

КАРСОНА COMPUTER SCIENCE

ОБЪЕКТНО-ОРИЕНТИРОВАННОЕ ПРОГРАММИРОВАНИЕ

В C++

4-Е ИЗДАНИЕ



Р. ЛАФОРЕ

SAMS

ПИТЕР®

Глава 11

Виртуальные функции

- ◆ Виртуальные функции
- ◆ Дружественные функции
- ◆ Статические функции
- ◆ Инициализация копирования и присваивания
- ◆ Указатель `this`
- ◆ Динамическая информация о типах

Теперь, когда мы уже знаем кое-что об указателях, перейдем к изучению более сложных вопросов C++. Эта глава посвящена таким довольно слабо связанным между собой темам, как виртуальные функции, дружественные функции, статические функции, перегружаемые операции и методы, а также указатель `this`. Все это — прогрессивные особенности языка, однако далеко не везде их стоит использовать. Небольшие программы могут вполне обойтись и без них. Тем не менее они часто применяются и очень важны для больших, серьезных программ. В частности, виртуальные функции необходимы при использовании полиморфизма — одного из краеугольных камней ООП.

Виртуальные функции

Виртуальный означает *видимый, но не существующий в реальности*. Когда используются виртуальные функции, программа, которая, казалось бы, вызывает функцию одного класса, может в этот момент вызывать функцию совсем другого класса. А для чего вообще нужны виртуальные функции? Представьте, что имеется набор объектов разных классов, но вам хочется, чтобы они все были в одном массиве и вызывались с помощью одного и того же выражения. Например, в графической программе MULTSHAP из главы 9 «Наследование» есть разные геометрические фигуры: треугольник, шар, квадрат и т. д. В каждом из этих классов есть функция `draw()`, которая прорисовывает на экране фигуры.

Теперь, допустим, вам захотелось создать картинку, сгруппировав некоторые из этих элементов. Как бы сделать это без лишних сложностей? Подход к реше-

нию этой задачи таков: создайте массив указателей на все неповторяющиеся элементы картинки:

```
shape* ptarr[100]: //массив из 100 указателей на фигуры
```

Если в этом массиве содержатся указатели на все необходимые геометрические фигуры, то вся картинка может быть нарисована в обычном цикле:

```
for(int j=0; j<N; j++)
    ptarr[j]->draw();
```

Это просто потрясающая возможность! Абсолютно разные функции выполняются с помощью одного и того же вызова! Если указатель в массиве `ptarr` указывает на шарик, вызовется функция, рисующая шарик, если он указывает на треугольник, то рисуется треугольник. Вот это и есть *полиморфизм*, то есть *различные формы*. Функции выглядят одинаково — это выражение `draw()`, но реально вызываются разные функции, в зависимости от значения `ptarr[j]`. Полиморфизм — одна из ключевых особенностей объектно-ориентированного программирования (ООП) наряду с классами и наследованием.

Чтобы использовать полиморфизм, необходимо выполнять некоторые условия. Во-первых, все классы (все эти треугольнички, шарики и т. д.) должны являться наследниками одного и того же базового класса. В MULTSHAP этот класс называется `shape`. Во-вторых, функция `draw()` должна быть объявлена виртуальной (*virtual*) в базовом классе.

Все это выглядит абстрактно, поэтому давайте напишем несколько небольших программ, которые выявят некоторые практические вопросы, чтобы потом можно было собрать их воедино.

Доступ к обычным методам через указатели

Наш первый пример покажет, что бывает, когда базовый и производные классы содержат функции с одним и тем же именем, и к ним обращаются с помощью указателей, но без использования виртуальных функций.

Листинг 11.1. Программа NOTVIRT¹

```
// notvirt.cpp
// Доступ к обычным функциям через указатели
#include <iostream>
using namespace std;
////////////////////////////////////
class Base //Базовый класс
{
public:
    void show() //Обычная функция
    { cout << "Base\n"; }
};
////////////////////////////////////
class Derv1 : public Base //Производный класс 1
{
public:
```

продолжение ⇨

¹ Все исходные тексты, приведенные в книге, вы найдете по адресу <http://www.piter.com/download>.

Листинг 11.1 (продолжение)

```

void show()
{ cout << "Derv1\n"; }
};

////////////////////////////////////
class Derv2 : public Base //Производный класс 2
{
public:
void show()
{ cout << "Derv2\n"; }
};

////////////////////////////////////
int main()
{
Derv1 dv1:           //Объект производного класса 1
Derv2 dv2:           //Объект производного класса 2
Base* ptr:           //Указатель на базовый класс

ptr = &dv1:          //Адрес dv1 занести в указатель
ptr->show()           //Выполнить show()
ptr = &dv2:          //Адрес dv2 занести в указатель
ptr->show()           //Выполнить show()
return 0;
}

```

Итак, классы Derv1 и Derv2 являются наследниками класса Base. В каждом из этих трех классов имеется метод show(). В main() мы создаем объекты классов Derv1 и Derv2, а также указатель на класс Base. Затем адрес объекта порожденного класса мы заносим в указатель базового класса:

```
ptr = &dv1: //Адрес объекта порожденного класса заносим в //указатель базового
```

Но постойте, а компилятор на нас не обидится за то, что мы присваиваем адрес объекта одного типа указателю на другой тип? Оказывается, наоборот — компилятор будет просто счастлив, потому что проверка типов отдыхает в этой ситуации. Мы скоро поймем, почему. Дело в том, что указатели на объекты порожденных классов совместимы по типу с указателями на объекты базового класса.

Теперь хорошо бы понять, какая же, собственно, функция выполняется в этой строчке:

```
ptr->show();
```

Это функция Base::show() или Derv1::show()? Опять же, в последних двух строчках программы NOTVIRT мы присвоили указателю адрес объекта, принадлежащего классу Derv2, и снова выполнили

```
ptr->show();
```

Так какая же из функций show() реально вызывается? Результат выполнения программы дает простой ответ:

```
Base
Base
```

Как видите, всегда выполняется метод базового класса. Компилятор не смотрит на *содержимое* указателя ptr, а выбирает тот метод, который удовлетворяет *типу* указателя, как показано на рис. 11.1.

Да, иногда именно это нам и нужно, но таким образом не решить поставленную в начале этой темы проблему доступа к объектам разных классов с помощью одного выражения.

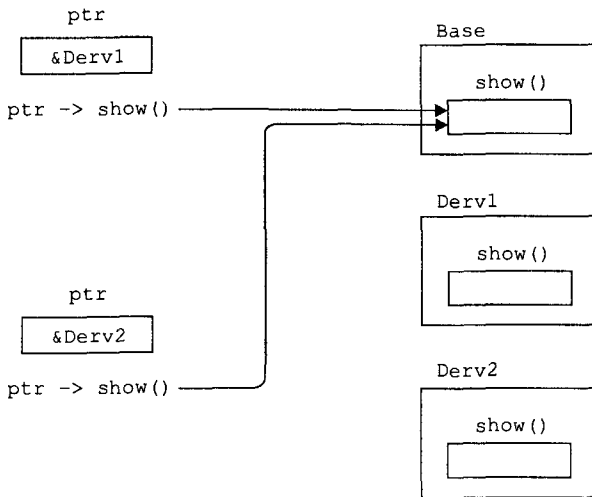


Рис. 11.1. Доступ через указатель без использования виртуальных функций

Доступ к виртуальным методам через указатели

Давайте сделаем одно маленькое изменение в нашей программе: поставим ключевое слово `virtual` перед объявлением функции `show()` в базовом классе. Вот листинг того, что получилось — программы VIRT:

Листинг 11.2. Программа VIRT

```
// virt.cpp
// Доступ к виртуальным функциям через указатели
#include <iostream>
using namespace std;
////////////////////////////////////
class Base                //Базовый класс
{
public:
    virtual void show()    //Виртуальная функция
    { cout << "Base\n": }
};
////////////////////////////////////
class Derv1 : public Base //Производный класс 1
{
public:
    void show()
    { cout << "Derv1\n": }
};
////////////////////////////////////
class Derv2 : public Base //Производный класс 2
```

Листинг 11.2 (продолжение)

```

{
    public:
    void show()
    { cout << "Derv2\n"; }
};
////////////////////////////////////
int main()
{
    Derv1 dv1;           //Объект производного класса 1
    Derv2 dv2;           //Объект производного класса 2
    Base* ptr;           //Указатель на базовый класс

    ptr = &dv1;          //Адрес dv1 занести в указатель
    ptr->show()           //Выполнить show()
    ptr = &dv2;          //Адрес dv2 занести в указатель
    ptr->show()           //Выполнить show()
    return 0;
}

```

На выходе имеем:

Derv1
Derv2

Теперь, как видите, выполняются методы производных классов, а не базового. Мы изменили содержимое ptr с адреса из класса Derv1 на адрес из класса Derv2, и изменилось реальное выполнение show(). Значит, один и тот же вызов

ptr->show()

ставит на выполнение разные функции в зависимости от содержимого ptr. Компилятор выбирает функцию, удовлетворяющую тому, что занесено в указатель, а не типу указателя, как было в программе NOTVIRT. Это показано на рис. 11.2.

