Объектно-ориентированное программирование на C++

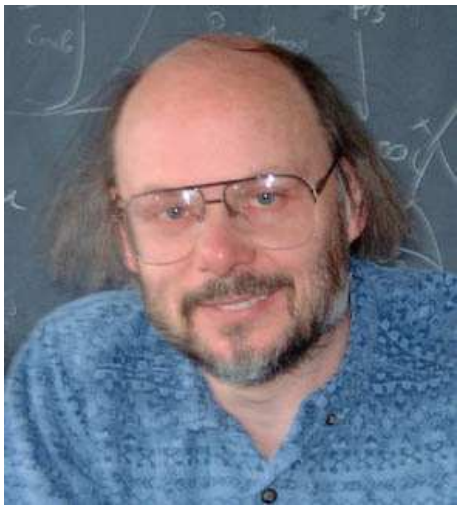
С.Ю. Скоробогатов

Весна 2019

Список литературы по модулю

1. Слайды лекций.
`www.dropbox.com/s/zup2onqbhkjgx9d/module_cpp.pdf`
2. Б. Страуструп. Язык программирования C++. Специальное издание. – М.: Бином, 2011.
`www.dropbox.com/s/6ii9ge0msux8hov/cppbook.djvu`
3. Б. Страуструп. Дизайн и эволюция C++. – М.: ДМК Пресс; Спб.: Питер, 2006.
`www.dropbox.com/s/o8empvmw92nsrq0/cppdesign.djvu`

История языка C++



Б. Страуструп

Основные этапы:

Начало 80-х: C with classes, транслятор cfront.

1983: переименование в C++, виртуальные функции, перегрузка функций и операторов, ссылки.

1985: выход книги Страуструпа The C++ Programming Language.

1989: C++ 2.0 (множественное наследование, абстрактные классы, обобщённые типы, статические функции-члены, функции-константы и защищённые члены).

1998: стандарт ISO/IEC 14882:1998 – C++98.

2003: стандарт ISO/IEC 14882:2003.

2011: стандарт ISO/IEC 14882:2011 – C++11 (выведение типов, расширенный `for`, замыкания).

Основные особенности C++

Совместимость с C: C++ содержит в себе язык C как подмножество (с незначительными изменениями, связанными с более строгим контролем типов);

Поддержка ООП: в C++ реализованы классы с поддержкой множественного наследования, перегрузки операций и обобщённого программирования;

Небезопасность: в C++ ошибки при работе с памятью могут привести к непредсказуемой работе программы;

Ручное управление памятью: автоматическое управление памятью в C++ не реализовано из принципиальных соображений;

Объекты не только в куче: в отличие от языка Java, в C++ объекты не обязаны размещаться на верхнем уровне кучи – они могут «жить» в локальных и глобальных переменных, в элементах массивов, в полях других объектов.

Программа «Hello, World»

Программа на C++ (hello.cpp):

```
1  #include <iostream>
2
3  using namespace std;
4
5  int main()
6  {
7      cout << "Hello, world!" << endl;
8      return 0;
9  }
```

Компиляция:

```
g++ -o hello hello.cpp
```

Запуск:

```
./hello
```

Простые расширения языка C

Однострочные комментарии. Начинаются с `//`.

Операторы-объявления. Объявление локальной переменной может располагаться в любом месте, где допустимо располагать оператор.

Объявления в заголовке for. В секции инициализации оператора `for` можно объявлять переменные.

Перегрузка функций. Допустимо объявлять несколько одноимённых функций с разными сигнатурами.

Теги как имена типов. Теги структур, объединений и перечислений – полноценные имена типов, которые можно использовать без ключевых слов `struct`, `union` и `enum`.

Булевский тип. Базовый тип `bool` с двумя значениями: `true` и `false`. Впрочем, для совместимости с C в условиях оператора `if` и циклов могут применяться числовые значения.

Ссылочный тип данных. *см. следующие слайды.*

Объявление переменных ссылочного типа

Ссылка в C++ – это типизированный указатель, к которому неприменимы арифметические операции, и который не может быть нулевым. Кроме того, недопустимы ссылки на ссылки.
Для объявления ссылки используется префиксный декларатор «&»:

```
тип &имя_переменной;
```

При этом ссылки в глобальных и локальных переменных должны быть обязательно инициализированы при объявлении (забегая вперёд: ссылки в полях объектов должны быть инициализированы в конструкторе класса).
Более того, значение, полученное ссылкой при инициализации, в дальнейшем не может быть изменено.

Инициализация и использование ссылок

Для того чтобы присвоить ссылке адрес некоторого значения в памяти, не нужно использовать операцию «&» для получения адреса объекта. Для доступа к значению, на которое указывает ссылка, не нужно использовать операцию разыменования «*».
Например,

```
int main()
{
    int x = 10;
    int &y = x;   // y указывает на x
    cout << y << "␣";
    y = 20;       // меняем значение x через y
    cout << x << endl;
    return 0;
}
```

Вывод:

```
10 20
```

Ссылки в формальных параметрах функций

Использование ссылок в качестве параметров функции позволяет имитировать var-параметры процедур и функций языка Pascal.
Например,

```
void swap(int &a, int &b) {
    int t = a;
    a = b;
    b = t;
}

int main() {
    int x = 10, y = 20;
    swap(x, y);
    cout << x << "␣" << y << endl;
    return 0;
}
```

Вывод:

```
20 10
```

Ссылки как возвращаемые значения функций

Ссылка может быть возвращаемым значением функции. При этом надо следить, чтобы случайно не вернуть ссылку на локальную переменную.
Например,

```
char &ith(char *s, int i)
{
    return s[i];
}

int main()
{
    char s[] = "qwerty";
    ith(s, 2) = 'x';
    cout << s << endl;
    return 0;
}
```

Вывод:

```
qwxrty
```

Константные ссылки

Как мы увидим в дальнейшем, особую роль играют константные ссылки, объявляемые с модификатором `const`:

```
const тип &имя_переменной;
```

Например,

```
int main()
{
    int x = 10;
    const int &y = x;
    cout << y << endl;
    y = 20; // Ошибка!
    return 0;
}
```

Сообщение об ошибке:

```
test.cpp: In function 'int main()':
test.cpp:11:6: error: assignment of
read-only reference 'y'
```

Объявление класса

В языке C++ любая структура de facto является классом. Однако, классы принято объявлять с помощью конструкции

```
class Имя
{
    ...
};
```

Отличие этой конструкции от объявления структуры заключается в том, что по умолчанию все члены структуры (поля, методы и т.д.) доступны извне структуры, а члены класса – недоступны.
В теле класса располагаются объявления полей, прототипы методов и конструкторов, а также в некоторых случаях определения методов и конструкторов (злоупотреблять определениями не стоит, чтобы не мешать раздельной компиляции).

Секции в объявлении класса

Для управления доступом к членам класса в C++ предусмотрены `public`-, `private`- и `protected`-секции внутри объявления класса:

```
class Имя
{
public:
    ...
private:
    ...
protected:
    ...
};
```

Секции могут быть перечислены внутри объявления класса в любом порядке и количестве. Члены класса, попадающие в `public`-секцию, видны отовсюду, члены из `private`-секции видны только из методов данного класса, а члены `protected`-секции – из методов данного класса и его классов-наследников.

Пример: секции в объявлении класса


```
class Person
{
public:
    string name;
    int year_of_birth;

    Person(string, int, string);
    string get_address();

private:
    string address;
};
```

В примере конструктор класса `Person`, а также метод `get_address` представлены прототипами. Их определения могут находиться в другом файле.

Объявление полей

В языке C++ экземплярные поля класса объявляются также, как и поля структур в языке C. При этом объявления статических полей начинаются с модификатора `static`.
По умолчанию поля недоступны извне класса, но доступны извне структуры или объединения.
Например,

```
class Point
{
public:
    int x, y;           // Координаты точки
    static int count;   // Общее количество точек
};
```

Отметим, что каждое статическое поле должно быть дополнительно *определено* а одном из единиц компиляции проекта (cpp-файле).

Определения статических полей

Особенностью языка C++ является то, что он расчитан на ту же схему компиляции, что и язык C. То есть программа на C++ состоит из набора cpp-файлов, каждый из которых компилируется в объектный файл. Чтобы связать эти объектные файлы, применяется линкер.
Так как класс в общем случае может использоваться в нескольких cpp-файлах, то в каждом файле должно быть его объявление. На практике объявление класса выносится в отдельный h-файл, который включается в нужные cpp-файлы.
Так как статическое поле – это фактически глобальная переменная, то она должна быть явно помещена в один из объектных файлов. Для этого в соответствующем cpp-файле должно быть дано её определение (имя статического поля выглядит как Имя_класса::Имя_поля).

```
int Point::count = 0;
```

Объявление методов

Объявление метода на C++ состоит из прототипа и определения. Прототип метода помещается в тело класса, а определение – в нужный cpp-файл (тут та же история, что и со статическими полями). Прототипы статических методов объявляются с модификатором `static`.
Пример (объявление класса с прототипом метода `dist`):

```
class Point
{
public:
    double x, y;
    double dist();
};
```

Пример (определение метода `dist`):

```
double Point::dist()
{
    return sqrt(x*x + y*y);
}
```

Виртуальные и абстрактные методы

По умолчанию для вызова методов в C++ используется раннее связывание.
Чтобы включить позднее связывание, в объявлении прототипа метода надо указать модификатор `virtual`:

```
class Point
{
public:
    virtual double dist();
};
```

Прототип абстрактного метода заканчивается на «`= 0`»:

```
class Point
{
public:
    virtual double dist() = 0;
};
```

Класс, в котором есть абстрактный метод, автоматически становится абстрактным.

