие цикла while станет ложным.
          Таким образом, метод operator void* можно использовать
          для проверки конца файла в объекте ввода вместо явного
          вызова метода eof. */
       outGroupFile<<name<<' '<<group<<' '<<AverMark<<endl;
       /* Информация записывается в файл "student.txt" с помощью объекта
          outGroupFile, связанного с этим файлом, и операции «поместить
          в поток» */
       cout<<"?: ";
    }
    outGroupFile.close();
    /* Метод close() осуществляет явный вызов деструктора
       объекта outGroupFile, закрывающего файл. */
    return 0;
}
```

Таблица. Режимы открытия файлов

Режим	**Описание**
ios::app	Записать данные в конец файла без модификации имеющихся данных в файле.
ios::ate	Переместиться в конец исходного открытого файла. Данные могут быть записаны в любое место файла.
ios::in	Открыть файл для ввода.
ios::out	Открыть файл для вывода. Если в файле есть данные, то они уничтожаются. Если файла не существует, он создается.
ios::trunc	Уничтожать содержимое файла, если он существует.
ios::binary	Открыть файл для двоичного ввода или вывода.

Таблица. Соответствие между режимами открытия файла класса *ios* и режимами открытия файла, описанными в *<stdio.h>*

Комбинация флагов ios					**Эквивалент stdio**
binary	**in**	**out**	**trunc**	**app**	
		+			“w”
		+		+	“a”
		+	+		“w”
	+				“r”

	+	+			“r+”
	+	+	+		“w+”
+		+			“wb”
+		+		+	“ab”
+		+	+		“wb”
+	+				“rb”
+	+	+			“r+b”
+	+	+	+		“w+b”

Замечание: кроме как с помощью конструктора, открыть файл можно с помощью метода *open*, имеющего такие же параметры как и конструктор. Например,

```
ofstream outGroupFile;
outGroupFile.open("student.txt",ios::out);
```

Пример: чтение данных из файла последовательного доступа и вывод их на экран.

```
#include <iostream.h>
#include <fstream.h>
#include <iomanip.h>
void OutputLine(const char* name, const char *group, double mark)
{
   cout<<setiosflags(ios::left)<<setw(20)<<name
   <<setw(10)<<group<<setw(8)<<setprecision(2)
   <<resetiosflags(ios::left)<<mark<<endl;
}
int main()
{
   ifstream inGroupFile;
   /* Как и класс ofstream класс ifstream имеет конструктор с двумя
      аргументами – именем файла и режимом его открытия. Если режим
      открытия файла не указан, то значением по умолчанию является
      ios::in. Таким образом, можно создать объект класса ifstream и
      передать ему один параметр – имя файла. Например,
         ifstream inGroupFile("student.dat");
      Для открытия файла на чтение можно также воспользоваться созданием
      объекта класса ifstream с помощью конструктора без параметров с
      последующим вызовом для него метода open */
   inGroupFile.open("student.dat",ios::in);
   if(!inGroupFile)
   {
      /* К объекту класса ifstream, как и к объекту класса ofstream,
         можно применить перегруженный метод operator! для того, чтобы
         проверить, успешно ли открылся файл */
      cerr<<"File not open"<<endl;
      return 0;
   }
   char name[30], group[10]; double AverMark;
   cout<<setiosflags(ios::left)<<setw(20)<<"Name of student"
   <<setw(10)<<"Group"<<"Av. mark"<<endl
   <<setiosflags(ios::fixed|ios::showpoint);
   while(inGroupFile>>name>>group>>AverMark)
      /* В условии цикла while используется перегруженный метод operator
         void*() для чтения данных из файла. Этот метод возвращает 0,
         если будет достигнут конец файла. */
      OutputLine(name,group,AverMark);
      inGroupFile.close();
   return 0;
}
```

Методы позиционирования указателя в файле

Классы *istream* и *ostream* содержат методы для позиционирования указателя файла. Под указателем понимается порядковый номер следующего байта в файле, который должен быть считан или записан. Любой объект класса *istream* имеет указатель «*get*», который показывает номер в файле очередного вводимого байта. Любой объект класса *ostream* имеет указатель «*put*», который показывает номер в файле очередного выводимого байта.

Функция