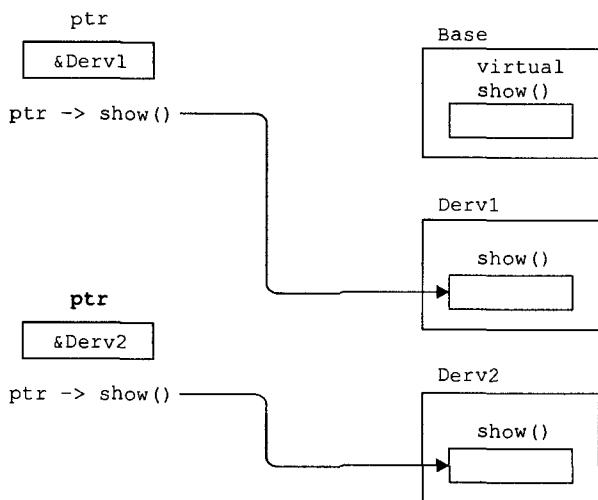


Рис. 11.2. Доступ через указатель к виртуальным функциям

Позднее связывание

Любознательный читатель может удивиться, как же компилятор узнает, какую именно функцию ему компилировать? В программе NOTVIRT у компилятора нет проблем с выражением

```
ptr->show();
```

Он всегда компилирует вызов функции `show()` из базового класса. Однако в программе VIRT компилятор не знает, к какому классу относится содержимое `ptr`. Ведь это может быть адрес объекта как класса `Derv1`, так и класса `Derv2`. Какую именно версию `draw()` вызывает компилятор — тоже загадка. На самом деле компилятор не очень понимает, что ему делать, поэтому откладывает принятие решения до фактического запуска программы. А когда программа уже поставлена на выполнение, когда известно, на что указывает `ptr`, тогда будет запущена соответствующая версия `draw`. Такой подход называется *поздним связыванием* или *динамическим связыванием*. (Выбор функций в обычном порядке, во время компиляции, называется *ранним связыванием* или *статическим связыванием*.) Позднее связывание требует больше ресурсов, но дает выигрыш в возможностях и гибкости.

Вскоре мы претворим эти идеи в жизнь, а сейчас вернемся к виртуальным функциям.

Абстрактные классы и чистые виртуальные функции

Давайте вернемся к классу `shape` из программы MULTSHAP (глава 9). Мы никогда не станем создавать объект из класса `shape`, разве что начертим несколько геометрических фигур — кругов, треугольников и т. п. Базовый класс, объекты которого никогда не будут реализованы, называется *абстрактным классом*. Такой класс может существовать с единственной целью — быть родительским по отношению к производным классам, объекты которых будут реализованы. Еще он может служить звеном для создания иерархической структуры классов.

Как нам объяснить людям, использующим созданную нами структуру классов, что объекты родительского класса не предназначены для реализации? Можно, конечно, заявить об этом в документации, но это никак не защитит наш базовый класс от использования не по назначению. Надо защитить его программно. Для этого достаточно ввести в класс хотя бы одну *чистую виртуальную функцию*. Чистая виртуальная функция — это функция, после объявления которой добавлено выражение `=0`. Продемонстрируем сказанное в примере VIRTPURE:

Листинг 11.3. Программа VIRTPURE

```
virtpure.cpp
./ Чистая виртуальная функция
#include <iostream>
using namespace std;
////////////////////////////////////
```

продолжение →

Листинг 11.3 (продолжение)

```

class Base                                //базовый класс
{
public:
    virtual void show() = 0; //чистая виртуальная
                             //функция
};
////////////////////////////////////
class Derv1 : public Base //порожденный класс 1
{
public:
    void show()
    { cout << ?Derv1\n?: }
};
////////////////////////////////////
class Derv2 : public Base //порожденный класс 2
{
public:
    void show()
    { cout << ?Derv2\n?: }
};
////////////////////////////////////
int main()
{
// Base bad:                        //невозможно создать объект
                                    //из абстрактного класса
    Base* arr[2]:                    //массив указателей на
                                    //базовый класс
    Derv1 dv1:                       //Объект производного класса 1
    Derv2 dv2:                       //Объект производного класса 2

    arr[0] = &dv1:                   //Занести адрес dv1 в массив
    arr[1] = &dv2:                   //Занести адрес dv2 в массив

    arr[0]->show():                  //Выполнить функцию show()
    arr[1]->show():                  //над обоими объектами
    return 0;
}

```

Здесь виртуальная функция `show()` объявляется так:

```
virtual void show() = 0; //чистая виртуальная функция
```

Знак равенства не имеет ничего общего с операцией присваивания. Нулевое значение ничему не присваивается. Конструкция `=0` — это просто способ сообщить компилятору, что функция будет чистой виртуальной. Если теперь в `main()` попытаться создать объект класса `Base`, то компилятор будет недоволен тем, что объект абстрактного класса пытаются реализовать. Он выдаст даже имя чистой виртуальной функции, которая делает класс абстрактным. Помните, что хотя это только объявление, но определение функции `show()` базового класса не является обязательным. Впрочем, если вам надо его написать, это можно сделать.

Как только в базовом классе окажется чистая виртуальная функция, необходимо будет позаботиться о том, чтобы избежать ее употребления во всех производных классах, которые вы собираетесь реализовать. Если класс использует чистую виртуальную функцию, он сам становится абстрактным, никакие объекты

из него реализовать не удастся (производные от него классы, впрочем, уже не имеют этого ограничения). Более из эстетических соображений, нежели из каких-либо иных, можно все виртуальные функции базового класса сделать чистыми.

Между прочим, мы внесли еще одно, не связанное с предыдущими, изменение в VIRTPURE: теперь адреса методов хранятся в массиве указателей и доступны как элементы этого массива. Обработка этого, впрочем, ничем не отличается от использования единственного указателя. VIRTPURE выдает результат, не отличающийся от VIRT:

```
Derv1
Derv2
```

Виртуальные функции и класс person

Теперь, уже зная, что такое виртуальные функции, рассмотрим области их применения. Примером будет служить расширение программ PTROBJ и PERSORT из главы 10 «Указатели». Новая программа использует все тот же класс person, но добавлены два новых класса: student и professor. Каждый из этих производных классов содержит метод isOutstanding(). С помощью этой функции администрация школы может создать список выдающихся педагогов и учащихся, которых следует наградить за их успехи. Листинг программы VIRTPERS:

Листинг 11.4. Программа VIRTPERS

```
// vitrpers.cpp
// виртуальные функции и класс person
#include <iostream>
using namespace std;
////////////////////////////////////
class person                                //класс person
{
protected:
    char name[40];
public:
    void getName()
    { cout << " Введите имя: "; cin >> name; }
    void putName()
    { cout << " Имя: " << name << endl; }
    virtual void getData() = 0;           //чистые
    virtual bool isOutstanding() = 0;     //виртуальные
                                        //функции
};
////////////////////////////////////
class student : public person             //класс student
{
private:
    float gpa;                           //средний балл
public:
    void getData()                        //запросить данные об ученике у
    {                                     //пользователя
        person::getName();
        cout << " Средний балл ученика: "; cin >> gpa;
    }
}
```

Листинг 11.4 (продолжение)

```

    bool isOutstanding()
    { return (gpa > 3.5) ? true : false; }
};
////////////////////////////////////
class professor : public person //класс professor
{
private:
    int numPubs;           //число публикаций
public:
    void getData()          //запросить данные о педагоге у
    {                       //пользователя
        person::getName();
        cout << " Число публикаций: "; cin >> gpa;
    }
    bool isOutstanding()
    { return (gpa > 100) ? true : false; }
};
////////////////////////////////////
int main()
{
    person* persPTR[100]; //массив указателей на person
    int n = 0;            //число людей, внесенных в список char choice;

    do {
        cout << " Учащийся (s) или педагог (p): ";
        cin >> choice;
        if(choice=='s' )           //Занести нового ученика
            persPtr[n] = new student; // в массив
        else                       //Занести нового
            persPtr[n] = new professor; //педагога в массив
        persPtr[n++]->getData();    //Запрос данных о персоне
        cout << " Ввести еще персону (y/n)? "; //создать еще
                                           //персону
        cin >> choice;
    } while( choice=='y' );        //цикл, пока ответ 'y'

    for(int j=0; j<n; j++){
        persPtr[j]->putName();      //Вывести все имена,
        if( persPtr[j]->isOutstanding() ) //сообщать о
            cout << " Это выдающийся человек!\n"; //выдающихся
    }
    return 0;
}
//Конец main()

```

Классы

Класс `person` — абстрактный, так как содержит чистые виртуальные функции `getData()` и `isOutstanding()`. Никакие объекты класса `person` не могут быть созданы. Он существует только в качестве базового класса для `student` и `professor`. Эти порожденные классы добавляют новые экземпляры данных для базового класса. `Student` содержит переменную `gpa` типа `float`, представляющую собой средний балл учащегося. В классе `Professor` мы создали переменную `numPubs` типа `int`, которая представляет собой число публикаций педагога. Учащиеся со средним баллом свыше 3,5 и педагоги, опубликовавшие более 100 статей, считаются выда-

ющимися. (Мы, пожалуй, воздержимся от комментариев по поводу критериев оценки. Будем считать, что они выбраны условно.)