Объявление конструкторов

В C++ объявление экземплярного конструктора, как и объявление обычного экземплярного метода, состоит из прототипа конструктора, записываемого в теле класса, и определения конструктора, размещённого в одном из cpp-файлов. В отличие от объявления метода, в объявлении конструктора имя конструктора совпадает с именем класса, и для конструктора не указывается тип возвращаемого значения. Чтобы запретить создание экземпляра класса, достаточно объявить конструктор в `private`- или `protected`-секции класса. При этом, естественно, в `public`-секции ни одного конструктора не должно быть объявлено.

Специфика реализации конструкторов в языке C++ диктует следующий крайне нетипичный для современных языков принцип: никогда не вызывайте из конструктора виртуальные методы – это может привести к проблемам в случае переопределения этих методов в классах-наследниках.

Пример: объявление конструкторов


```
1  class Point
2  {
3  private:
4      double x, y;
5  public:
6      Point(double x, double y);
7  };
8
9  Point::Point(double x, double y)
10 {
11     this->x = x;  this->y = y;
12 }
```

Ключевое слово `this` в теле метода (в том числе и конструктора) обозначает указатель на объект, для которого вызван конструктор. Мы вынуждены использовать `this` для доступа к полям объекта, так как имена параметров конструктора совпадают с именами полей.

Создание объектов в динамической памяти

Рассмотрим, как в C++ принято создавать объекты классов. При этом мы на данном этапе изучения C++ ограничимся созданием объектов в динамической памяти.
Вообще, динамическое выделение памяти под значение любого типа выполняется в C++ операцией **new**, имеющей следующий синтаксис:

```
new имя_примитивного_типа
```

или

```
new имя_примитивного_типа(начальное_значение)
```

или

```
new имя_класса(фактические_параметры_конструктора)
```

Например,

```
int *x = new int;
float *y = new float(3.14);
Point *p = new Point(1.5, 2.3);
```

Создание массивов в динамической памяти

Операция `new` совмещает выделение памяти с инициализацией (с вызовом конструктора) и возвращает указатель на созданный и инициализированный объект.
В C++ вообще не принято использовать функцию `malloc`, потому что для создания массивов также используется специальная форма операции `new`:

```
new тип_элемента [размер]
```

Например,

```
int *a = new int [10];
Point **pa = new Point* [20];
```

Обратите внимание на то, что массив объектов можно выделять только в случае, если в классе определён конструктор по умолчанию. При этом этот конструктор будет вызван для каждого объекта в массиве.

```
Point *b = new Point [20]; // ошибка
```

Удаление объектов и массивов

Так как в C++ не предусмотрено автоматическое управление памятью (нет сборщика мусора), то все объекты, созданные операцией `new`, должны быть в какой-то момент явно освобождены.
Для этого существует оператор `delete`, имеющий следующий синтаксис:

```
delete указатель_на_объект;     // для объектов
delete [] указатель_на_массив; // для массивов
```

Например,

```
Point *p = new Point(10.0, 20.0);
delete p;
int *a = new int [20];
delete [] a;
```

Объявление деструктора

Внутреннее состояние объекта может содержать указатели (ссылки) на другие объекты, созданные в его конструкторе. В языке без сборки мусора освобождение такого объекта может привести к утечке памяти.

Деструктор – это экземплярный метод, предназначенный для освобождения ресурсов, принадлежащих объекту, непосредственно перед уничтожением этого объекта.

В языке C++ деструкторы жизненно необходимы. Прототип деструктора имеет вид

```
~имя_класса();
```

Деструктор вызывается при уничтожении объекта. В частности, его автоматически вызывает оператор `delete`.

Пример: объявление деструктора


```
1  class IntArray {
2  private:
3      int *a;
4      std::size_t n;
5  public:
6      IntArray(std::size_t n);
7      int &at(int i);
8      virtual ~IntArray();
9  };
10
11 IntArray::IntArray(std::size_t n): n(n) {
12     a = new int [n];
13 }
14
15 int &IntArray::at(int i) {
16     return a[i];
17 }
18
19 IntArray::~IntArray() {
20     delete [] a;
21 }
```

Объекты в автоматической памяти

В отличие от языка Java, в C++ объекты могут размещаться в глобальных и локальных переменных, в параметрах методов, в элементах массивов и полях других объектов. Синтаксис объявления переменной, содержащей объект, выглядит как

```
имя_класса имя_переменной(параметры_конструктора);
```

Такие объявления, кстати, можно применять и для переменных, тип которых не является классом.
Примеры:

```
IntArray a(100);
Point p(10.5, 15.0);
int i(666);
const char *s("qwerty");
```

Объявление переменной, содержащей объект, вызывает экземплярный конструктор класса. Более того, при выходе из блока, где такая переменная объявлена, автоматически вызывается деструктор.

Пример: автоматический вызов деструктора


```
1  #include <iostream>
2
3  using namespace std;
4
5  class Demo {
6  private:
7      int x;
8  public:
9      Demo(int x);
10     virtual ~Demo();
11 };
12
13 Demo::Demo(int x)
14 {
15     this->x = x;
16     cout << "cons:" << x << "␣";
17 }
18
19 Demo::~Demo() { cout << "destr:" << x << "␣";  }
```

Пример: автоматический вызов деструктора (продолжение)


```
21  int main()
22  {
23      Demo d(100);
24      for (int i = 0; i < 2; i++) {
25          Demo d2(i);
26      }
27      return 0;
28  }
```

Вывод:

```
cons:100 cons:0 destr:0 cons:1 destr:1 destr:100
```

Объекты в глобальной памяти

Если объект размещается в глобальной переменной или в статическом поле класса, то конструктор для него вызывается до передачи управления в функцию main, а деструктор – после завершения функции main.

Пример:

```
Demo d(100);

int main()
{
    cout << "main␣";
    return 0;
}
```

Вывод:

```
cons:100 main destr:100
```

Объекты в полях других объектов

Если объект класса `X` содержится в поле `x` объекта `Y`, то конструктор для этого поля вызывается из конструктора `Y` посредством следующей синтаксической конструкции:

```
Y::Y(формальные_параметры_конструктора_Y)
    : x(фактические_параметры_конструктора_X) {
    ...
}
```

Между прочим, любое поле объекта может быть инициализировано таким образом, даже если тип этого поля не является классом. Например, конструктор класса Demo может быть переписан как

```
Demo::Demo(int x): x(x) {
    cout << "cons:" << x << "␣";
}
```

Пример: объекты в полях других объектов


```
class Cont {
private:
    Demo d;
public:
    Cont(int x);
};

Cont::Cont(int x): d(x) {}

int main()
{
    Cont c(100);
    return 0;
}
```

Вывод:

```
cons:100 destr:100
```

Обратите внимание: деструктор класса `Demo` был автоматически вызван при уничтожении объекта класса `Cont`.

Проблема копирования объектов

Следует понимать, что передача объекта в качестве параметра при вызове метода автоматически влечёт создание копии объекта. Смысл копирования заключается в том, что изменение копии объекта внутри метода не приводит к изменению объекта-оригинала (семантика копирования).
Кроме этого, копирование объектов осуществляется при инициализации объявляемых переменных.
По умолчанию создаётся побитовая копия внутреннего состояния объекта. Однако, во многих случаях для обеспечения семантики копирования побитового копирования недостаточно.
Например, для рассмотренного ранее класса `IntArray` побитовое копирование объекта приводит к тому, что возникает два объекта, которые содержат в поле `a` указатели на один и тот же массив целых чисел. Поэтому после уничтожения одного из них второй объект «потеряет» массив.