```
istream& seekg(streamoff offset, ios::seek_dir origin);
```

при извлечении данных из файла перемещает указатель на *offset* байт от позиции, заданной параметром *origin*.

Функция

```
ostream& seekp(streamoff offset, ios::seek_dir origin);
```

во время чтения данных из файла перемещает указатель на *offset* байт от позиции, заданной параметром *origin*.

Параметр *offset* должен принимать целое значение.

Параметр *origin* представляет собой перечисление, которое имеет следующие значения:

– *ios::beg* – смещение от начала;

– *ios::cur* – смещение от текущей позиции;

– *ios::end* – смещение от конца.

Функции

```
ostream& seekp(streampos position);
istream& seekg(streampos position);
```

перемещают указатель от начала файла на *position* байт, соответственно, при чтении и извлечении данных из файла.

Функция

```
streampos tellg();
```

возвращает значение указателя «*get*» объекта *istream*.

Функция

```
streampos tellp();
```

возвращает значение указателя «*put*» объекта *ostream*. Например,

```
istream FileObject;
// позиционирование FileObject на n-й байт от начала файла
FileObject.seekg(n);
// позиционирование FileObject на n байтов вперед
FileObject.seekg(n,ios::cur);
// позиционирование FileObject на k-й байт от конца файла
FileObject.seekg(k,ios::end);
// позиционирование FileObject на конец файла
FileObject.seekg(0,ios::end);
// присваивание переменной location значения файлового указателя «get»
location=FileObject.tellg();
```

Файлы произвольного доступа

Файлы последовательного доступа не подходят для решения задач, требующих немедленного доступа к локализованной записи из большого объема данных (системы резервирования билетов, банковские системы, система

терминалов для производства платежей в месте совершения покупок, банковские автоматы). Для решения таких задач используются файлы произвольного доступа. Для создания файлов произвольного используется несколько методов. Наиболее простым из них является метод, при котором к записям в файле предъявляется требование одинаковой длины. В этом случае местоположение любой записи определяется функцией, зависящей от размера записи и ключа записи.

Данные могут быть вставлены в файл прямого доступа без разрушения других данных файла. Данные, которые в нем хранятся, могут быть изменены или удалены без перезаписи всего файла.

ЛЕКЦИЯ 7.
ШАБЛОННЫЕ ФУНКЦИИ И КЛАССЫ

Общие сведения о шаблонных функциях и классах

Шаблоны – одно из последних нововведений стандарта языка C++. Шаблоны дают возможность определять при помощи одного фрагмента кода целый набор взаимосвязанных функций (перегруженных), называемых *шаблонными функциями*, или набор связанных классов, называемых *шаблонными классами*.

Например, можно написать один шаблон функции сортировки массива, на основе которого C++ будет автоматически генерировать отдельные шаблонные функции, сортирующие массивы типов int, float, строк и т.д. Или достаточно описать один шаблон класса стеков, а затем C++ будет автоматически создавать отдельные шаблонные классы типа стек целых, стек вещественных чисел, стек строк и т.д.

Шаблонные функции

Шаблонные функции обычно используются для выполнения похожих операций над различными типами данных. Если для каждого типа данных должны выполняться идентичные операции, то оптимальным решением является использование шаблонных функций. При этом программист должен написать всего одну шаблонную функцию. Основываясь на типах аргументов, использованных при вызове этой функции, компилятор будет автоматически генерировать объектные коды функций, обрабатывающих каждый тип данных. В языке C эта задача выполнялась при помощи макросов, определяемых директивой препроцессора *#define*. Однако при выполнении макросов компилятор не выполняет проверку соответствия типов, из-за чего нередко возникают ошибки. Шаблонные функции, являясь таким же компактным решением, как и макросы, позволяют компилятору полностью контролировать соответствие типов.

Все описания шаблонных функций начинаются с ключевого слова *template*, за котором следует список формальных параметров, заключаемый в угловые скобки «<» и «>»; каждому формальному параметру должно предшествовать ключевое слово *class* или *typename*. Например,