Функция `isOutstanding()`

`isOutstanding()` объявлена чистой виртуальной функцией в классе `person`. В классе `student` эта функция возвращает `true`, если `gpa` больше 3,5, в противном случае — `false`. В классе `professor` она возвращает `true`, если `numPubs` больше 100.

Функция `GetData()` запрашивает у пользователя GPA, если она запускается для обслуживания класса `student`, или число публикаций для `professor`.

Программа `main()`

В функции `main()` мы вначале даем пользователю возможность ввести несколько имен учащихся и педагогов. К тому же программа спрашивает средний балл учащихся и число публикаций педагогов. После того, как пользователь закончит ввод данных, программа выводит на экран имена всех учащихся и педагогов, пометчая выдающихся. Приведем пример работы программы:

```
Учащийся (s) или педагог (p): s
Введите имя: Сергеев Михаил
Средний балл ученика: 1.2
Ввести еще персону (y/n)? y
Учащийся (s) или педагог (p): s
Введите имя: Пупкин Василий
Средний балл ученика: 3.9
Ввести еще персону (y/n)? y
Учащийся (s) или педагог (p): s
Введите имя: Борисов Владимир
Средний балл ученика: 4.9
Ввести еще персону (y/n)? y
Учащийся (s) или педагог (p): p
Введите имя: Михайлов Сергей
Число публикаций: 714
Ввести еще персону (y/n)? y
Учащийся (s) или педагог (p): p
Введите имя: Владимиров Борис
Число публикаций: 13
Ввести еще персону (y/n)? n
```

```
Имя: Сергеев Михаил
Имя: Пупкин Василий
Это выдающийся человек!
Имя: Борисов Владимир
Это выдающийся человек!
Имя: Михайлов Сергей
Это выдающийся человек!
Имя: Владимиров Борис
```

Виртуальные функции в графическом примере

Давайте рассмотрим еще один пример использования виртуальных функций. На сей раз это будет доработка программы `MULTSHAP` из главы 9 «Наследование». Как уже отмечалось в начале этой темы, вам может понадобиться нарисовать

некоторое число фигур с помощью одного и того же выражения. Это с успехом осуществляет программа VIRTSHAP. Помните, что компилировать эту программу следует, используя соответствующий файл для рисования графики, который описан в приложении Г «Упрощенный вариант консольной графики».

Листинг 11.5. Программа VIRTSHAP

```
// virtshap.cpp
// Виртуальные функции и геометрические фигуры
#include <iostream>
using namespace std;
#include "msoftcon.h"           //для графических функций
////////////////////////////////////
class shape                     //базовый класс
{
protected:
    int xCo, yCo;               //координаты центра
    color fillColor;            //цвет
    fstyle fillstyle;           //заполнение
public:                         //конструктор без аргументов
    shape() : xCo(0), yCo(0), fillColor(cWHITE),
fillstyle(SOLID_FILL)
    { }                         //конструктор с четырьмя аргументами
    shape(int x, int y, color fc, fstyle fs) :
        xCo(x), yCo(y), fillColor(fc), fillstyle(fs)
    { }
    virtual void draw() = 0 //чистая виртуальная функция
    {
        set_color(fillColor);
        set_fill_style(fillstyle);
    }
};
////////////////////////////////////
class ball : public shape
{
private:
    int radius;                //центр с координатами(xCo, yCo)
public:
    ball() : shape()           //конструктор без аргументов
    { }
    ball(int x, int y, int r, color fc, fstyle fs)
        : shape(x, y, fc, fs), radius(r)
    { }
    void draw()                //нарисовать шарик
    {
        shape::draw();
        draw_circle(xCo, yCo, radius);
    }
};
////////////////////////////////////
class rect : public shape
{
private:
    int width, height;         //(xCo, yCo) - верхний левый угол
public:
    rect() : shape(), height(0), width(0) //конструктор //без аргументов
    { }
    rect(int x, int y, int h, int w, color fc, fstyle fs) :
        //конструктор с шестью аргументами
```

```

        shape(x, y, fc, fs). height(h). width(w)
    { }
void draw()          //нарисовать прямоугольник
{
    shape::draw();
    draw_rectangle(xCo, yCo, xCo+width, yCo+height);
    set_color(cWHITE); //нарисовать диагональ
    draw_line(xCo, yCo, xCo+width, yCo+height);
}
};
////////////////////////////////////
class tria : public shape
{
private:
    int height;          //(xCo, yCo) - вершина пирамиды
public:
    tria() : shape(). height(0) //конструктор без аргументов
    { } //конструктор с пятью аргументами
    tria(int x, int y, int h, color fc, fstyle fs) :
        shape(x, y, fc, fs). height(h)
    { }
    void draw()          //нарисовать треугольник
    {
        shape::draw();
        draw_pyramid(xCo, yCo, height);
    }
};
////////////////////////////////////
int main()
{
    int j;
    init_graphics();          //инициализация графики
    shape* pShapes[3];        //массив указателей на фигуры
                                //определить три фигуры
    pShapes[0] = new ball(40, 12, 5, cBLUE, X_FILL);
    pShapes[1] = new rect(12, 7, 10, 15, cRED, SOLID_FILL);
    pShapes[2] = new tria(60, 7, 11, cGREEN, MEDIUM_FILL);

    for(j=0; j<3; j++)        //нарисовать все фигуры
        pShapes[j]->draw();

    for(j=0; j<3; j++)        //удалить все фигуры
        delete pShapes[j];
    set_cursor_pos(1, 25);
    return 0;
}

```

Спецификаторы классов в этой программе аналогичны спецификаторам из программы MULTSHAP, за исключением того, что draw() в классе shape стала чистой виртуальной функцией.

В main() мы задаем массив ptarr указателей на фигуры. Затем создаем три объекта, по одному из каждого класса, и помещаем их адреса в массив. Теперь с легкостью можно нарисовать все три фигуры:

```
ptarr[j]->draw();
```

Переменная j при этом меняется в цикле.

Как видите, это довольно мощный инструмент для соединения большого числа различных графических элементов в одно целое.

Виртуальные деструкторы

Знаете ли вы, что деструкторы базового класса обязательно должны быть виртуальными? Допустим, чтобы удалить объект порожденного класса, вы выполнили `delete` над указателем базового класса, указывающим на порожденный класс. Если деструктор базового класса не является виртуальным, тогда `delete`, будучи обычным методом, вызовет деструктор для базового класса вместо того, чтобы запустить деструктор для порожденного класса. Это приведет к тому, что будет удалена только та часть объекта, которая относится к базовому классу. Программа `VIRTDEST` демонстрирует это.

Листинг 11.6. Программа `VIRTDEST`

```
//vertdest.cpp
//Тест неvirtуальных и virtуальных деструкторов
#include <iostream>
using namespace std;
////////////////////////////////////
class Base
{
public:
    ~Base()                //невirtуальный деструктор
//    virtual ~Base()      //virtуальный деструктор
    { cout << "Base удален\n"; }
};
////////////////////////////////////
class Derv : public Base
{
public:
    ~Derv()
    { cout << "Derv удален\n"; }
};
////////////////////////////////////
int main()
{
    Base* pBase = new Derv;
    delete pBase;
    return 0;
}
```

Программа выдаст такой результат:

```
Base удален
```

Это говорит о том, что деструктор для `Derv` не вызывается вообще! К такому результату привело то, что деструктор базового класса в приведенном листинге неvirtуальный. Исправить это можно, закомментировав первую строчку определения деструктора и активизировав вторую. Теперь результатом работы программы является:

```
Derv удален
Base удален
```

Только теперь обе части объекта порожденного класса удалены корректно. Конечно, если ни один из деструкторов ничего особенно важного не делает (например, просто освобождает память, занятую с помощью `new`), тогда их виртуальность перестает быть такой уж необходимой. Но в общем случае, чтобы быть уверенным в том, что объекты порожденных классов удаляются так, как нужно, следует всегда делать деструкторы в базовых классах виртуальными.

Большинство библиотек классов имеют базовый класс, в котором есть виртуальный деструктор, что гарантирует нам наличие виртуальных деструкторов в порожденных классах.

Виртуальные базовые классы

Прежде чем расстаться с темой виртуальных элементов программирования, нам следует коснуться вопроса *виртуальных базовых классов*, так как они имеют отношение к множественному наследованию.

Рассмотрим ситуацию, представленную на рис. 11.3. Базовым классом является `parent`, есть два порожденных класса — `child1`, `child2` и есть еще четвертый класс — `Grandchild`, порожденный одновременно классами `Child1` и `Child2`.

В такой ситуации проблемы могут возникнуть, если метод класса `Grandchild` захочет получить доступ к данным или функциям класса `Parent`. Что в этом случае будет происходить, показано в программе `NORMBASE`.