Объявление конструктора копий

Для создания «правильных» копий объектов необходимо объявить конструктор копий, прототип которого выглядит как

```
имя_класса(const имя_класса &obj);
```

Этот конструктор будет автоматически вызываться всякий раз, когда необходимо создать копию объекта.
Пример:

```
class IntArray {
    ...
    IntArray(const IntArray &obj);
    ...
};

IntArray::IntArray(const IntArray &obj): n(obj.n) {
    a = new int [n];
    std::copy(obj.a, obj.a + n, a);
}
```

Перегруженная операция присваивания

Проблема копирования объектов возникает также при присваивании. Поэтому, забегая вперёд, покажем на примере класса `IntArray`, как определить свою операцию присваивания:

```
class IntArray {
    ...
    IntArray& operator= (const IntArray &obj);
    ...
};

IntArray& IntArray::operator= (const IntArray &obj) {
    if (this != &obj) {
        int *new_a = new int [obj.n];
        std::copy(obj.a, obj.a + obj.n, new_a);
        delete [] a;
        n = obj.n;
        a = new_a;
    }
    return *this;
}
```

Основной способ наследования в C++

Синтаксис одиночного наследование в C++ выглядит как

```
class ИмяПроизводногоКласса: public ИмяБазовогоКласса
{
    ...
};
```

Если конструктор базового класса не имеет параметров (является конструктором по умолчанию), его вызов добавляется компилятором C++ в конструктор производного класса автоматически.
В противном случае необходимо явно вызывать конструктор базового класса в конструкторе производного класса.
В языке C++ вызов конструктора базового класса `X` из конструктора производного класса `Y` выполняется аналогично вызовам конструкторов полей:

```
Y::Y(формальные_параметры_конструктора_Y)
    : X(фактические_параметры_конструктора_X)
{ ... }
```

Пример: одиночное наследование


```
1  class Animal
2  {
3  private:
4      string species;
5  public:
6      Animal(const string &species);
7  };
8
9  Animal::Animal(const string &species):
10     species(species) {}
11
12 class Dog: public Animal
13 {
14 private:
15     string breed;
16 public:
17     Dog(const string &bread);
18 };
19
20 Dog::Dog(const string &bread):
21     Animal("Canis␣lupus␣familiaris"), breed(breed) {}
```

Переопределение методов

В C++, как и в языке Java, для переопределения метода, унаследованного от базового класса, достаточно объявить и определить этот метод в производном классе.
В качестве примера рассмотрим базовый класс `Animal`:

```
1  class Animal
2  {
3  private:
4      string species;
5  public:
6      Animal(const string &species);
7      virtual void draw();
8  };
9
10 Animal::Animal(const string &s): species(s) { }
11
12 void Animal::draw() {
13     cout << "     @ @    \n"
14             "    ('o')   \n"
15             "   o(___)o  \n";
16 }
```

Пример: переопределение методов

Переопределим метод `draw` в производном классе `Dog`:

```
18  class Dog: public Animal
19  {
20  private:
21      string breed;
22  public:
23      Dog(const string &bread);
24      virtual void draw();
25  };
26
27  Dog::Dog(const string &bread):
28      Animal("Canis lupus familiaris"), breed(breed) {}
29
30  void Dog::draw() {
31      cout << "       / |__       \n"
32              "      (    @'___   \n"
33              "    /           O \n"
34              "   /    (_____/   \n"
35              "  /_____/    U    \n";
36  }
```

Динамическое приведение типов

Операция динамического приведения объекта `obj` к типу `T` проверяет, является ли тип `T` одним из типов объекта `obj`, и возвращает `obj`, если является. В противном случае операция либо возвращает нулевой указатель, либо порождает исключение.
В C++ операция динамического приведения типа может применяться к указателям и ссылкам на объекты и записывается как

```
dynamic_cast<T> (obj)
```

Так как нулевых ссылок не существует, то при неудачном приведении порождается исключение `std::bad_cast`.

Пример: динамическое приведение типов не всегда допустимо


```
1  class Animal{};
2  class Dog: public Animal{};
3  class Cat: public Animal{};
4
5  int main()
6  {
7      Animal *a = new Dog();
8      Dog *dog = dynamic_cast<Dog*> (a);
9      cout << dog;
10     return 0;
11 }
```

g++ выдаёт ошибку:

```
cannot dynamic_cast 'a' (of type 'class Animal*') to
type 'class Dog*' (source type is not polymorphic)
```

Т.к. в классе Dog нет виртуальных методов, то для экономии памяти объект этого класса не содержит информации о типе, тем самым делая невозможным динамическое приведение типов.

Пример: динамическое приведение типов


```
1  class Animal
2  {
3  public:
4      virtual void f() {}
5  };
6  class Dog: public Animal{};
7  class Cat: public Animal{};
8
9  int main()
10 {
11     Animal *a = new Dog(), *b = new Cat();
12     Dog *x = dynamic_cast<Dog*> (a),
13         *y = dynamic_cast<Dog*> (b);
14     cout << x << ",␣" << y;
15     return 0;
16 }
```

Вывод:

```
0x603010, 0
```

private- и protected-наследование

Наследование от нескольких базовых классов

C++ поддерживает множественное наследование. Если класс `Y` – производный от нескольких базовых классов `X1`, `X2`, ..., `Xn`, то объявление класса `Y` выглядит на C++ как

```
class Y: мод X1, мод X2, ..., мод Xn
{
    ...
};
```

Здесь «мод» – это `public`, `protected` или `private`.
При этом в конструкторе класса `Y` должны выполняться вызовы конструкторов всех базовых классов, кроме, возможно, тех из них, которые имеют конструкторы по умолчанию:

```
Y::Y(...):
    X1(...),
    X2(...),
    ...,
    Xn(...) {
    ...
}
```

Противоречия в именах наследуемых членов класса

При множественном наследовании в производном классе могут возникать два или более методов с совпадающими именами и сигнатурами. То же самое справедливо и для полей. Пример:

```
1  class A { public: virtual void m() {  } };
2  class B { public: virtual void m() {  } };
3  class C: public A, public B {  };
4
5  int main()
6  {
7      C c;
8      c.m();
9      return 0;
10 }
```

g++ выдаёт ошибку в строке 8:

```
error: request for member 'm' is ambiguous
```

Разрешение противоречий в именах

Для разрешения противоречий в именах наследуемых членов класса необходимо использовать квалифицированные имена.

Пример:

```
1  class A { public: virtual void m() {  } };
2  class B { public: virtual void m() {  } };
3  class C: public A, public B {  };
4
5  int main()
6  {
7      C c;
8      c.A::m();
9      return 0;
10 }
```

Иерархия наследования и классы противоречия

Иерархия наследования – это ориентированный ациклический граф, множеством узлов которого является множество классов программы. При этом если класс `Y` является непосредственным базовым классом для класса `X`, то из узла `X` исходит дуга, входящая в `Y`.

Мы будем говорить, что класс `B` является *классом противоречия*, если существует такой класс `A`, что в иерархии наследования можно провести не менее двух непересекающихся путей из `A` в `B`.

Иерархия наследования называется *противоречивой*, если в ней существуют классы противоречия.

Основная проблема противоречивых иерархий

Основной проблемой противоречивых иерархий является возможность многократного включения полей базового класса в производный класс.

Пример:

```
1  class A
2  {
3  private:
4      int x;
5  };
6  class B: public A {};
7  class C: public A {};
8  class D: public B, public C {};
```

Поле `x` будет содержаться в объекте класса `D` дважды.

Виртуальное наследование

Избежать многократного включения базового класса в производный класс позволяет виртуальное наследование.
Виртуальное наследование – это способ реализации наследования, гарантирующий, что базовый класс не будет включён ни в один из производных классов более чем в одном экземпляре.
Если класс `B` виртуально наследует классу `A`, то говорят, что `A` – *виртуальный базовый класс* для класса `B`. Синтаксически это выражается как

```
class B: мод virtual A
{
    ...
};
```

Здесь «мод» – это `public`, `protected` или `private`.

Пример: виртуальное наследование


```
1  class R
2  {
3  public:
4      int q;
5      R() { q = 13; }
6  };
7
8  class A: virtual R { public: void setq(int x); };
9  class B: virtual R { public: int getq(); };
10 class C: public A, public B {  };
11
12 void A::setq(int x) { q = x; }
13 int B::getq() { return q; }
14
15 int main(void)
16 {
17     C c;
18     c.setq(666);
19     cout << c.getq(); // Выведет 666
20     return 0;
21 }
```

Понятие шаблона

В отличие от языка Java, обобщённые классы в C++ в строгом смысле отсутствуют. Вместо них в C++ используется развитый язык макроопределений, предназначенный для порождения кода во время компиляции программы. Ключевым элментом этого языка является понятие *шаблона*, который является рецептом для генерации кода класса или функции (метода).
Объявление шаблона начинается с ключевого слова `template`, за которым следует список формальных параметров шаблона:

```
template <формальные параметры>
определение класса или функции
```

Тело шаблона представляет собой определение класса или функции, тем самым позволяя объявлять *шаблон класса* и *шаблон функции*.

Типовые формальные параметры шаблона

Формальные параметры шаблона – это идентификаторы, областью видимости которых является тело шаблона. Они могут обозначать типы, значения или, в свою очередь, другие шаблоны. При применении шаблона формальные параметры в его теле заменяются на конкретные типы, значения и шаблоны, и полученный код компилируется.

Типовые параметры шаблона обозначают типы: синтаксически имя типового параметра в списке формальных параметров шаблона предваряется ключевым словом `typename` (или `class`).