```
template <class T>
```

или

```
template <typename ElementType>
```

или

```
template <class BorderType, class FillType>
```

Формальные параметры в описании шаблонной функции используются (наряду с параметрами встроенных типов или типов, определяемых пользователем) для определения типов параметров функции, типа возвращаемого функцией значения и типов переменных, объявляемых внутри функции. Далее, за этим заголовком, следует обычное описание функции. Ключевое слово *class* или *typename*, используемое в шаблоне функции при задании типов параметров, означает «любой встроенный тип или тип, определяемый пользователем».

Замечание: ошибкой является отсутствие ключевого слова *class* (или *typename*) перед каждым формальным параметром типа шаблона функции.

Пример: Описание шаблонной функции, инкрементирующей значение переменной произвольного типа.

```
template <class T>
T& inc_value(T &val)
{
   ++val;
   return val;
}
int main()
{
   int x = 0;
   x=(int)inc_value<int>(x);
   cout<<x<<endl;
   char c = 0;
   c=(char)inc_value<char>(c);
   cout<<c<<endl;
   return 0;
}
```

Из предыдущего примера видно, что инкрементирование может применяться также и к символьным переменным. В этом случае речь идет о увеличении ASCII кода символа.

Пример: Описание шаблонной функции, выводящей на экран массив произвольного типа.

```
template <class T1>
void PrintArray(const T1* array, const int count)
{
   for (int i=0; i<count; i++)
      cout<<array[i]<<" ";
   cout<<endl;
}
int main()
{
   const int aCount=5, bCount=7, cCount=6;
   int a[aCount] = {1,2,3,4,5};
   double b[bCount] = {1.1, 2.2, 3.3, 4.4, 5.5, 6.6, 7.7};
   char c[cCount]="HELLO";         // 6-я позиция для null
   cout<<"Array a:"<<endl;
   PrintArray<int>(a,aCount);      // шаблон для integer
   cout<<"Array b:"<<endl;
   PrintArray<double>(b,bCount);   // шаблон для double
   cout<<"Array c:"<<endl;
   PrintArray<char>(c,cCount);     // шаблон для char
   return 0;
```

```
}
```

В шаблонной функции *PrintArray* объявляется указатель на формальный параметр T1 для массива, который будет выводиться этой функцией.

В примере шаблонный механизм позволяет программисту избежать необходимости написания трех отдельных функций с прототипами:

```
void PrintArray(const int*, const int);
void PrintArray(const double*, const int);
void PrintArray(const char*, const int);
```

которые используют один и тот же код, кроме кода для типа T1.

T1 и T в предыдущих примерах называется *параметром типа*. Когда компилятор обнаруживает в тексте программы вызов функции *PrintArray*, он заменяет T1 во всей области определения шаблона на тип первого параметра функции *PrintArray* и C++ создает шаблонную функцию вывода массива указанного типа данных. После этого вновь созданная функция компилируется.

Шаблоны функций расширяют возможности многократного использования программного кода, но программа может создавать слишком много копий шаблонных функций и шаблонных классов. Для этих копий могут потребоваться значительные ресурсы памяти.

Перегрузка шаблонных функций

Шаблонные функции и перегрузка функций тесно связаны друг с другом. Все родственные функции, полученные из шаблона, имеют одно и то же имя; поэтому компилятор использует механизм перегрузки для того, чтобы обеспечить вызов соответствующей функции.

Сам шаблон функции может быть перегружен несколькими способами. Во–первых, можно определить другие шаблоны, имеющие то же самое имя функции, но различные наборы параметров. Например,

```
template <class T1>
void PrintArray(const T1* array, const int count)
{
   for (int i=0; i<count; i++)
      cout<<array[i]<<" ";
   cout<<endl;
}
template <class T1>
void PrintArray(const T1* array,
                const int lowSubscript,
                const int highSubscript)
{
   for (int i=lowSubscript; i<=highSubscript; i++)
      cout<<array[i]<<" ";
   cout<<endl;
}
int main()
{
   const int aCount=5, bCount=7, cCount=6;
   int a[aCount] = {1,2,3,4,5};
   double b[bCount] = {1.1, 2.2, 3.3, 4.4, 5.5, 6.6, 7.7};
   char c[cCount]="HELLO";        // 6-я позиция для null
   ...
   cout<<"Array a from 1 to 3:"<<endl;
   PrintArray<int>(a,0,2);        // шаблон для integer
   cout<<"Array b from 4 to 7:"<<endl;
```