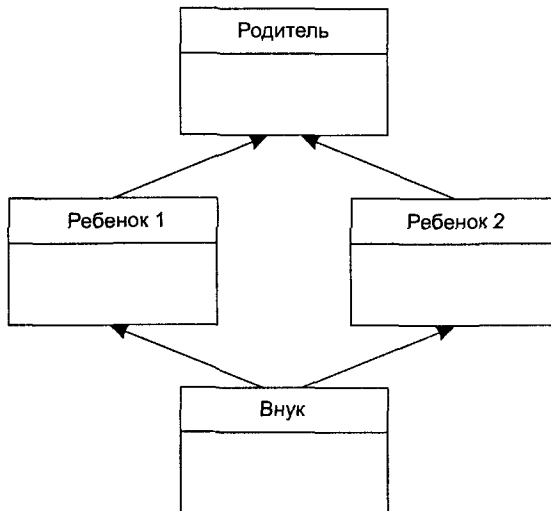


Рис. 11.3. Виртуальные базовые классы

Листинг 11.7. Программа `NORMBASE`

```
// normbase.cpp
// неоднозначная ссылка на базовый класс
class Parent
{
protected:
```

продолжение ➤

Листинг 11.7 (продолжение)

```

    int basedata;
};
class Child1 : public Parent
{
};
class Child2 : public Parent
{
};
class Grandchild : public Child1, public Child2
{
public:
    int getdata()
    { return basedata; } // ОШИБКА: неоднозначность
};

```

Ошибка компилятора возникла вследствие того, что метод `Getdata()` из класса `Grandchild` попытался получить доступ к `basedata` из класса `parent`. И что же в этом такого? Все дело в том, что каждый из порожденных классов (`Child1` и `Child2`) наследует свою копию базового класса `Parent`. Эта копия называется *подобъектом*. Каждый из двух подобъектов содержит собственную копию данных базового класса, включая `basedata`. Затем, когда `Grandchild` ссылается на `basedata`, к какой из двух копий базового класса он получает доступ? Ситуация неоднозначная, о чем компилятор и сообщает.

Для устранения неоднозначности сделаем `Child1` и `Child2` наследниками виртуального базового класса, как показано в примере `VIRTBASE`.

Листинг 11.8. Программа VIRTBASE

```

// virtbase.cpp
// Виртуальные базовые классы
class Parent
{
protected:
    int basedata;
};
class Child1 : virtual public Parent // наследует копию
                                     // класса Parent
{
};
class Child2 : virtual public Parent // наследует копию
                                     // класса Parent
{
};
class Grandchild : public Child1, public Child2
{
public:
    int getdata()
    { return basedata; } // OK: только одна копия
                       // класса Parent
};

```

Использование ключевого слова `virtual` в этих двух классах приводит к тому, что они наследуют единый общий подобъект базового класса `Parent`. Так как теперь у нас есть только одна копия `basedata`, неоднозначность при обращении к базовому классу устраняется.

Необходимость использования виртуального базового класса показывает, что при множественном наследовании возникает ряд концептуальных проблем, поэтому такой подход должен использоваться осторожно.

Дружественные функции

Принцип инкапсуляции и ограничения доступа к данным запрещает функциям, не являющимся методами соответствующего класса, доступ к скрытым (`private`) или защищенным данным объекта. Политика этих принципов ООП такова, что, если функция не является членом объекта, она не может пользоваться определенным рядом данных. Тем не менее есть ситуации, когда такая жесткая дискриминация приводит к значительным неудобствам.

Дружественные функции как мосты между классами

Представьте, что вам необходимо, чтобы функция работала с объектами двух разных классов. Например, функция будет рассматривать объекты двух классов как аргументы и обрабатывать их скрытые данные. В такой ситуации ничто не спасет, кроме `friend`-функции. Приведем простой пример `FRIEND`, дающий представление о работе дружественных функций в качестве мостов между двумя классами.

Листинг 11.9. Программа `FRIEND`

```
// friend.cpp
// Дружественные функции
#include <iostream>
using namespace std;
////////////////////////////////////
class beta;           //нужно для объявления frifunc

class alpha
{
private:
    int data;
public:
    alpha() : data(3) { }           //конструктор без
                                   //аргументов
    friend int frifunc(alpha, beta); //дружественная
                                   //функция
};
////////////////////////////////////
class beta
{
private:
    int data;
public:
    beta() : data(7) { }           //конструктор без
                                   //аргументов
    friend int frifunc(alpha, beta); //дружественная
                                   //функция
};
////////////////////////////////////
int frifunc(alpha a, beta b)       //определение функции
```

Листинг 11.9 (продолжение)

```

    {
        return( a.data + b.data );
    }
//-----
int main()
{
    alpha aa;
    beta bb;

    cout << frifunc(aa, bb) << endl;    //вызов функции
    return 0;
}

```

В этой программе мы видим два класса — `alpha` и `beta`. Конструкторы этих классов задают их единственные элементы данных в виде фиксированных значений (3 и 7 соответственно).

Нам необходимо, чтобы функция `frifunc()` имела доступ и к тем, и к другим скрытым данным, поэтому мы делаем ее дружественной функцией. Этой цели в объявлениях внутри каждого класса служит ключевое слово `friend`:

```
friend int frifunc(alpha, beta);
```

Это объявление может быть расположено где угодно внутри класса. Нет никакой разницы, запишем мы его в `public`- или в `private`-секцию.

Объект каждого класса передается как параметр функции `frifunc()`, и функция имеет доступ к скрытым данным обоих классов посредством этих аргументов. Функция выполняет не очень сложную работу по складыванию значений данных и выдаче их суммы. В `main()` осуществляется вызов этой функции и выводится на экран результат.

Напоминаем, что к классу нельзя обращаться до того, как он объявлен в программе. Объявление функции `frifunc()` (класс `alpha`) ссылается на класс `beta`, а значит, `beta` должен быть объявлен до `alpha`. Отсюда определение

```
class beta:
```

в начале программы.

Ломая стены

Надо отметить, что идея дружественных функций несколько сомнительна. Во время разработки C++ на эту тему велись споры и приводились аргументы против включения в язык такой возможности. С одной стороны, дружественные функции повышают гибкость языка, но, с другой стороны, они не соответствуют *принципу ограничения доступа к данным*, который гласит, что только функции-члены могут иметь доступ к скрытым данным класса.

Насколько серьезно противоречит единству концепции использование дружественных функций? Дружественная функция объявляется таковой в том классе, к данным которого она захочет впоследствии получить доступ. Таким образом, программист, не имеющий доступа к исходному коду класса, не может сделать функцию дружественной. В этом смысле целостность данных сохраняется.

Но все равно такие функции принципиально нечистоплотны и потенциально могут привести к «spaghetti-code» (код со слишком большим числом передач управления), когда множество дружественных функций фактически стирают границы между классами. По этой причине дружественные функции должны быть по крайней мере разбросаны по программе и встречаться как можно реже. Если вы обнаружили, что используете слишком много этих функций, стоит подумать об изменении программы.

Пример с английскими мерами длины

Иногда дружественные функции слишком удобны, чтобы от них отказываться. Наверное, самым общим примером является ситуация, когда они используются для повышения многосторонности перегружаемых операций. Следующая программа показывает ограниченность применения таких операторов без использования дружественных функций. Пример является вариацией на тему ENGLPLUS и ENGLCONV из главы 8 «Перегрузка операций». Программа называется NOFRI.

Листинг 11.10. Программа NOFRI

```
// nofri.cpp
// ограничение перегрузки оператора «+»
#include <iostream>
using namespace std;
////////////////////////////////////
class Distance          //Класс английских расстояний
{
private:
    int feet;
    float inches;
public:
    Distance() : feet(0), inches(0.0) //конструктор
                                     // (без аргументов)
    { }                               //конструктор (1 арг.)
    Distance(float fltfeet)           //переводит float в
                                     //Distance
    {                                 //feet – это целая часть
        feet = static_cast<int>(fltfeet);
        inches = 12*(fltfeet-feet);   //слева – дюймы
    }
    Distance(int ft, float in)        //конструктор (2 арг.)
    { feet = ft; inches = in; }
    void showdist()                   //вывод на экран расстояния
    { cout << feet << "\'-" << inches << "\'": }
    Distance operator + (Distance);
};
//-----
//прибавить расстояние к d2
Distance Distance::operator + (Distance d2) //сумма
{
    int f = feet + d2.feet;           //добавить футы
    float i = inches + d2.inches;     //добавить дюймы
    if(i >= 12.0)                     //если сумма превышает 12.0.
    { i -= 12.0; f++; }               //уменьшить на 12 дюймов.
                                     //прибавить 1 фут
}
```

Листинг 11.10 (продолжение)

```

    return Distance(f,i);          //Новый Distance с суммой
}
////////////////////////////////////
int main()
{
    Distance d1 = 2.5;             //конструктор переводит
    Distance d2 = 1.25;           //feet типа float в Distance
    Distance d3;
    cout << "\nd1 = "; d1.showdist();
    cout << "\nd2 = "; d2.showdist();

    d3 = d1 + 10.0;               //distance + float: OK
    cout << "\nd3 = "; d3.showdist();
    // d3 = 10.0 + d1;            //float + Distance: ОШИБКА
    // cout << "\nd3 = "; d3.showdist();
    cout << endl;
    return 0;
}

```

В этой программе оператор «+» переопределен на сложение двух объектов типа `Distance`. К тому же есть конструктор с одним аргументом, который переводит значение типа `float`, представляющее футы и десятые части футов, в значение типа `Distance` (то есть переводит числа из формата 10.25' в 10'-3").

Когда есть такой конструктор, можно написать такое выражение в `main()`:

```
d3 = d1 + 10.0;
```

Перегружаемому + нужны объекты типа `Distance` и справа, и слева, но, найдя справа аргумент типа `float`, в соответствии с конструктором он переводит этот `float` в `distance` и осуществляет сложение.