Например, шаблон функции:

```
template <typename T>
void do_swap(T& a, T& b)
{
    T c = a;
    a = b;
    b = c;
}
```

Нетиповые формальные параметры шаблона

Нетиповые параметры шаблона представляют значения, а не типы. Их синтаксис похож на объявление переменных. Внутри тела шаблона имя такого параметра обозначает константу указанного типа.
Например, шаблон для матриц:

```
template <typename T, int M, int N>
class Matrix
{
private:
    T a[M][N];
public:
    T& at(int i, int j);
};

template <typename T, int M, int N>
T& Matrix<T,M,N>::at(int i, int j)
{
    assert(i >= 0 && i < M && j >= 0 && j < N);
    return a[i][j];
}
```

Шаблонные формальные параметры шаблона

Шаблонные параметры шаблона позволяют параметризовать шаблон другим шаблоном. Синтаксически такой параметр записывается как объявление шаблона без тела. Например, умножение матриц:

```
template<template <typename, int, int> class M,
    typename T, int P, int Q, int R>
void multiply(M<T,P,R>& c, M<T,P,Q>& a, M<T,Q,R>& b)
{
    M<T,P,R> res;
    for (int i = 0; i < P; i++) {
        for (int j = 0; j < R; j++) {
            res.at(i,j) = 0;
            for (int k = 0; k < Q; k++) {
                res.at(i,j) += a.at(i,k)*b.at(k,j);
            }
        }
    }
    c = res;
}
```

Значения параметров шаблона по умолчанию

Формальные параметры шаблона могут иметь значения по умолчанию. Синтаксически это оформляется путём добавления после имени параметра знака «=», за которым следует значение:

```
template <typename T = double, int M = 3, int N = 3>
class Matrix
{
private:
    T a[M][N];
public:
    T& at(int i, int j);
};
```

Если формальный параметр шаблона имеет значение по умолчанию, то все следующие за ним параметры также должны иметь значения по умолчанию.
Значения по умолчанию используются в случае, если при применении (инстанциации) шаблона соответствующие фактические параметры не указаны.

Зависимые идентификаторы в шаблоне

Зависимый идентификатор – это любое имя внутри определения шаблона, которое зависит от формальных параметров шаблона. Смысл зависимых идентификаторов становится ясен компилятору только при инстанциации шаблона, когда известны его фактические параметры. Поэтому по умолчанию компилятор считает, что зависимые идентификаторы обозначают поля и методы. Чтобы указать, что некоторый зависимый идентификатор обозначает тип, перед ним нужно указывать ключевое слово `typename`.

```
struct Outer
{
    struct Inner { int x; };
};

template <typename T> class Sample
{
    T::Inner x;             // Ошибка!
    typename T::Inner y;    // Правильное объявление
};
```

Инстанциация шаблонов

Инстанциация шаблона – это порождение кода по шаблону и списку фактических параметров. Инстанциация осуществляется при первом использовании конструкции

```
имя_шаблона<список фактических параметров>
```

Например, в момент, когда компилятор обрабатывает объявления переменной

```
Matrix<int, 4, 5> m;
```

он порождает код, получаемый путём подстановки в тело шаблона `Matrix<typename,int,int>` фактических параметров `int`, `4` и `5`.
Отметим, что если последний фактический параметр шаблона – типовый, и к тому же представляет собой инстанциацию другого шаблона, то нужно ставить пробел между заключительнми угловыми скобками. Например,

```
do_swap<Matrix<int,4,5> >(m,m2);
```

Выведение фактических параметров при инстанциации шаблона функции

Если параметры шаблона функции используются в списке формальных параметров этой функции, то компилятор может вывести значения фактических параметров шаблона самостоятельно.
Например, рассмотрим следующей фрагмент кода:

```
Matrix<int,3,1> a;
Matrix<int,1,3> b;
Matrix<int,3,3> c;
a.at(0,0) = 1;
a.at(1,0) = 2;
a.at(2,0) = 3;
b.at(0,0) = 1;
b.at(0,1) = 2;
b.at(0,2) = 3;
multiply<Matrix, int, 3, 1, 3>(c,a,b);
```

Автоматическое выведение фактических параметров шаблона функции позволяет упростить вызов функции `multiply`:

```
multiply(c,a,b);
```

Требования к фактическим параметрам шаблонов

Тело шаблона накладывает требования на его фактические параметры. Например, в теле шаблона функции `multiply` подразумевается, что значения типа `M` – это объекты, имеющие метод `at`, который возвращает ссылку на значение, к которому могут быть применены операции сложения и умножения.

Если фактические параметры шаблона не удовлетворяют требованиям, инстанциация шаблона с этими фактическими параметрами приведёт к ошибке времени компиляции. Например,

```
template <typename T, int M, int N>
struct Array2D { T a[M][N]; };
...
Array2D<int,3,1> x;
Array2D<int,1,3> y;
Array2D<int,3,3> z;
multiply(z,x,y); // error: struct Array2D<int, 3, 3>
                 // has no member named 'at'
```

Специализация шаблона функции

Специализация шаблона – это разработка отдельной версии порождаемого шаблоном кода для конкретного набора фактических параметров.
Пусть имеется шаблон функции вида

```
template <формальные_параметры>
тип имя(...)
{ ... }
```

Специализированная версия функции для конкретного набора фактических параметров записывается как

```
template <>
тип имя<фактические_параметры>(...)
{ ... }
```

Следует иметь в виду, что специализация шаблона функции не даёт особенных преимуществ над обычной перегрузкой функций.

Пример: специализация шаблона функции


```
1  template <typename T> void do_swap(T& a, T& b)
2  {
3      T c = a;
4      a = b;
5      b = c;
6  }
7
8  template <> void do_swap<int>(int& a, int& b)
9  {
10     a += b;
11     b = a-b;
12     a = a-b;
13 }
14
15 int main()
16 {
17     int x = 10, y = 20;
18     do_swap(x, y); // будет вызвана функция
19                    // в строчках 8..13
20     return 0;
21 }
```

Пример: перегрузка имеет больший приоритет, чем специализация


```
1  template <typename T>
2  T t_max(T a, T b)
3  {
4      cout << "1␣";  return a > b ? a : b;
5  }
6
7  template <>
8  int t_max<int>(int a, int b)
9  {
10     cout << "2␣";  return a > b ? a : b;
11 }
12
13 int t_max(int a, int b) {
14     cout << "3␣";  return a > b ? a : b;
15 }
16
17 int main()
18 {
19     int x = 10, y = 20;
20     cout << t_max(x,y); // Вывод: 3 20
21     return 0;
22 }
```

Специализация шаблона класса

Специализированные версии шаблонов классов создаются аналогично специализации шаблонов функций.
Например, рассмотрим шаблон класса, представляющего кортежи фиксированной длины:

```
1  template <typename T, size_t N>
2  class Tuple
3  {
4  private:
5      T a[N];
6  public:
7      T get(int i);
8      void set(int i, T x);
9  };
10
11 template <typename T, size_t N>
12 T Tuple<T,N>::get(int i) { return a[i]; }
13
14 template <typename T, size_t N>
15 void Tuple<T,N>::set(int i, T x) { a[i] = x; }
```

Пример: специализация шаблона класса


```
17  template <>
18  class Tuple<bool, 8>
19  {
20  private:
21      unsigned char a;
22  public:
23      bool get(int i);
24      void set(int i, bool x);
25  };
26
27  bool Tuple<bool,8>::get(int i)
28  {
29      return (a >> i) & 1;
30  }
31
32  void Tuple<bool,8>::set(int i, bool x)
33  {
34      a |= (unsigned char)x << i;
35  }
```

Частичная специализация шаблона класса

При частичной специализации фиксируется только часть фактических параметров шаблона. Например,

```
37  template <typename T>
38  class Tuple<T,1>
39  {
40  private:
41      T a;
42  public:
43      T get(int i);
44      void set(int i, T x);
45  };
46
47  template <typename T>
48  T Tuple<T,1>::get(int i) { return a; }
49
50  template <typename T>
51  void Tuple<T,1>::set(int i, T x) { a = x; }
```

Вычисления во время компиляции

Специализация шаблонов позволяет заставить компилятор выполнить некоторые вычисления во время компиляции. Например, можно попросить компилятор вычислить факториал числа:

```
5  template <int N>
6  struct Fact {
7      static const unsigned long val =
8          N * Fact<N-1>::val;
9  };
10
11 template <>
12 struct Fact<0> {
13     static const unsigned long val = 1;
14 };
15
16 int main()
17 {
18     cout << Fact<3>::val << '\n';
19     return 0;
20 }
```

Пример: возведение числа в степень


```
5  template <typename T, int N>
6  struct Power {
7      static T eval(T x) {
8          if (N % 2 == 0) {
9              return Power<T,N/2>::eval(x*x);
10         }
11         return x * Power<T,N-1>::eval(x);
12     }
13 };
14
15 template <typename T>
16 struct Power<T,0> {
17     static T eval(T x) { return (T)1; }
18 };
19
20 int main()
21 {
22     int x;
23     cin >> x;
24     cout << Power<int,10>::eval(x) << '\n';
25     return 0;
26 }
```

Понятие перегрузки операций

Вообще, de facto практически в любом языке программирования операции перегружены. Например, в языке Pascal смысл операции «+» меняется в зависимости от типов операндов вплоть до того, что «сложение» строк означает их конкатенацию.