```
   PrintArray<double>(b,3,6);     // шаблон для double
   cout<<"Array c from 3 to 5:"<<endl;
   PrintArray<char>(c,2,4);       // шаблон для character
   return 0;
}
```

Шаблон функции может быть также перегружен, если ввести другую нешаблонную функцию с тем же самым именем, но другим набором параметров. Например, шаблон функции *PrintArray* можно перегрузить версией нешаблонной функции, которая выводит массив символов в столбик.

```
template <class T1>
void PrintArray(const T1* array, const int count)
{
   ...
}
template <class T1>
void PrintArray(const T1* array,
                const int lowSubscript,
                const int highSubscript)
{
   ...
}
void PrintArray(char* array, const int count)
{
   for (int i=0; i<count; i++)
      cout<<array[i]<<endl;
}

int main()
{
   const int cCount=6;
   char c[cCount]="HELLO";        // 6-я позиция для null
   cout<<"Array c:"<<endl;
   PrintArray(c,cCount);
   ...
   return 0;
}
```

Компилятор действует по следующему алгоритму. Сначала он пытается найти и использовать функцию, которая точно соответствует по своему имени и типам параметров вызываемой функции. Если такая функция не найдена, то компилятор ищет шаблонную функцию, с помощью которой он может сгенерировать эту функцию. Если такая шаблонная функция обнаруживается, то компилятор ее генерирует.

Замечание: компилятор ищет шаблон, полностью соответствующий вызываемой функции по типу всех параметров; автоматическое преобразование типов не производится.

Шаблонные классы

С помощью шаблонных классов можно достаточно просто определить необходимый функционал для объектов, не привязанных к конкретному типу данных, например, создать контейнерные классы, списки, ассоциативные массивы и т.п. Кроме того, шаблонные классы позволяют определить сразу для целого семейства типов обобщенные функции, например, такие, как *sort* (сортировка).

Рассмотрим простой шаблонный класс *stack* (стек). Понятие «стек» не зависит

от типа данных, которые в него помещаются. Тип данных должен быть задан только тогда, когда приходит время создать стек «фактически». Поэтому достаточно создать некое общее описание понятия стека, выполняющего базовые операции, и на основе этого шаблонного класса создавать классы, являющиеся версиями для конкретных типов данных.

Пример: Шаблонный класс *stack* для хранения данных произвольного типа.

```
template<class T>
class stack
{
   private:
      T* data;
      int sz;
      int sp;
   public:
      stack(int s)
      {
         if (s<=0)
         {
            cout << "Error. Size must be > 0";
            this -> data = NULL;
         }
         else
         {
            this -> sz = s;
            this -> data = new T[s];
         }
         this -> sp=0;
      }
      ~stack()
      {
         delete[] data;
      }
      void push(T a)
      {
         if (this -> sp >= this -> sz)
         {
             cout << "Error. Stack overflow";
             return;
         }
         this -> data[this->sp] = a;
         this -> sp++;
      }
      T pop()
      {
         if (this -> sp > 0)
         {
             cout << "Error. Stack is empty";
             return;
         }
         T temp = this -> data[this->sp];
         this -> sp--;
         return temp;
      }
      int size() const
      {
         return this -> sp;
      }
};
```

Префикс *template<class T>* указывает, что описывается шаблон класса с параметром T, обозначающим тип данных классов семейства *Stack,* которые будут

создаваться на основе этого шаблонного класса.

Имя шаблонного класса, за которым следует тип, заключенный в угловые скобки <>, является именем конкретного класса, созданного на основе шаблонного класса, и его можно использовать как любой другой класс. Например, ниже определяется объект *sc* класса *stack<char>*:

```
stack<char> sc(100); // стек символов
```

Если не считать особую форму записи имени, класс *stack<char>* полностью эквивалентен классу определенному так:

```
class stackChar
{
   private:
      char* data;
      int sz;
      int sp;
   public:
      stack(int s)
      {
         if (s<=0)
         {
            cout << "Error. Size must be > 0";
            this -> data = NULL;
         }
         else
         {
            this -> sz = s;
            this -> data = new char[s];
         }
         this -> sp=0;
      }
      ~stack()
      {
         delete[] data;
      }
      void push(char a)
      {
         if (this -> sp >= this -> sz)
         {
             cout << "Error. Stack overflow";
             return;
         }
         this -> data[this->sp] = a;
         this -> sp++;
      }
      char pop()
      {
         if (this -> sp > 0)
         {
             cout << "Error. Stack is empty";
             return;
         }
         char temp = this -> data[this->sp];
         this -> sp--;
         return temp;
      }
      int size() const
      {
         return this -> sp;
      }
};
```

Имея определение шаблонного класса *stack*, можно следующим образом

определять и использовать различные стеки:

```
// Вспомогательный класс для хранения точек
class Point
{
   private:
      int x, y;
   public:
      Point(int x, int y)
      {
         this -> x = x;
         this -> y = y;
      }
      friend ostream& operator<<(ostream &output, Point &p);
};
ostream& operator<< (ostream &output, Point &p)
{
   output << "(" << p.x << "," << p.y << ") ";
   return output;
}
void main()
{
   // стек указателей на классы Point в статической памяти
   stack<Point*> sp(100);
   Point* p1 = new Point(10, 20);
   Point* p2 = new Point(30,40);
   sp.push(p1);
   sp.push(p2);
   // стек указателей на целые числа в динамической памяти
   stack<int>*p = new stack<int>(100);
   for(int i = 0; i<100; i++)
      p->push(i);
   // выведет 2, так как в первый стек добавлено 2 точки
   cout<<sp.size()<<endl;
   // выведет 100, так как во второй стек добавлено 100 чисел
   cout<<p->size()<<endl;
}
```

В случае внешней реализации методов шаблона классов *stack* сам шаблон определяется следующим образом:

```
template<class T>
class stack
{
   private:
      T* data;
      int sz;
      int sp;
   public:
      stack(int);
      ~stack();
      void push(T);
      T pop();
      int size() const;
};
```

В этом случае определение методов шаблона классов *stack* выполняется, как и для функций-членов обычных, нешаблонных классов. Подобные функции так же параметризируются типом, служащим параметром для их шаблонного класса, поэтому определяются они с помощью шаблона класса для функции. Если это происходит вне шаблонного класса, это надо делать явно:

```
template<class T> void stack<T>::push(T a)
{
   ...
```

```
}
template<class T> T stack<T>::pop()
{
   ...
}
template<class T> int stack<T>::size() const
{
   ...
}
template<class T> stack<T>::stack(int s)
{
   ...
}
template<class T> stack<T>::~stack()
{
   ...
}
```

Важно составлять определение шаблона классов таким образом, чтобы его зависимость от глобальных данных была минимальной. Дело в том, шаблон классов будет использоваться для порождения функций и классов на основе заранее неизвестного типа и в неизвестных контекстах. Практически любая, даже слабая зависимость от контекста может проявиться как проблема при отладке программы пользователем, который, вероятнее всего, не был создателем шаблона классов.

Рассмотрим еще один пример использования шаблонных классов - создание однонаправленного списка. Как и в предыдущем случае, шаблонный класс позволяет не привязываться к конкретному типу данных, элементы которого будут храниться в этом списке. Кроме того, однонаправленный список будет динамичным - ограничения на количество элементов в нем не будет.

Пример: Шаблонный класс для хранения данных произвольного типа, построенный по принципу однонаправленного списка.