Но вот что происходит при едва уловимом изменении в выражении:

```
d3 = 10.0 + d1;
```

Разве это выражение обрабатывается как следует? К сожалению, нет, так как объект, чьим методом является перегружаемый +, должен быть переменной, находящейся *слева* от оператора. Когда мы туда помещаем оператор другого типа или константу, компилятор использует + для сложения с этим типом (`float` в данном случае) вместо того, чтобы добавлять объекты типа `Distance`. Увы, этот оператор не умеет конвертировать `float` в `Distance`, поэтому не может выполнить сложение. Вот результат работы `NOFRI`:

```

d1 = 2'-6''
d2 = 1'-3''
d3 = 12'-6''

```

Второе сложение не будет скомпилировано, поэтому эти выражения закоментированы. Выйти из этой ситуации можно, создав новый объект типа `Distance`:

```
d3 = Distance(10.0) + d1;
```

Но такое выражение ни очевидно, ни элегантно. Как написать нормальное, красивое выражение, которое могло бы содержать нужные типы как справа, так и слева от оператора? Как вы уже, наверное, догадались, нам поможет дружественная функция. Программа `FRENGL` демонстрирует, как именно.

Листинг 11.11. Программа FRENGL

```

// frengl.cpp
// Дружественная перегружаемая операция +
#include <iostream>
using namespace std;
////////////////////////////////////
class Distance          //Класс английских расстояний
{
private:
    int feet;
    float inches;
public:
    Distance()           //конструктор без аргументов
    { feet = 0; inches = 0.0; }
    Distance( float fltfeet ) //конструктор (1 арг.)
    { //Переводит float в Distance
      feet = int(fltfeet); //feet - целая часть
      inches = 12*(fltfeet-feet); //слева - дюймы
    }
    Distance(int ft, float in) //конструктор (2 арг.)
    { feet = ft; inches = in; }
    void showdist()           //Вывести длину
    { cout << feet << "\'" << inches << '\\"'; }
    friend Distance operator + (Distance, Distance): //дружественный
};
//-----

Distance operator + (Distance d1, Distance d2) // d1 + d2
{
    int f = d1.feet + d2.feet; //+ футы
    float i = d1.inches + d2.inches; //+ дюймы
    if(i >= 12.0) //если больше 12 дюймов.
    { i -= 12.0; f++; } //уменьшить на 12 дюймов.
    //прибавить 1 фут
    return Distance(f,i); //Новая длина с суммой
}
//-----

int main()
{
    Distance d1 = 2.5; //конструктор переводит
    Distance d2 = 1.25; //float-feet в Distance
    Distance d3;
    cout << "\nd1 = "; d1.showdist();
    cout << "\nd2 = "; d2.showdist();

    d3 = d1 + 10.0; //distance + float: OK
    cout << "\nd3 = "; d3.showdist();
    d3 = 10.0 + d1; //float + Distance: OK
    cout << "\nd3 = "; d3.showdist();
    cout << endl;
    return 0;
}

```

Итак, перегружаемая операция + сделана дружественной:

```
friend Distance operator + (Distance, Distance);
```

Помните, что для того, чтобы сделать функцию дружественной, только объявление функции внутри класса нужно начинать с ключевого слова `friend`. Определение класса пишется без изменений, как и вызов функции.

Иногда дружественная функция позволяет использовать более понятный синтаксис вызова функции, чем обычный метод. Например, нам нужна функция, возводящая в квадрат экземпляр объекта типа `Distance` из приведенного выше примера и возвращающая значение в квадратных футах как тип `float`. Пример `MISQ` показывает, как это может быть сделано с помощью обычного метода.

[illegible]

```
Distance dist(3, 6.0); //конструктор с 2 арг. (3'-6")
float sqft;

sqft = dist.square(); //вычислить квадрат расстояния
//показать расст. и квадрат расст.
cout << "\nРасстояние = "; dist.showdist();
cout << "\nКвадрат расст. = " << sqft << " кв. футов\n";
return 0;
}
```

В `main()` программа создает значение `Distance`, возводит его в квадрат и выводит результат. Вот как это выглядит на экране:

```
Расстояние = 3'-6''
Квадрат расст. = 12.25 кв. футов
```

В `main()` мы использовали выражение

```
sqft = dist.square();
```

чтобы отыскать квадрат `dist` и присвоить его переменной `sqft`. Это выполняется корректно, но если мы все-таки хотим работать с объектами `Distance`, используя тот же синтаксис, что мы используем для работы с обычными числами, видимо, имеет смысл предпочесть функциональную запись:

```
sqft = square(dist);
```

Такого эффекта можно добиться, сделав `square()` дружественной функцией для класса `Distance`. Это показано в программе `FRISQ`.

Листинг 11.13. Программа `FRISQ`

```
// frisq.cpp
// Дружественная для Distance функция square()
#include <iostream>
using namespace std;
////////////////////////////////////
class Distance //Класс английских длин
{
private:
    int feet;
    class Distance //Класс английских длин
    {
private:
        int feet;
        float inches;
public:
        Distance() : feet(0), inches(0.0) //конструктор (без
            //аргументов)
        { }

            //конструктор (2 аргумента)
        Distance(int ft, float in) : feet(ft), inches(in)
        { }
        void showdist() //вывести расстояние
        { cout << feet << "'-" << inches << "'"; }
        friend float square(Distance); //дружественная ф-ция
    };
}
//-----
```

Листинг 11.13 (продолжение)

```

float square(Distance d)           //вернуть квадрат
{                                  //расстояния
    float fltfeet = d.feet + d.inches/12; //конвертировать
                                           //в тип float
    float feetsqrd = fltfeet * fltfeet;   //вычислить квадрат
    return feetsqrd;                     //вернуть квадратные футы
}

////////////////////////////////////
int main()
{
    Distance dist(3. 6.0);             //конструктор с двумя
                                       //аргументами (3'-6")

    float sqft;

    sqft = square(dist);               //вернуть квадрат dist
                                       //Вывести расстояние и его квадрат
    cout << "\nРасстояние = "; dist.showdist();
    cout << "\nКвадрат расст. = " << sqft << " square feet\n";
    return 0;
}

float inches;
public:                             //конструктор (без аргументов)
    Distance() : feet(0), inches(0.0)
{ }                                 //constructor (two args)
    Distance(int ft, float in) : feet(ft), inches(in)
    { }
    void showdist()                 //показать длину
    { cout << feet << "'-" << inches << "'\n"; }
    float square():                 //обычный метод
    { };

//-----
float Distance::square()             //возвращает квадрат
{                                   //расстояния
    float fltfeet = feet + inches/12; //перевод во float
    float feetsqrd = fltfeet * fltfeet; //возведение в
                                           //квадрат
    return feetsqrd;                 //возвратить квадрат
}

////////////////////////////////////
int main()
{
    Distance dist(3. 6.0);           //конструктор с двумя арг. (3'-6")
    float sqft;

    sqft = dist.square();             //вычислить квадрат расстояния
                                       //показать расст. и квадрат расст.
    cout << "\nРасстояние = "; dist.showdist();
    cout << "\nКвадрат расст. = " << sqft << " кв. футов\n";
    return 0;
}

```

Тогда как функция `Square()` в `MISQ` не требовала аргументов, будучи обычным методом, во `FRISQ` ей уже требуются аргументы. Вообще говоря, дружественная функция требует на один аргумент больше, чем метод. Функция `square()` во `FRISQ` похожа на аналогичную в `MISQ`, но она обращается к данным объекта `Distance` как `d.inches` или как `d.feet`, в отличие от обращения `inches` и `feet`.

Дружественные классы

Методы могут быть превращены в дружественные функции одновременно с определением всего класса как дружественного. Программа FRICLASS показывает, как это выглядит на практике.

Листинг 11.14. Программа FRICLASS

```
// friclass.cpp
// Дружественные классы
#include <iostream>
using namespace std;
////////////////////////////////////
class alpha
{
private:
    int data1;
public:
    alpha() : data1(99) { } //конструктор
    friend class beta;      //beta - дружественный класс
};
////////////////////////////////////
class beta
{
//все методы имеют доступ
public:
    //к скрытым данным alpha
    void func1(alpha a) { cout << "\ndata1=" << a.data1;}
    void func2(alpha a) { cout << "\ndata1=" << a.data1;}
};
////////////////////////////////////
int main()
{
    alpha a;
    beta b;

    b.func1(a);
    b.func2(a);
    cout << endl;
    return 0;
}
```

Для класса alpha весь класс beta провозглашен дружественным. Теперь все методы beta имеют доступ к скрытым данным класса alpha (в данной программе это лишь единственная переменная data1).

Обратите внимание: мы объявляем именно класс дружественным, используя выражение

```
friend class beta;
```

А можно было сделать и так: сначала объявить обычный класс beta:

```
class beta
```

затем объявить класс alpha, а внутри определения alpha объявить дружественность класса beta:

```
friend beta;
```

Статические функции

Если вы откроете главу 6 «Объекты и классы», то сможете найти там пример `STATIC`, в котором было дано представление о статических данных. Напоминаем, что статические данные не дублируются для каждого объекта. Скорее, один элемент данных используется всеми объектами класса. В том же примере был показан класс, который запоминал, сколько у него было объектов. Давайте попробуем расширить эту идею, показав, что функции, как и данные, могут быть статическими. Кроме демонстрации статических функций, наша программа создаст класс, который присваивает идентификационный номер (ID) каждому из своих объектов. Это позволяет спросить у объекта, кто он таков, что, кроме всего прочего, бывает полезно при отладке программ. Еще одна задача этого примера состоит в том, чтобы пролить свет на деструкторы и работу с ними.