Язык C++ позволяет распространить принцип перегрузки на пользовательские типы данных, т.е. на классы. При этом, в отличие от языков, в которых пользователь может сам составлять знаки операций и задавать их приоритет и ассоциативность (например, ML или Haskell), ассортимент операций и их характеристики в C++ жёстко заданы.

Технически перегрузка операций в C++ реализована через определение функций со специальными именами вида «`operator знак_операции`», осуществляющих выполнение операций.

Ассортимент перегружаемых операций

В языке C++ допускается перегрузка следующего набора операций:

```
// унарные:
++ -- - + ! ~ * & (тип)
// бинарные:
= += -= *= /= %= &= |= ^| <<= >>=
+ - * / % & | ^ << >>
== != > < >= <=
&& ||
[] -> ->* () ,
```

Кроме того, допускается перегружать операции «`new`» и «`delete`».

Перегруженные операции имеют те же приоритеты и ассоциативность, что и соответствующие операции над базовыми типами языка C++.

Нельзя перегружать операции «`.`», «`.*`», «`? :`», «`sizeof`», «`dynamic_cast`» и некоторые другие.

Перегрузка операции присваивания

Операция присваивания для некоторого класса `A` перегружается путём объявления в классе метода, прототип которого выглядит как

```
A& operator= (const A &obj);
```

Схема работы перегруженной операции присваивания:

1. проверить, не присваивается ли объект сам себе (если так, то перейти к пункту 5);
2. выделить память, в которую будет скопировано содержимое `obj`;
3. освободить память, используемую внутри объекта;
4. выполнить копирование данных;
5. вернуть `*this`.

Вызов перегруженной операции присваивания можно выполнять двумя способами:

```
a = b;            // 1-ый способ
a.operator=(b); // 2-ой способ
```

Операция присваивания возвращает левое значение

Форма перегрузки операции присваивания в C++ наводит на мысль, что она возвращает *левое значение*, т.е. результату, возвращаемому операцией присваивания, можно чего-то присвоить:

```
A x, y, z;
(x = y) = z;
```

Этот код действительно работает. Более того, в отличие от языка C, язык C++ допускает такое использование операции присваивания и для встроенных типов данных. Например,

```
int a = 2, b = 3, c = 4;
(a = b) = c;
```

Поэтому не стоит пытаться запретить такое поведение путём, например, возвращения константой ссылки:

```
const A& operator= (const A &obj);
```

Составные операции присваивания

Составные операции присваивания перегружаются по той же схеме, что и обычная операция присваивания:

```
A& operator+= (const A &obj);
```

Как правило, в случае сложного внутреннего состояния объекта реализация перегруженной составной операции присваивания также должна предусматривать возможность присваивания объекта самому себе.
Составные операции присваивания также возвращают левое значение, обеспечивая работоспособность конструкций вида

```
(x += y) *= z;
```

Перегрузка бинарных арифметических операций

Прототип перегруженной бинарной арифметической операции может выглядеть как

```
A operator+ (const A &other) const;
```

Отметим, что ключевое слово `const` в конце прототипа метода означает, что метод не меняет внутреннее состояние объекта, для которого он вызван. Если объект – константный, то к нему применимы только константные методы!

Удобно реализовать бинарные арифметические операции через уже реализованные операции составного присваивания:

```
A A::operator+ (const A &other) const {
    return A(*this) += other;
}
```

А вот так делать не надо (лишний вызов конструктора копий + вызов деструктора):

```
A A::operator+ (const A &other) const {
    A result = *this;  result += other;
    return result;
}
```

Перегрузка операций сравнения

Операции сравнения при перегрузке имеют следующие прототипы:

```
bool operator== (const A &other) const;
bool operator!= (const A &other) const;
bool operator<  (const A &other) const;
bool operator>  (const A &other) const;
bool operator<= (const A &other) const;
bool operator>= (const A &other) const;
```

При этом удобно при реализации одних операций сравнения использовать уже реализованные другие операции. Например, реализовав операцию «==», мы можем использовать её отрицание в коде операции «!=»:

```
const bool A::operator!= (const A &other) const {
    return !(*this == other);
}
```

Этот приём, в частности, автоматически обеспечивает непротиворечивость реализации операций сравнения.

Перегрузка операций с помощью функций

Перегруженные операции могут иметь операнды разных типов. Это полезно, например, при реализации операции умножения матрицы на скалярное значение:

```
Matrix operator* (int k) const;
```

Однако, использование такой операции возможно только в случае, когда матрица является первым операндом. Чтобы справиться с этой проблемой, в C++ разрешена перегрузка операций с помощью функций. Например, в нашем случае функция умножения скалярного значения на матрицу может выглядеть как

```
Matrix operator* (int k, const Matrix &m) {
    return m * k;
}
```

Следует иметь в виду, что хотя бы один операнд функции, осуществляющей перегрузку операций, должен быть объектом класса. Это гарантирует невозможность перегрузки операций над встроенными типами.

Перегрузка операций «&&» и «||»

Операции «&&» и «||» могут быть перегружены точно так же, как и другие бинарные арифметические операции:

```
A operator&& (const A &other) const;
A operator|| (const A &other) const;
```

Однако, перегрузка операций «&&» и «||» не рекомендуется. Дело в том, что эти операции применительно к встроенным типам данных вычисляют свой второй операнд не всегда. Эта их особенность широко используется в программировании, т.е. любой программист ожидает, что они будут работать именно по этой схеме.
Перегруженные версии этих операций всегда будут вычислять оба операнда. Тем самым, их использование будет контринтуитивно, что обязательно приведёт к ошибкам в программах.

Перегрузка операции «[ ]»

Операция «[ ]» перегружается следующим образом:

```
тип1& operator[] (тип2 index);
```

Операция возвращает ссылку, что даёт возможность использовать её результат в левой части операции присваивания. К сожалению, таким образом перегруженную операцию невозможно применять к константным объектам. Поэтому принято определять ещё и специальную версию операции «[ ]» для константных объектов:

```
const тип1& operator[] (тип2 index) const;
```

Естественно, в этом случае необязательно, чтобы операция возвращала ссылку. Поэтому, если «тип1» допускает дешёвое копирование, от ссылки можно избавиться. Например,

```
double operator[] (int index) const;
```

Отметим, что операция «[ ]» не может быть перегружена с помощью функции.

Перегрузка унарных операций

Унарные операции перегружаются по одной из приведённых схем (далее «@» – некоторый знак операции):

```
class A
{
    ...
    тип operator@ ();     // с помощью метода
    ...
};
тип operator@ (A &obj);   // с помощью функции
```

Исключение составляют постфиксные ++ и --. Чтобы их отличать от префиксных инкремента и декремента, они имеют дополнительный фиктивный параметр типа int:

```
class A
{
    ...
    A operator++ (int);
    ...
};
A operator++ (A &obj, int);
```

Перегрузка операций «*» и «->»

Перегрузка операций «*» и «->» позволяет использовать объекты класса так, как если бы они были указателями:

```
class A
{
    ...
    тип& operator* ();
    тип& operator-> ();
    ...
};
тип& operator* (A &a);
```

Особенностью операции «->» является тот факт, что она последовательно применяется к своему же возвращаемому значению до тех пор, пока не получится указатель. Т.е., например, если в классе `A` операция «->» возвращает ссылку на объект класса `B`, а в классе `B` операция «->» возвращает указатель на объект класса `C`, то применение «->» к переменной `x` типа `A` открывает доступ к полям и методам объекта класса `C`.

Пример: перегрузка операции «->»


```
1  struct C { int field; };
2
3  struct B {
4      C c;
5      C* operator-> ();
6  };
7
8  C* B::operator-> () { return &c; }
9
10 struct A {
11     B b;
12     B& operator-> ();
13 };
14
15 B& A::operator-> () { return b; }
16
17 int main()
18 {
19     A x;
20     x->field = 666;
21     return 0;
22 }
```

Перегрузка операции «&»

Перегрузка операции «&» позволяет вместо адреса объекта возвращать всё, что угодно (например, «умный» указатель, т.е. объект с перегруженными операциями «*» и «->», осуществляющий подсчёт ссылок на объект и автоматическое его удаление при уничтожении последней ссылки):

```
class A
{
    ...
    тип operator& ();
    ...
};
тип operator& (A &a);
```

Впрочем, перегрузка этой операции не рекомендуется, т.к. может привести к трудноуловимым ошибкам в коде. Вместо её перегрузки рекомендуется определить метод, который делает то же самое, но не экранирует стандартную реализацию «&»:

```
тип address_of ();
```

Перегрузка операции приведения типа

Перегрузка операции приведения объекта к некоторому типу осуществляется путём объявления метода

```
operator тип() const;
```

Например,

```
struct A {
    int x;
    operator int() const;
};

A::operator int() const { return x; }
```

Обратите внимание на то, что, хотя операция приведения типа возвращает значение, в её объявлении тип возвращаемого значения не указывается. В этом она напоминает конструктор.