```
// Вспомогательный класс для хранения точек
class Point
{
   private:
      int x, y;
   public:
      Point(int x, int y)
      {
         this -> x = x;
         this -> y = y;
      }
      friend ostream& operator<<(ostream &output, Point &p);
};
ostream& operator<< (ostream &output, Point &p)
{
   output << "(" << p.x << "," << p.y << ") ";
   return output;
}
// Основной шаблонный класс
template<class type>
class CListTemplate
{
   private:
      struct SNode
      {
         type *data;
         SNode *next;
```

```
   };
SNode *head, *end;
public:
   CListTemplate()
   {
      this -> head = this -> end = NULL;
   }
   void AddElem(type *d)
   {
      if(this -> head)
      {
         this -> end -> next = (SNode*)malloc(sizeof(SNode));
         this -> end = this -> end -> next;
      }
      else
      {
         this -> head = (SNode*)malloc(sizeof(SNode));
         this -> end = this -> head;
      }
      this -> end -> data = d;
      this -> end -> next = NULL;
   }
   bool RemoveElem(int index)
   {
      SNode *curr = this -> head;
      SNode *p1;
      if (index == 0)
      {
         p1 = this -> head;
         this -> head = this -> head -> next;
         delete p1;
         return true;
      }
      else
      {
         while (curr -> next && index > 1)
         {
            index--;
            curr = curr -> next;
         }
         if (curr -> next)
         {
             p1 = curr -> next;
             if (!curr -> next -> next)
                this -> end = curr;
             curr -> next = curr -> next -> next;
             delete p1;
             return true;
         }
      }
      return false;
   }
   void DisplayList()
   {
      SNode *curr = this -> head;
      while (curr)
      {
         cout << *(curr -> data) << " ";
         curr = curr -> next;
      }
      cout << endl;
   }
   void clear()
   {
```

```
            SNode *curr = this -> head;
            while(head)
            {
               curr=this -> head;
               this -> head=this -> head -> next;
               delete curr;
            }
         }
         ~CListTemplate(){
            clear();
         }
};
void main()
{
   CListTemplate<Point> c_point;
   CListTemplate<int> c_int;
   for (int i=0; i<10; i++)
   {
      Point* p = new Point(i * 10, i*10 + 1);
      c_point.AddElem(p);
      c_int.AddElem(new int(i));
   }
   c_point.DisplayList();
   c_point.RemoveElem(2);
   c_point.DisplayList();
   c_int.DisplayList();
   c_int.RemoveElem(3);
   c_int.DisplayList();
}
```

Шаблоны классов и нетиповые параметры

Шаблон класса *stack*, рассмотренный ранее, использовал только параметр типа в заголовке шаблона. Но в шаблоне имеется возможность использовать и нетиповые параметры. Например, в заголовок шаблона можно добавить нетиповой параметр *int elements*, который будет содержать количество элементов стека:

```
template <class T, int elements>
```

Тогда оператор

```
Stack <double, 100> s1;
```

задает шаблонный класс *Stack* с именем *s1*, состоящий из 100 элементов типа *double*.

Замечание: если можно определить размер класса-контейнера (такого как массив или стек) во время компиляции (например, при помощи нетипового параметра), это повысит скорость выполнения программы, устранив потери времени на динамическое выделение памяти при выполнении программы.

Вложенные шаблонные классы

Объявление вложенного шаблонного класса может быть выполнено следующим образом:

```
template<class ta> class A
{
   public:
      template<class tai> class AI
      {
         public:
```

```
            AI(ta ta1,tai tai1):ta_(ta1),tai_(tai1)
            {
            }
         private:
            ta ta_;
            tai tai_;
      };
};
int main()
{
   A<int> a;
   A<int>::AI<int> ai(10,20);
   return 0;
}
```

В функции *main()* показана форма вызова конструктора вложенного шаблонного класса.

Пример: вызов функций из вложенных шаблонных классов внутри родительского класса:

```
template<class ta> class A
{
   public:
      template<class tai> class AI
      {
         public:
            void func()
            {
            }
      };
      void myfunc()
      {
         AI<double> a;
         a.func();
      }
};
```

Добавим в конструктор вложенного класса параметры: *template<class ta>*

```
class A
{
   public:
      template<class tai> class AI
      {
         public:
            AI(const ta &v1,const tai &v2)
            {
            }
      };
      void myfunc()
      {
         AI<double> a;
         a.func();
      }
};
int main()
{
   A<int> a;
   // a.myfunc();
   return 0;
}
```

Пример успешно откомпилируется. Но если раскомментировать строку с функцией *a.myfunc()*, то компилятор выдаст ошибку. В данном случае имеет место следующая особенность шаблонов: компилятор не проверяет ни параметры, ни

вызываемые функции до тех пор, пока они не используются.

Данные особенности поведения шаблонов позволяют написать такой фрагмент программы:

```
template<class ta> class A
{
   public:
      template<class tai> class AI
      {
         public:
            AI(const ta &v1,const tai &v2)
            {
               v1.showValue();
               v1.checkValue(10);
            }
      };
   void myfunc()
   {
      AI<double> a(100,20.3);
   }
};
int main()
{
   A<int> a;
   //a.myfunc();
   return 0;
}
```

В теле конструктора вложенного шаблонного класса *AI(const ta &v1,const tai &v2)* вызываются две функции *showValue()* и *checkValue(10)* от типа-шаблона, причем в данном случае компилятор не проверяет, есть ли у данного типа такие функции-члены. Раскомментировав в функции *main()* строку *a.myfunc(),* мы получим ошибку компиляции. А теперь добавим к этому примеру определение структуры *SBase*:

```
struct SBase
{
   int val;
   SBase(int v=0) : val(v)
   {
   }
   void showValue()const
   {
      printf("value: %d\n",val);
   }
   void checkValue(int)const
   {
   }
};
template<class ta=SBase> class A
{
   public:
      template<class tai> class AI
      {
         public:
            AI(const ta &v1,const tai &v2)
            {
               v1.showValue();
               v1.checkValue(10);
            }
      };
      void myfunc()
      {
```

```
            AI<double> a(100,20.3);
        }
};
int main()
{
   A<SBase> a;
   a.myfunc();
   return 0;
}
```

В строке *AI<double> a(100,20.3)* в качестве первого параметра передано число, т.е. предполагается, что шаблонный тип имеет конструктор с типом *int*. В нашем случае именно для этого определен конструктор структуры *SBase.*

```
SBase(int v=0):val(v){}.
```

Теперь рассмотрим такой пример:

```
template<class T>
class A
{
   private:
   template<class T1> class Iterator
   {
      private:
      void calc()
      {
      }
   };
   public:
      void func()
      {
         Iterator<int> it; it.calc();
      }
};
int main()
{
   A<double> a;
   a.func();
   return 0;
}
```

В этом примере была предпринята попытка обратится к private-функции вложенного класса - компилятор, естественно, нам этого не позволил. Чтобы исправить ситуацию нужно объявить родительский класс дружественным для вложенного. Делается это так:

```
template<class T> class A
{
   private:
      template<class T1> class Iterator
      {
         friend class A<T>;
         private:
            void calc()
            {
            }
      };
   public:
      void func()
      {
         Iterator<int> it;
         it.calc();
      }
};
int main()
```

```
{
   A<double> a;
   a.func();
   return 0;
}
```

Теперь все работает.

Наследование шаблонных классов

Механизм наследования шаблонных классов практически ничем не отличается от механизма наследования обычных классов, отличия только в необходимости объявлять шаблонные типы-параметры.

Пример: наследование шаблонных классов.

```
template<class T> class A
{
   public:
      A(const T &t_):t(t_)
      {
      }
   private:
      T t;
};
template<class T1> class B : public A<T1>
{
   public:
      B(const T1 &t_) : A<T1>(t_)
      {
      }
};
int main()
{
   B<double> b(10.0);
   return 0;
}
```

Рассмотрим применение на практике полезных особенностей шаблонных классов.

Например, есть несколько сходных классов, необходимо переопределить одну (или несколько) виртуальную функцию. Подразумевается, что данные базовые классы логически связаны, т.е. переопределяемая виртуальная функция выполняет одинаковое для всех классов действие, например, возвращает координату объекта или еще что-либо. Эту проблему можно решить двумя способами.

Способ первый. От каждого класса произвести потомка с перекрытой соответствующей функцией. В этом случае программа будет работать корректно, но в ней будет присутствовать дублирование кода в нескольких классах.

Пример: переопределение методов при помощи шаблонного дочернего класса.