Листинг 11.15. Программа `STATFUNC`

```
// statfunc.cpp
// Статические функции и ID объектов
#include <iostream>
using namespace std;
/////////////////////////////////////////////////////////////////
class gamma
{
private:
    static int total;           //всего объектов класса
                                //(только объявление)
    int id;                    //ID текущего объекта
public:
    gamma()                    //конструктор без аргументов
    {
        total++;               //добавить объект
        id = total;            //id равен текущему значению total
    }
    ~gamma()                   //деструктор
    {
        total--;
        cout << "Удаление ID " << id << endl;
    }
    static void showtotal() // статическая функция
    {
        cout << "Всего: " << total << endl;
    }
    void showid()             // Нестатическая функция
    {
        cout << "ID: " << id << endl;
    }
};
//-----
int gamma::total = 0;         // определение total
/////////////////////////////////////////////////////////////////
int main()
{
    gamma g1;
    gamma::showtotal();

    gamma g2, g3;
```

```
gamma::showtotal();

g1.showid();
g2.showid();
g3.showid();
cout << "-----конец программы-----\n";
return 0;
}
```

Доступ к статическим функциям

В приведенной программе у нас имеется статический элемент данных под именем `total`, принадлежащий классу `gamma`. В этой переменной хранится информация о том, сколько объектов создано в классе. Ее значение увеличивается конструктором и уменьшается деструктором.

Давайте предположим, что нам нужно иметь доступ к `total`, находясь вне класса. В листинге можно увидеть функцию `showtotal()`, показывающую значение этой переменной. Каким образом получить доступ к этой функции?

Когда некоторый элемент данных объявляется статическим, на весь класс выделяется единственная копия этого элемента, при этом не имеет значения, сколько объектов класса мы создали. На самом деле может вообще не быть никаких объектов, но нам интересно узнать даже об этом факте. Можно создать, конечно, некое чудо в перьях для вызова нужного метода, как в приведенных ниже строчках:

```
gamma dummyObj; //создать объект только для того, чтобы //иметь возможность вызвать функцию
dummyObj.showtotal(); //Вызов функции
```

Но, согласитесь, это выглядит некрасиво. По правилам хорошего тона не следует обращаться к объекту, когда мы делаем что-либо, относящееся к классу в целом. Более целесообразно использовать просто имя класса и оператор явного задания функции:

```
gamma::showtotal(); //Так правильнее
```

Тем не менее, такой прием не работает, если `showtotal()` — обычный метод. В этом случае, действительно, придется написать имя объекта, точку и название функции. Хотите иметь доступ к `showtotal()`, используя только лишь имя класса, — объявляйте ее статической. Это мы и проделали в `STATFUNC`, объявив

```
static void showtotal()
```

Объявленная таким образом функция будет иметь доступ просто через имя класса. Вот пример работы программы:

```
Всего: 1
Всего: 3
ID: 1
ID: 2
ID: 3
-----конец программы-----
Удаление ID 3
Удаление ID 2
Удаление ID 1
```

Здесь мы задаем один объект `g1`, после чего выводим значение `total`, оно равно 1. Потом создаем `g2` и `g3` и снова выводим значение `total`, на сей раз оно уже равно 3.

Нумерация объектов

В класс `gamma` мы поместили еще одну функцию, предназначенную для вывода идентификационных номеров его объектов. Эти ID положены равными переменной `total` в момент создания объекта, так что номер каждого объекта оказывается уникальным. Функция `showID()` выводит ID своего объекта. В нашей программе она вызывается трижды в выражениях

```
g1.showID();  
g2.showID();  
g3.showID();
```

Как показывают результаты работы программы, у каждого объекта имеется действительно свой уникальный номер. `G1` пронумерован 1, `g2` — 2, `g3` — 3.

Деструкторы

После того как мы узнали, что объекты можно нумеровать, покажем одно интересное свойство деструкторов. `STATFUNC` выводит на экран сообщение «Конец программы», это написано в последней строчке кода. Но, как показывает пример работы программы, после вывода этой строчки еще кое-что происходит. А именно, три объекта, которые были созданы в процессе работы программы, должны быть удалены до ее завершения, чтобы восстановился доступ к памяти. Об этом заботится компилятор, выполняя деструктор.

Вставив в деструктор выражение, выводящее на экран сообщение, мы можем наблюдать за процессом его работы. Благодаря введенной нумерации, мы также можем наблюдать, в каком порядке удаляются объекты. Как видим, удаление происходит в порядке, обратном созданию. Можно предположить, что локальные объекты хранятся в стеке, и используется LIFO-подход.

Инициализация копирования и присваивания

Компилятор C++ всегда стоит на вашей стороне, выполняя рутинную работу, которой программиста незначит загружать. Если вы обязуетесь контролировать ситуацию, он может посчитаться с вашим мнением и предоставить некоторые вопросы решать самому, но по умолчанию компилятор все делает сам. Двумя чрезвычайно важными примерами этого процесса являются оператор присваивания и конструктор копирования.

Вы постоянно пользуетесь оператором присваивания, может быть, даже не задумываясь о том, что за ним стоит с точки зрения компилятора. Пусть `a1` и `a2` — некоторые объекты. Несмотря на то, что вы говорите компилятору: «присвоить `a2` значение `a1`», выражение

```
a2 = a1;
```

заставит компилятор копировать данные из `a1` элемент за элементом в `a2`. Таковы действия по умолчанию для оператора присваивания.

Вы, возможно, знакомы с инициализацией переменных. Инициализация одного объекта другим, как в выражении

```
alpha a2(a1):    //инициализировать a2 значением a1
```

вызывает подобные действия. Компилятор создает новый объект `a2`, затем элемент за элементом копирует данные из `a1` в `a2`. Это то, что по умолчанию выполняет конструктор копирования.

Оба этих стандартных действия выполняются без проблем компилятором. Если поэлементное копирование из одного объекта в другой — это то, что вам нужно, то можно больше ничего не предпринимать в этом направлении. Тем не менее, если вы хотите научиться заставлять присваивание или инициализацию выполнять какие-либо более интеллектуальные действия, то придется разобратся в том, как обойти стандартную реакцию на эти операторы. Обсуждать технику перегрузки оператора присваивания и конструктора копирования мы будем отдельно друг от друга, затем соединим полученные знания в один пример, который дает возможность классу `String` более эффективно использовать память. Кроме этого, вы узнаете об объектной диаграмме UML.

Перегрузка оператора присваивания

Следующий коротенький пример демонстрирует технологию перегрузки оператора присваивания.

Листинг 11.16. Программа ASSIGN

```
// assign.cpp
// перегрузка операции присваивания (=)
#include <iostream>
using namespace std;
/////////////////////////////////////////////////////////////////
class alpha
{
private:
    int data;
public:
    alpha()                //конструктор без аргументов
    { }
    alpha(int d)           //конструктор с одним аргументом
    { data = d; }
    void display()         //вывести данные
    { cout << data; }
    alpha operator = (alpha& a) //перегружаемый =
    {
        data = a.data;     //не выполняется автоматически
        cout << "\nЗапущен оператор присваивания";
        return alpha(data); //возвращает копию alpha
    }
};
/////////////////////////////////////////////////////////////////
```

продолжение ➤

Листинг 11.16 (продолжение)

```

int main()
{
    alpha a1(37);
    alpha a2;

    a2 = a1;                //запуск перегружаемого =
    cout << "\na2=": a2.display(); //вывести a2

    alpha a3 = a2;          //НЕ запускается =
    cout << "\na3=": a3.display(); //вывести a3
    cout << endl;
    return 0;
}

```

Класс `alpha` очень прост, в нем содержится только один элемент данных. Конструкторы инициализируют данные, а методы выводят их значения на экран. Вот и все, что тут делается. А новизна примера `ASSIGN` заключается в функции `operator=()`, перегрузившей оператор `=` `operator`.

В `main()` мы определяем переменную `a1` и присваиваем ей значение 37, определяем переменную `a2`, но значения ей не присваиваем. Затем, как видите, присваиваем `a2` значение `a1`:

```
a2 = a1;           //Выражение присваивания
```

Тем самым мы запускаем нашу перегружаемую функцию `operator=()`. Программа `ASSIGN` заканчивается с таким результатом:

```

Запущен оператор присваивания
a2=37
a3=37

```

Инициализация — это не присваивание

В последних двух строчках кода `ASSIGN` мы инициализируем объект `a3` объектом `a2` и выводим его значение на экран. И пусть вас не смущает синтаксис в приведенном ниже выражении. Выражению

```

alpha a3 = a2;    //инициализация копирования, а не
                  //присваивание!