Перегрузка операции «( )»

Перегрузка операции ( ) даёт возможность «вызывать» объекты класса, как если бы они были функциями. Прототип перегруженной операции ( ) выглядит как

```
тип operator()(список формальных параметров);
```

Можно сказать, что операция ( ) – n-арная, потому что внутри скобок может располагаться произвольное количество формальных параметров.
Естественно, в классе можно определить несколько перегруженных операций ( ), если они будут различаться сигнатурами.

Оператор throw

В отличие от языка Java, в качестве исключения в C++ может выступать любое значение (объект, строка, значение базового типа и т.п.).
Для порождения исключения предназначен оператор `throw`, который может вызываться как с указанием значения, описывающего исключительную ситуацию, так и без его указания:

```
throw значение;
throw; // вариант без указания значения
```

Механизм исключений в C++ не предусматривает запоминание информации о стеке вызовов в момент порождения исключения. Поэтому сообщение, выводимое при аварийном завершении программы по причине необработанного исключения, менее информативно.

Пример: оператор throw


```
1  int main()
2  {
3      throw;
4      return 0;
5  }
```

Вывод:

```
terminate called without an active exception
```

```
1  int f(int x) { return 2*x; }
2  int main()
3  {
4      throw f;
5      return 0;
6  }
```

Вывод:

```
terminate called after throwing an instance
of 'int (*)(int)'
```

Синтаксис try- и catch-блоков

Как и в языке Java, участки кода, в которых ожидается возникновение исключительной ситуации, обрамляются try-блоками. С try-блоками связаны один или несколько catch-блоков, осуществляющих перехват исключений по типу:

```
try {
    /*...*/
}
catch (тип_исключения1 переменная) {
    /*...*/
}
catch (тип_исключения2 переменная) {
    /*...*/
}
```

Catch-блок, который может перехватить любое исключение, записывается как

```
catch (...) {
    /*...*/
}
```

Пример: перехват исключения std::bad_alloc


```
1  #include <iostream>
2  using namespace std;
3
4  int main()
5  {
6      try {
7          size_t n;
8          cin >> n;
9          int *a = new int[n];
10         cout << "success\n";
11         delete[] a;
12     } catch (bad_alloc) {
13         cout << "out␣of␣memory\n";
14     }
15     return 0;
16 }
```

Исключение `std::bad_alloc` порождается операцией `new` при невозможности выделить блок динамической памяти.

Жизненный цикл объектов-исключений

Исследуем жизненный цикл объектов исключений с помощью класса A, который протоколирует факты создания, копирования и уничтожения своих экземпляров:

```
struct A {
    bool is_copy;
    A();
    A(const A &obj);
    ~A();
    void report();
};

A::A(): is_copy(false)
{ cout << "new" << '\n'; }

A::A(const A &obj): is_copy(true)
{ cout << "copy" << '\n'; }

A::~A() { cout << "destroy " << is_copy << '\n'; }

void A::report() { cout << "report\n"; }
```

Пример 1: жизненный цикл объектов-исключений

Порождение объекта-исключения в куче и перехват, при котором он передаётся в catch-блок по указателю. В конце catch-блока необходимо явное уничтожение объекта.

```
int main()
{
    try {
        throw new A();
    } catch (A *a) {
        a->report();
        delete a;
    }
    return 0;
}
```

Вывод:

```
new
report
destroy 0
```

Пример 2: жизненный цикл объектов-исключений

Порождение временного объекта-исключения и перехват, при котором создаётся его копия.

```
int main()
{
    try {
        throw A();
    } catch (A a) {
        a.report();
    }
    return 0;
}
```

Вывод:

```
new
copy
report
destroy 1
destroy 0
```

Пример 3: жизненный цикл объектов-исключений

Порождение временного объекта-исключения и перехват, при котором он передаётся в catch-блок по ссылке, т.е. без создания копии.

```
int main()
{
    try {
        throw A();
    } catch (A &a) {
        a.report();
    }
    return 0;
}
```

Вывод:

```
new
report
destroy 0
```

Пример: уничтожение автоматических объектов в процессе передачи исключения в catch-блок


```
1  #include <iostream>
2
3  struct Dummy { ~Dummy(); };
4
5  Dummy::~Dummy() { std::cout << "#␣"; }
6
7  void recur(int n) {
8      if (n == 0) throw "base␣of␣recursion";
9      Dummy d;
10     recur(n-1);
11     std::cout << "should␣never␣be␣reached!";
12 }
13
14 int main()
15 {
16     try { recur(10); } catch (...) { }
17     return 0;
18 }
```

Вывод:

```
# # # # # # # # # #
```

Перехват исключений в инициализаторах конструкторов

Напомним, что конструкторы в C++ могут иметь списки инициализаторов, в которых вызываются конструкторы объектов, расположенных в полях, и конструкторы базовых классов.

Так как конструкторы полей и базовых классов могут порождать исключения, нужно уметь эти исключения перехватывать, например, для протоколирования неудачного создания объектов в логах. Для этой цели служат специальные try-блоки, обрамляющие тело конструктора, которые синтаксически выглядят как

```
Класс::Класс(параметры_конструктора) try
    : список_инициализации
{ тело_конструктора }
catch (исключение) { тело_обработчика_исключения }
...
```

Отметим, что если catch-блок не оканчивается порождением другого исключения, старое исключение не уничтожается и передаётся дальше.

Пример: перехват исключений в инициализаторах конструкторов


```
1  #include <iostream>
2  using namespace std;
3
4  struct A { A(int x); };
5  struct B: public A { B(int x); };
6
7  A::A(int x) { if (x < 0) throw "negative"; }
8
9  B::B(int x) try: A(x) { }
10 catch (const char *s) { cout << "1␣"; }
11
12 int main()
13 {
14     try {
15         B b(-10);
16     } catch(...) {
17         cout << "2\n";
18     }
19     return 0;
20 }
```

```
1 2
```

Спецификатор throw в заголовках функций

Как и в языке Java, в C++ можно перечислить типы исключений, которые может порождать вызов функции. Для этого после списка формальных параметров следует разместить спецификатор `throw`:

```
throw (список типов исключений)
```

Семантика этого спецификатора несколько отличается от семантики соответствующей конструкции языка Java, а именно:

- вызов функции, для которой разрешённые исключения не указаны, может порождать любые исключения;
- вызов функции со спецификатором `throw()` не порождает никаких исключений;
- компилятор не следит за тем, какие исключения порождаются внутри функции: вместо этого неразрешённые исключения просто не могут выйти за пределы функции и приводят к аварийному завершению программы.

Пример 1: спецификатор throw


```
1  #include <iostream>
2  using namespace std;
3
4  void f() throw(int)
5  {
6      throw 2;
7  }
8
9  int main()
10 {
11     try {
12         f();
13     } catch (int e) {
14         cout << "int: " << e << '\n';
15     }
16     return 0;
17 }
```

Вывод (исключение 2 «пропущено» наружу из функции `f`):

```
int: 2
```

Пример 2: спецификатор throw


```
1  #include <iostream>
2  using namespace std;
3
4  void f() throw()
5  {
6      throw 2;
7  }
8
9  int main()
10 {
11     try {
12         f();
13     } catch (int e) {
14         cout << "int: " << e << '\n';
15     }
16     return 0;
17 }
```

Вывод (исключение 2 «не выпущено» из функции `f`):

```
terminate called after throwing an instance
of 'int'
```

Спецификатор throw и переопределение виртуальных методов

По очевидным соображениям, связанным с поддержкой полиморфизма, переопределение виртуальных методов не должно ослаблять ограничения, накладываемые спецификатором `throw`, т.е. переопределённому методу не разрешается порождать исключения, не порождаемые методом базового класса.

```
struct A {
    virtual void f() throw(int) {}
};

struct B: public A {
    void f() throw(float) {}
};
```

Сообщение об ошибке:

```
error: looser throw specifier for
'virtual␣void␣B::f()␣throw␣(float)'
overriding 'virtual␣void␣A::f()␣throw␣(int)'
```

Создание классов исключений

Рекомендуемым способом создания классов исключений в языке C++ является наследование их от библиотечного класса `exception` (его объявление расположено в заголовочном файле `<exception>`):

```
class exception {
public:
    exception () throw();
    exception (const exception&) throw();
    exception& operator= (const exception&) throw();
    virtual ~exception() throw();
    virtual const char* what() const throw();
};
```

В производном от `exception` классе нужно переопределить виртуальный метод `what`, возвращающий ASCIIZ-строку с описанием исключительной ситуации.