```
class A
{
   public:
      virtual void print()
      {
         printf("A::print()\n");
      }
};
class B
{
   public:
```

```
      virtual void print()
      {
         printf("B::print()\n");
      }
};
template<class T>
class D : public T
{
   public:
      virtual void print()
      {
         printf("template D::print()\n");
      }
};
int main()
{
   D<A> a1;
   a1.print(); // template D::print()
   D<B> b1;
   b1.print(); // template D::print()
   return 0;
}
```

Способ второй. Использование шаблона. Пример:

```
class A
{
   public:
      void calc()
      {
      }
};
template<class T>
class B : public T
{
   public:
      void print()
      {
      }
};
class C : public B<A>
{
   public:
      void anyfunc()
      {
      }
};
int main()
{
   C c;
   c.print();
   c.anyfunc();
   c.calc();
   return 0;
}
```

В качестве вывода можно заметить следующее. Шаблоны и наследование связаны друг с другом следующим образом:

– шаблон класса может быть производным от шаблонного класса;

– шаблон класса может являться производным от нешаблонного класса;

– шаблонный класс может быть производным от шаблона класса;

– нешаблонный класс может быть производным от шаблона класса.

Шаблоны и друзья

Для шаблонов классов могут быть определены отношения дружественности. Дружественность может быть установлена между шаблоном класса и глобальной функцией, функцией членом другого класса (возможно, шаблонного) или целым классом (возможно, шаблонным).

Если внутри шаблона класса X, который объявлен как

```
template <class T>
class X
```

находится объявление дружественной функции

```
friend void f1();
```

то функция *f1* является дружественной для каждого шаблонного класса, полученного из данного шаблона.

Внутри шаблона классов можно объявить функцию-член другого класса дружественной для любого шаблонного класса, полученного из данного шаблона. Для этого нужно использовать имя функции-члена другого класса, имя этого класса и бинарную операцию разрешения области действия. Например, если внутри шаблона класса *X*, который был объявлен как

```
template <class T>
class X
```

объявляется дружественная функция в форме

```
friend void A::f2();
```

то функция-член *f2* класса *A* будет дружественной для каждого шаблонного класса, полученного из данного шаблона.

Внутри шаблона класса *X*, объявленного как

```
template <class T>
class X
```

можно объявить дружественный класс *Y*

```
friend class Y;
```

В результате чего каждая из функций-членов класса *Y* будет дружественной для каждого шаблонного класса, произведенного из шаблона класса *X*.

Если внутри шаблона класса *X*, который был объявлен как

```
template <class T>
class X
```

объявлен второй класс *Z* в виде

```
friend class Z<T>;
```

то при создании шаблонного класса с конкретным типом *T*, например, типом *double*, все члены класса *Z<double>* будут друзьями шаблонного класса *X<double>*.

Шаблоны и статические члены

В нешаблонных классах статическое свойство или метод принадлежит классу, а не объекту, и, как следствие, всегда существует в единственном экземпляре.

Каждый шаблонный класс, полученный из шаблона классов, имеет собственную копию каждого статического члена данных шаблона; все экземпляры этого шаблонного класса используют свой статический член данных. Статические члены данных шаблонных классов должны быть инициализированы в области

действия файла. Каждый шаблонный класс получает собственную копию статической функции-члена шаблонного класса.

```
template<class T>
class A
{
   public:
      static int a;
};
template<class T> int A<T>::a=10;
int main()
{
   A<double> b1;
   b1.a=20;
   A<double> b2;
   b2.a=50;
   cout<<b1.a<<endl; //50
   A<char> b3;
   cout<<b3.a<<endl;
   A<char> b4;
   b4.a = b3.a * 2;
   cout<<b3.a<<endl;
   return 0;
}
```

КОНСПЕКТ ЛЕКЦИЙ
по дисциплине «Объектно-ориентированное программирование» (часть 1)

Составитель:

Павлий Виталий Александрович – кандидат технических наук, доцент кафедры компьютерного моделирования и дизайна ГОУВПО «ДОННТУ»

Ответственный за выпуск:

Карабчевский Виталий Владиславович – кандидат технических наук, доцент, заведующий кафедрой компьютерного моделирования и дизайна ГОУВПО «ДОННТУ»