```

соответствует не присваивание, а инициализация. А эффект будет при этом тот же, что при использовании выражения

```

alpha a3(a2);     //альтернативный вариант инициализации
                  //копирования

```

Именно поэтому оператор присваивания выполняется только один раз, что и отражено в результатах работы программы в единственной строчке вызова:

```
Запущен оператор присваивания
```

Ответственность

Когда перегружается оператор присваивания, подразумевается, что за все, что по умолчанию делал этот оператор, отвечает программист. Часто это связано с копированием элементов данных из одного объекта в другой. Класс `alpha` из про-

граммы ASSIGN включает в себя только один элемент — data, поэтому функция `operator=()` копирует его значение с помощью следующего выражения:

```
data = a.data;
```

В обязанности этой функции также входит вывод на экран строки «Запущена операция присваивания», по которой мы можем определить, что функция выполняется.

Передача по ссылке

Аргумент для функции `operator=()` передается по ссылке. Нельзя сказать, что это совершенно необходимо, но обычно именно так и делается. Зачем? Как вы знаете, аргумент, передающийся по значению, создает в памяти собственную копию для функции, которой он передается. Аргумент, передающийся функции `operator=()`, не является исключением. А если объекты большие? Их копии могут занять в памяти очень много места, причем от них нет никакой пользы. Когда же значения аргументов передаются по ссылке, копии не создаются, что помогает экономить память.

Есть и еще одна причина. В определенных ситуациях необходимо отслеживать количество созданных объектов (как в примере STATFUNC, где мы нумеровали созданные объекты). Если же компилятор генерирует еще какие-то объекты всякий раз, когда используется оператор присваивания, то есть шанс сильно завысить оценку их количества. С помощью передачи по ссылке удастся ликвидировать прецеденты ложного создания объектов.

Возврат значений

Как вы уже успели заметить, функция может возвращать результат в вызывающую программу по значению или по ссылке. При возвращении по значению происходит фактически передача результата, а значит, создается копия объекта, которая и возвращается в программу. В вызывающей программе этот вновь созданный объект может присваиваться какому-либо другому объекту или использоваться как-то еще, это уже не так важно. Если же возвращение происходит по ссылке, никакой новый объект не создается. Ссылка на исходный объект — вот и все, что возвращается в качестве результата.

Функция `operator=()` возвращает результат путем создания временного объекта `alpha` и его инициализации с помощью одноаргументного конструктора в выражении

```
return alpha(data)
```

Возвращаемое значение — это копия исходного, а не тот же самый объект, чьим методом является перегружаемая операция `=`. Такой стиль возвращения значений позволяет выстраивать эти функции в цепочку:

```
a3 = a2 = a1;
```

Да, впечатляет, но все равно возвращение результата по значению имеет все те же недостатки, что и передача по значению. Прежде всего, это создание в памяти копии возвращаемого объекта, что может привести к некоторой путанице.

А можем мы вернуть результат по ссылке, используя описатель, показанный ниже, для перегружаемого присваивания?

```
alpha& operator = (alpha& a)    //Неудачная мысль
```

К сожалению, нет, мы не можем осуществить возврат результата по ссылке при работе с локальными переменными данной функции.

Напомним, что локальные переменные, то есть созданные внутри функции и не объявленные статическими, уничтожаются при выходе из функции. Но, как мы знаем, возвращение по ссылке передаст в вызывающую программу только адрес данных, который для локальных переменных указывает на данные, находящиеся внутри функции. При возврате из функции указатель хранит уже какое-то не имеющее смысла значение. Компилятор может сигнализировать о такой ситуации предупреждением. В разделе «Указатель this» этой главы мы покажем, как можно решить эту проблему.

Ограничение на наследование

Оператор присваивания уникален среди остальных операторов тем, что он не может наследоваться. Перегрузив присваивание в базовом классе, вы не сможете использовать ту же функцию в порожденных классах.

Конструктор копирования

Мы уже обсуждали, что можно определять объекты и одновременно их инициализировать другими с помощью двух видов выражений:

```
alpha a3(a2):           //инициализация копирования
alpha a3 = a2;          //инициализация копирования.
                        // альтернативная запись
```

Оба стиля определения включают в себя конструктор копирования, то есть конструктор, создающий новый объект и копирующий в него свои аргументы. По умолчанию этот конструктор производит поэлементное копирование. Это очень похоже на то, что делает оператор присваивания, с той лишь разницей, что конструктор создает новый объект.

Как и оператор присваивания, конструктор копирования может быть перегружен. В программе XOFXREF мы покажем, как это делается.

Листинг 11.17. Программа XOFXREF

```
// xofxref.cpp
// конструктор копирования: X(X&)
#include <iostream>
using namespace std;
////////////////////////////////////
class alpha
{
private:
    int data;
public:
    alpha()                //конструктор без аргументов
```

```

    { }
    alpha(int d)          //конструктор с одним аргументом
    { data = d; }
    alpha(alpha& a)       //конструктор копирования
    {
        data = a.data;
        cout << "\nЗапущен конструктор копирования";
    }
    void display()        //display
    { cout << data; }
    void operator = (alpha& a) //overloaded = operator
    {
        data = a.data;
        cout << "\nЗапущен оператор присваивания";
    }
};
////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
int main()
{
    alpha a1(37);
    alpha a2;

    a2 = a1;              //запуск перегружаемого =
    cout << "\na2=": a2.display(); //вывести a2

    alpha a3(a1);         //запуск конструктора копирования
    // alpha a3 = a1;      //эквивалентное определение a3
    cout << "\na3=": a3.display(); //вывести a3
    cout << endl;
    return 0;
}

```

В этой программе перегружается и оператор присваивания, и конструктор копирования. Перегружаемое присваивание подобно аналогичному из программы ASSIGN. У конструктора копирования имеется один аргумент — объект типа `alpha`, переданный по ссылке. Вот его описатель:

```
alpha(alpha&)
```

Он имеет вид `X(X&)`. Ниже приведен результат работы программы:

```
Запущен оператор присваивания
a2=37
```

Запущен конструктор копирования

```
a3=37
```

Выражение

```
a2 = a1;
```

запускает оператор присваивания, тогда как

```
alpha a3(a1):
```

запускает конструктор копирования. Для последней цели может быть использовано другое эквивалентное выражение:

```
alpha a3 = a1;
```

Как уже было показано, конструктор копирования может быть запущен во время определения объекта. Он также запускается, когда аргументы передаются функциям по значению и когда возвращается результат работы функций. Давайте кратко рассмотрим эти ситуации.

Аргументы функции

Итак, конструктор копирования запускается при передаче по значению аргумента функции. Он создаст копию объекта, с которой функция и работает. Таким образом, если бы функция

```
void func(alpha);
```

была объявлена в XOFXREF и если бы она вызывалась выражением

```
func(a1);
```

то конструктор копирования создал бы копию объекта `a1` для его использования `func()`. Само собой разумеется, что конструктор копирования не запускается при передаче аргумента по ссылке или при передаче указателя на объект. В этих случаях никаких копий не создается, и функция работает с исходной переменной.

Возврат результатов работы функции

Конструктор копирования также создает временный объект, когда значение возвращается из функции в основную программу. Допустим, что в XOFXREF была какая-либо подобная функция:

```
alpha func();
```

и она вызывалась выражением

```
a2 = func();
```

Конструктор копирования в этом случае запустился бы для того, чтобы создать копию возвращаемого функцией `func()` результата, а затем, после запуска оператора присваивания, этот результат был бы сохранен в `a2`.

Почему не просто X(X)?

А надо ли нам использовать ссылку в аргументе конструктора копирования? Может быть, можно просто передать значение? Но нет, компилятор сообщает о «выходе за пределы памяти» при попытке откомпилировать выражение

```
alpha(alpha a)
```

Почему? Да потому, что при передаче аргумента по значению создается его копия. Кто ее создает? Конструктор копирования. Но мы ведь и запускаем конструктор копирования, то есть он пытается запустить самого себя, поймать собственный хвост, до тех пор, пока не выйдет за пределы памяти. Так что в конструктор копирования аргумент должен передаваться по ссылке, что не приведет к созданию копий объекта.

Следите за деструкторами

В разделах «Передача по ссылке» и «Возврат значений» мы обсуждали передачу функциям аргументов по значению и возврат результатов в вызывающую программу по значению. В этих ситуациях также запускается деструктор, когда при выходе из функции уничтожаются временные объекты. Это может вызвать недоумение, если вы не ожидаете такого поворота событий. Мораль сей басни такова: работая с объектами, которым требуется нечто большее, нежели простое поэлементное копирование, передавайте и возвращайте данные по ссылкам везде, где это возможно.

Определение и конструктора копирования, и оператора присваивания

Перегружая оператор присваивания, вы почти наверняка намереваетесь заодно переопределить и конструктор копирования (и наоборот). Дело в том, что вам не всегда требуется использовать эти перегружаемые операции, но, с другой стороны, есть места в программах, где они необходимы. Даже если вы не предполагаете использовать в программе одно либо другое, можно заметить, что компилятор сам использует различные варианты поведения в сложных ситуациях, таких, как, например, передача аргумента по значению или возврат результата по значению.

Фактически, если конструктор класса включает в себя операции по работе с системными ресурсами — памятью, файлами на диске, — вам следует почти всегда перегружать и конструктор копирования, и оператор присваивания, не забывая убедиться в том, что они работают корректно.