Пример: создание классов исключений


```
1  #include <exception>
2  using namespace std;
3
4  struct MyException: public exception
5  {
6      const char *what() const throw()
7      { return "This is my exception"; }
8  };
9
10 int main()
11 {
12     throw MyException();
13     return 0;
14 }
```

Вывод:

```
terminate called after throwing an instance
of 'MyException'
  what():  This is my exception
Aborted
```

Чрезвычайно большую роль при программировании на C++ играют так называемые *умные указатели* – объекты классов, «притворяющиеся» указателями.
Подобно обычному указателю, умный указатель реализует следующую функциональность:

- он обеспечивает доступ к некоторому значению в памяти, а именно:
  - имитирует «разыменование» с помощью перегруженной унарной операции *;
  - в случае, если значение является структурой, разрешает обращение к её полям с помощью перегруженной операции ->;
- применение к умному указателю арифметических аддитивных операций позволяет переключаться с одного значения на другое (эта возможность реализована в *итераторах*).

Пример: ленивый указатель

Для демонстрации идеи умного указателя напишем шаблон класса `lazy_ptr` (в синтаксисе C++11), реализующего ленивое выделение памяти для некоторого значения типа `T`.

```
1  template<typename T>
2  class lazy_ptr final {
3  private:
4      T *value;
5      lazy_ptr(const lazy_ptr<T>&);
6      lazy_ptr<T>& operator=(const lazy_ptr<T>&);
7  public:
8      lazy_ptr(): value(nullptr) {}
9      ~lazy_ptr() { delete value; }
10
11     T& operator* () {
12         if (!value) value = new T;
13         return *value;
14     }
15
16     T *operator-> () { return &**this; }
17 };
```

Пример: ленивый указатель (продолжение)

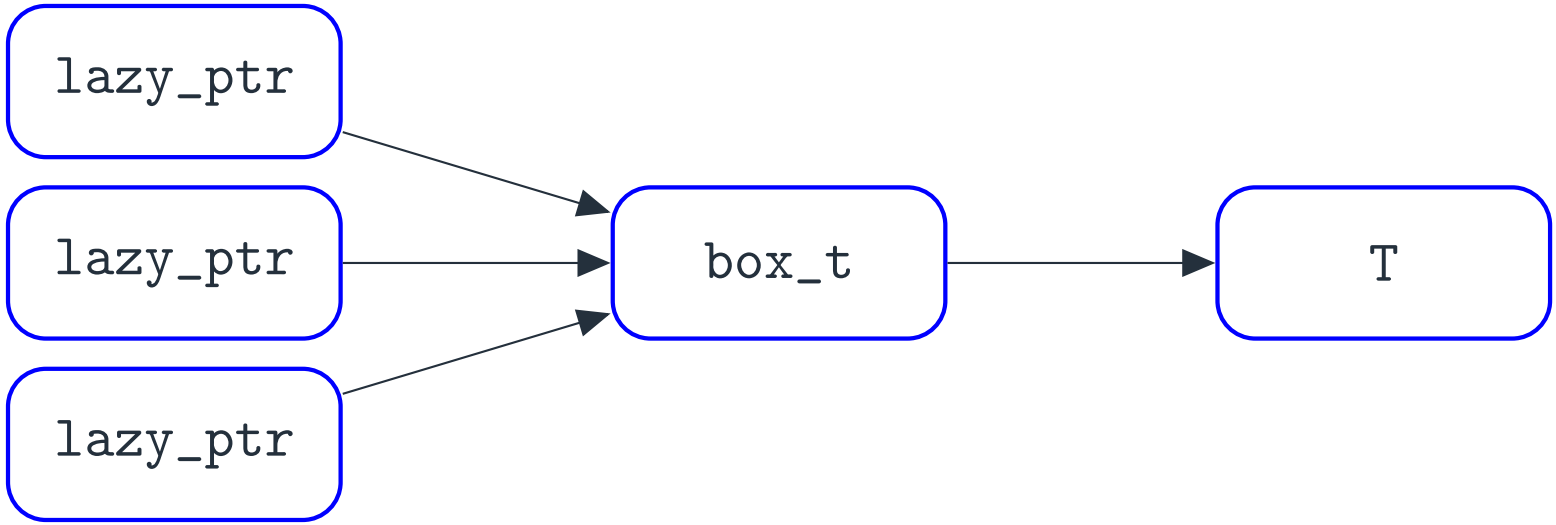
Обращение к динамической переменной через `lazy_ptr` осуществляется также, как если бы мы использовали для этого обычаный указатель, однако память выделяется автоматически при первом обращении (в строчке 26):

```
19  struct Point {
20      int x, y;
21  };
22
23  int main()
24  {
25      lazy_ptr<Point> a;
26      a->x = 10;
27      a->y = 20;
28      std::cout << a->x << ' ' << a->y << std::endl;
29      return 0;
30  }
```

Освобождение памяти также происходит автоматически при уничтожении указателя. (Для этого пришлось запретить копирование и присваивание указателей.)

Пример: ленивый указатель с подсчётом ссылок

Если разрешить копирование и присваивание указателей `lazy_ptr`, то получится, что сразу несколько указателей могут ссылаться на одну динамическую переменную.



Для организации удаления динамической переменной мы можем завести вспомогательную структуру типа `box_t` (т.н. *трамплин*): она будет содержать указатель на переменную и счётчик умных указателей. Переменная будет удаляться, когда счётчик принимает нулевое значение.

Пример: ленивый указатель с подсчётом ссылок (продолжение)

Реализация шаблона box_t:

```
1  template<typename T>
2  class box_t final {
3  private:
4      T *ptr;
5      size_t count;
6
7      box_t(const box_t<T>&);
8      box_t& operator= (const box_t<T>&);
9  public:
10     box_t(): ptr(nullptr), count(1) {}
11     ~box_t() { delete ptr; }
12
13     void inc() { count++; }
14     bool dec() { return --count == 0; }
15
16     T *get() {
17         if (!ptr) ptr = new T;
18         return ptr;
19     }
20 };
```

Пример: ленивый указатель с подсчётом ссылок (продолжение)

Реализация шаблона `lazy_ptr`:

```
22  template<typename T>
23  class lazy_ptr final {
24  private:
25      box_t<T> *box;
26  public:
27      lazy_ptr(): box(new box_t<T>) {}
28      ~lazy_ptr() { if (box->dec()) delete box; }
29
30      lazy_ptr(const lazy_ptr<T>& p): box(p.box) {
31          box->inc();
32      }
33
34      lazy_ptr<T>& operator= (lazy_ptr<T> copy) {
35          std::swap(box, copy.box);
36          return *this;
37      }
38
39      T& operator* () { return *box->get(); }
40      T *operator-> () { return box->get(); }
41  };
```

Пример: ленивый указатель с подсчётом ссылок (продолжение)

Теперь для `lazy_ptr` доступно копирование и присваивание. В частности, в тестовом коде мы передаём копию указателя в функцию `f`, в которой, собственно, и происходит первое обращение к структуре `Point`, вызывающее выделение памяти под структуру.

```
43  struct Point {
44      int x, y;
45  };
46
47  void f(lazy_ptr<Point> p) {
48      p->x = 10;
49      p->y = 20;
50  }
51
52  int main() {
53      lazy_ptr<Point> a;
54      f(a);
55      std::cout << a->x << ' ' << a->y << std::endl;
56      return 0;
57  }
```

Умные указатели в стандартной библиотеке

Начиная с C++11, в заголовочном файле `<memory>` присутствуют два стандартных шаблона умных указателей:

- `unique_ptr` – единолично владеет динамической переменной и освобождает занимаемую ею память при своём уничтожении;
- `shared_ptr` – делит владение динамической переменной с группой других умных указателей типа `shared_ptr` (при этом освобождение памяти, занимаемой этой переменной, происходит только после уничтожения всех указателей группы).

Кроме того, существует шаблон `weak_ptr`, инстанциация которого хранит адрес динамической переменной, которой владеет некоторый `shared_ptr` (память, занимаемая переменной, может быть освобождена независимо от наличия ссылающихся на неё указателей `weak_ptr`).

Замечание. «*Владение*» переменной означает ответственность за освобождение занимаемой ею памяти.

Шаблон unique_ptr

Параметром шаблона `unique_ptr` является тип динамической переменной, которой умный указатель будет владеть. Конструктор `unique_ptr` принимает в качестве параметра адрес динамической переменной. Например,

```
std::unique_ptr<Point> a(new Point);
```

Замечание. Имеет смысл совместить вызов конструктора `unique_ptr` с созданием динамической переменной, чтобы адрес переменной вообще не хранился нигде, кроме как внутри умного указателя.

Доступ к динамической переменной должен осуществляться через операции `*` и `->`, перегруженные у `unique_ptr`.

Замечание. При острой необходимости можно получить адрес динамической переменной, которой владеет `unique_ptr`, вызвав его метод `get`. Но это не очень хорошая идея, т.к. по смыслу единоличного владения переменной только сам `unique_ptr` имеет право знать этот адрес.

Передача владения переменной

Для `unique_ptr` запрещено копирование и присваивание, т.е. следующий фрагмент кода не откомпилируется:

```
std::unique_ptr<Point> a(new Point);
std::unique_ptr<Point> b = a; // ошибка
b = a; // ошибка
```

Замечание. Запрет на копирование и присваивание распространяется и на объекты, имеющие `unique_ptr` в полях.

Тем не менее, можно передать владение переменной от одного `unique_ptr` к другому с помощью функции `move`:

```
std::unique_ptr<Point> a(new Point);
std::unique_ptr<Point> b = std::move(a); // OK
std::unique_ptr<Point> c;
c = std::move(b); // OK
```

Замечание. После применения функции `move` к `unique_ptr` умный указатель лишается владения динамической переменной, т.е. попытка использовать его для обращения к переменной приводит к «Segmentation fault».

Пример: использование unique_ptr


```
1  #include <iostream>
2  #include <memory>
3
4  struct Point {
5      int x, y;
6  };
7
8  int main() {
9      std::unique_ptr<Point> a(new Point);
10     a->x = 10;
11     a->y = 20;
12     std::cout << a->x << ' ' << a->y << std::endl;
13
14     std::unique_ptr<Point> b = std::move(a);
15     std::cout << b->x << ' ' << b->y << std::endl;
16     return 0;
17 }
```

В строчке 14 происходит передача владения структурой `Point` от `a` к `b`. При завершении функции `main` память, занимаемая структурой, автоматически освобождается в деструкторе умного указателя `b`.

Шаблон shared_ptr

В отличие от `unique_ptr`, умные указатели, созданные из шаблона `shared_ptr` можно копировать и присваивать:

```
std::shared_ptr<Point> a(new Point);
std::shared_ptr<Point> b = a; // OK
std::shared_ptr<Point> c;
c = a; // OK
```

При этом образуется группа указателей, владеющая одной и той же динамической переменной.
Кроме адреса переменной, каждый указатель группы содержит адрес вспомогательной структуры, в которой хранится размер группы. Этот размер изменяется при создании и уничтожении указателей, входящих в группу.
Деструктор `shared_ptr` проверяет размер группы и удаляет как динамическую переменную, так и вспомогательную структуру, если выясняется, что уничтожаемый указатель – последний в группе.

Пример: использование shared_ptr


```
1  #include <iostream>
2  #include <memory>
3
4  struct Point {
5      int x, y;
6  };
7
8  void f(std::shared_ptr<Point> p) {
9      p->x = 10;
10     p->y = 20;
11 }
12
13 int main() {
14     std::shared_ptr<Point> a(new Point);
15     f(a);
16
17     std::shared_ptr<Point> b;
18     b = a;
19     std::cout << b->x << ' ' << b->y << std::endl;
20     return 0;
21 }
```

Проблема циклических ссылок

Автоматическое освобождение памяти, реализованное в `shared_ptr`, не работает, если умные указатели образуют циклические ссылки, т.е. каким-либо образом ссылаются друг на друга. Например, в следующей программе будут утечки памяти:

```
1  struct B;
2
3  struct A { std::shared_ptr<B> ptr; };
4
5  struct B { std::shared_ptr<A> ptr; };
6
7  int main() {
8      std::shared_ptr<A> a(new A);
9      std::shared_ptr<B> b(new B);
10     a->ptr = b;
11     b->ptr = a;
12     return 0;
13 }
```

Шаблон weak_ptr

Объект типа `weak_ptr` работает в сочетании с `shared_ptr`: он может хранить ссылку на динамическую переменную, которой владеет `shared_ptr`, но при этом сам данной переменной не владеет.
`weak_ptr` получает ссылку на динамическую переменную из `shared_ptr`:

- `shared_ptr` может быть передан в качестве параметра конструктору `weak_ptr`;

```
std::shared_ptr<Point> a(new Point);
std::weak_ptr<Point> b(a);
```

- `shared_ptr` может быть присвоен `weak_ptr`.

```
std::shared_ptr<Point> a(new Point);
std::weak_ptr<Point> b;
b = a;
```

Память, занимаемая переменной, на которую ссылается `weak_ptr`, будет освобождена с уничтожением последнего `shared_ptr`, владеющего этой переменной.

Пример: решение проблемы циклических ссылок с помощью weak_ptr


```
1  struct B;
2
3  struct A {
4      std::shared_ptr<B> ptr;
5  };
6
7  struct B {
8      std::weak_ptr<A> ptr;
9  };
10
11 int main() {
12     std::shared_ptr<A> a(new A);
13     std::shared_ptr<B> b(new B);
14     a->ptr = b;
15     b->ptr = a;
16     return 0;
17 }
```

Работа с динамической переменной через weak_ptr

Сам по себе `week_ptr` не предназначен для обращения к динамической переменной, адрес которой в нём содержится (в частности, у него не перегружены операции `*` и `->`).
Однако, у `week_ptr` имеется метод `lock`, формирующий и возвращающий `shared_ptr`, владеющий соответствующей переменной.
Например,

```
int main() {
    std::shared_ptr<Point> a(new Point);
    a->x = 10;
    a->y = 20;

    std::weak_ptr<Point> b(a);
    std::shared_ptr<Point> c = b.lock();
    std::cout << c->x << ' ' << c->y << std::endl;
    return 0;
}
```

Категории итераторов в C++

Итератор в C++ – это умный указатель, предназначенный для организации доступа к элементам, содержащимся внутри объекта контейнерного класса.
Итераторы подразделяются на пять категорий в зависимости от реализуемой функциональности:

1. итераторы ввода;
2. итераторы вывода;
3. однонаправленные итераторы – одновременно являются как итераторами ввода, так и итераторами вывода (в том случае, если не являются константными);
4. двунаправленные итераторы – одновременно являются однонаправленными итераторами;
5. итераторы произвольного доступа – одновременно являются двунаправленными итераторами.

Замечание. Любые итераторы можно копировать и присваивать.

Итераторы ввода и итераторы вывода

Итераторы ввода предназначены для чтения элементов контейнера, а итераторы вывода – для записи элементов в контейнер.
Как итераторы ввода, так и итераторы вывода поддерживают операции префиксного и постфиксного инкремента, означающие переход к следующему элементу контейнера:

```
++a
a++
```

Итераторы ввода поддерживают сравнения на равенство/неравенство (два итератора равны, если указывают на один и тот же элемент):

```
a == b
a != b
```

Разыменование итератора ввода даёт значение элемента (не может стоять в левой части операции присваивания). Разыменование итератора вывода даёт левое значение.

Однонаправленные и двунаправленные итераторы

Однонаправленные итераторы обладают всей функциональностью итераторов ввода и итераторов вывода (в случае, если они не являются константными), и, кроме того, реализуют следующие возможности:

- проверка, что достигнут конец контейнера – у итератора есть специальное значение, порождаемое конструктором по умолчанию и обозначающее псевдоэлемент, следующий за последним элементом контейнера;
- повторное использование – ни разыменование, ни инкремент итератора не влияют на возможность его разыменования в дальнейшем:

```
b = a;
std::cout << *a++;
std::cout << *b;
```

Двунаправленные итераторы расширяют функциональность однонаправленных поддержкой декремента, означающего переход к предыдущему элементу контейнера.

Итераторы произвольного доступа

Итераторы произвольного доступа обладают всей функциональностью двунаправленных итераторов и, кроме того, дают возможность непоследовательного доступа к элементам контейнера.
Итераторы произвольного доступа поддерживают прибавление и вычитание целых чисел, означающие пропуск x элементов вперёд или назад:

```
a + x
x + a
a - x
```

Также поддерживаются составные операции присваивания:

```
a += x
a -= x
```

Кроме того, разность двух итераторов возвращает количество элементов между ними:

```
a - b
```

Итераторы произвольного доступа (продолжение)

Итераторы произвольного доступа можно сравнивать на больше/меньше (порядок на множестве итераторов, относящихся к одному и тому же контейнеру, определяется порядком следования элементов контейнера):

```
a < b
a <= b
a > b
a >= b
```

К итераторам произвольного доступа можно применять операцию индексации:

```
a[x]
```

(Запись `a[x]` должна означать то же самое, что и `*(a + x)`.)

Шаблон iterator

Итераторы принято наследовать от класса, задаваемого шаблоном `iterator` из заголовочного файла `<iterator>`:

```
template <class Category, class T,
          class Distance = ptrdiff_t,
          class Pointer = T*, class Reference = T&>
struct iterator {
    typedef T          value_type;
    typedef Distance   difference_type;
    typedef Pointer    pointer;
    typedef Reference  reference;
    typedef Category   iterator_category;
};
```

Здесь `Category` – это один из следующих типов:

```
struct input_iterator_tag {};
struct output_iterator_tag {};
struct forward_iterator_tag {};
struct bidirectional_iterator_tag {}
struct random_access_iterator_tag {};
```