Запрещение копирования

Мы обсудили, как настроить процедуру копирования объектов с помощью конструктора копирования и оператора присваивания. Но иногда бывает нужно запретить копирование. Сделать это можно, используя те же инструменты. Например, может быть важно то, что каждый член класса создается с уникальным значением определенного элемента данных, который поставляется в качестве аргумента конструктору. Но вы ведь понимаете, что если создастся копия объекта, ей будет присвоено то же значение, и объект перестанет быть уникальным. Чтобы избежать копирования, просто-напросто перегружайте присваивание и конструктор копирования, делайте их скрытыми членами класса:

```
class alpha
{
private:
    alpha& operator = (alpha&); //Скрытое присваивание
    alpha(alpha&);             //Скрытое копирование
};
```

Как только вы попытаетесь скопировать информацию

```
alpha a1, a2;
a1 = a2;           //присваивание
alpha a3(a1);      //копирование
```

компилятор сообщит, что функция недоступна. Вам нет никакой необходимости определять функции, которые никогда не будут вызваны.

Объектные диаграммы UML

В предыдущих главах мы видели примеры диаграмм классов. Наверное, не будет сюрпризом то, что UML поддерживает и объектные диаграммы. Эти диаграммы отображают некоторые определенные объекты (например, объект `Mike_Gonzalez` для класса `Professor`). Поскольку отношения между объектами могут меняться во время работы программы, объекты могут даже добавляться и исчезать, объектная диаграмма представляет собой некую мгновенную картину состояния объектов. Это называют *статической* диаграммой UML.

Объектную диаграмму можно использовать для моделирования какого-либо конкретного действия, выполняемого программой. Вы как бы приостанавливаете программу и смотрите на текущее состояние объектов в том аспекте, который вас интересует, и на соотношения объектов в этот момент времени.

В объектной диаграмме объекты представляются в виде прямоугольников как классы в диаграмме классов. Имя, атрибуты и операции отображаются аналогично. Отличить объекты от классов можно по тому, что их названия подчеркнуты. В строке названия можно использовать имя объекта и имя класса, разделенные двоеточием:

```
anObj:aClass
```

Если имя объекта вам неизвестно (например, потому что доступ к нему осуществляется только через указатель), можно использовать просто имя класса, начинающееся с двоеточия:

```
:aClass
```

Линии между объектами называются *связями* и отображают, разумеется, связь между объектами. Возможности навигации между объектами также могут быть показаны на диаграмме. Значение атрибута показывается с помощью знака равенства:

```
count = 0
```

Еще одним достоинством UML считаются *примечания*. Они показываются в прямоугольниках с загнутым уголком и содержат комментарии или пояснения. Пунктирная линия соединяет примечание с соответствующим элементом диаграммы. В отличие от связей и соединений, примечание может ссылаться на элемент внутри прямоугольника, содержащего класс или объект. Примечания могут использоваться в любых типах диаграмм UML.

В этой главе вы увидите довольно много объектных диаграмм.

Эффективное использование памяти классом `String`

Программы `ASSIGN` и `X0FXREF` на самом деле вовсе не нуждаются в перегрузке присваиваний и конструкторов копирования. Ведь они используют крайне простые

классы с единственным элементом данных, поэтому присваивание и копирование по умолчанию работали бы вполне нормально. Но сейчас мы рассмотрим пример, в котором крайне важно использовать перегрузку этих операторов.

Недостатки класса String

Мы уже рассмотрели на разные версии нашего кустарного класса String в предыдущих главах. И эти версии не являются на самом деле оптимальными. Было бы здорово, наверное, перегрузить оператор присваивания, тогда мы смогли бы присваивать значение одного объекта класса String другому с помощью выражения `s2 = s1`:

Но, если мы и вправду перегрузим оператор присваивания, встанет вопрос, как работать с реальными строками (то есть массивами символьного типа char), которые являются принципиальными элементами данных для класса String.

Можно сделать так, чтобы каждый объект имел специально отведенное «место» для хранения строки. Тогда, присваивая один объект класса String другому, мы просто копируем строку из исходного объекта в объект назначения. Но если вы все же хотите сэкономить память, то необходимо обеспокоиться тем, чтобы одна и та же строка не дублировалась попусту в двух и более местах памяти. Поверьте, это не слишком эффективный подход, особенно учитывая то, что строчки могут быть довольно длинными. На рис. 11.4 показано, как это странно выглядит:

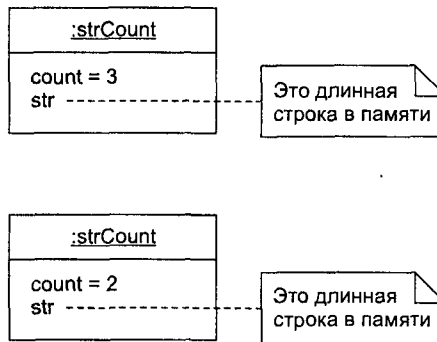


Рис. 11.4. Объектная диаграмма UML: дублирование строк

Вместо того чтобы держать в каждом объекте класса String символьную строку, можно занести в объект только лишь *указатель* на строку! Теперь, присваивая значения одного объекта другому, мы копируем только указатель из одного объекта в другой. Оба указателя, разумеется, будут указывать на одну и ту же строку. Да, это действительно намного лучше, ведь мы храним в памяти только один экземпляр каждой строки. На рис. 11.5 это показано.

Конечно, используя такую систему, необходимо внимательно следить за удалением объектов класса String. Если деструктор этого класса использует `delete` для освобождения памяти, занятой символьной строкой, и если имеются несколько указателей на эту строку, объекты так и останутся с висящими указателями, указывающими туда, где строки уже давно нет.

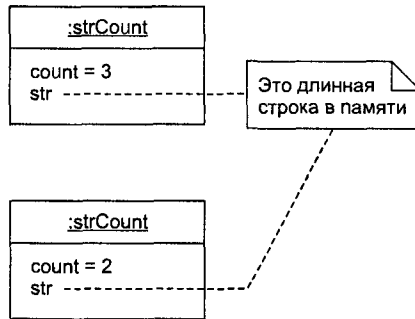


Рис. 11.5. Объектная диаграмма UML: дублирование указателей на строку

Выходит, что для того, чтобы использовать указатели на строки в объектах класса `String`, необходимо следить за тем, сколько именно объектов указывают на конкретную строку. Тем самым мы можем избежать использования `delete` по отношению к строкам до тех пор, пока не будет удален последний объект, указывающий на данную строку. Наш следующий пример, `STRIMEM`, предназначен как раз для демонстрации этого.

Класс-счетчик строк

Допустим, имеется несколько объектов класса `String`, указывающих на определенную строку, и мы хотим научить программу считать, сколько именно объектов на нее указывают. Где нам хранить этот счетчик?

Было бы слишком обременительно для каждого объекта класса `String` считать, сколько его коллег указывают на данную строку, поэтому мы не будем вводить счетчик в состав переменных класса `String`. Может быть, использовать статическую переменную? А ведь это мысль! Мы могли бы создать статический массив, который хранил бы список адресов строк и их порядковые номера. Да, но, с другой стороны, накладные расходы слишком велики. Рациональней будет создать новый особый класс для подсчета строк и указателей. Каждый объект такого класса, назовем его `strCount`, будет содержать счетчик и сам указатель на строчку. В каждый объект класса `String` поместим указатель на соответствующий объект класса `strCount`. Схема такой системы показана на рис. 11.6.

Для того чтобы убедиться в нормальном доступе объектов `String` к объектам `strCount`, сделаем `String` дружественным по отношению к `strCount`. Кроме того, нам хотелось бы достоверно знать, что класс `strCount` используется только классом `String`. Чтобы запретить несанкционированный доступ к каким-либо его функциям, сделаем все методы `strCount` скрытыми. Поскольку `String` является дружественным, на него это ограничение не распространяется. Приводим листинг программы `STRIMEM`.

Листинг 11.18. Программа `STRIMEM`

```

// strmem.cpp
// Класс String с экономией памяти
// Перегружаемая операция присваивания и конструктор          // копирования
#include <iostream>
#include <cstring>          // для strcpy() и т. д.
using namespace std;

```

```

////////////////////////////////////
class strCount                                // Класс-счетчик уникальных строк
{
    private:
        int count;                          // собственно счетчик
        char* str;                          // указатель на строку
        friend class String;                // сделаем себя доступными
        //методы скрыты
//-----
    strCount(char* s)                        // конструктор с одним аргументом
    {
        int length = strlen(s);             // длина строкового
                                              // аргумента
        str = new char[length+1];           // занять память
                                              // под строку
        strcpy(str, s);                     // копировать в нее аргументы
        count=1;                            // считать с единицы
    }
//-----
    ~strCount()                             // деструктор
    { delete[] str; }                       // удалить строку
};
////////////////////////////////////
class String                                // класс String
{
    private:
        strCount* psc;                      // указатель на strCount
    public:
        String()                            // конструктор без аргументов
        { psc = new strCount("NULL"); }
//-----
        String(char* s)                     // конструктор с одним аргументом
        { psc = new strCount(s); }
//-----
        String(String& S)                   // конструктор копирования
        {
            psc = S.psc;
            (psc->count)++;
        }
//-----
        ~String()                           // деструктор
        {
            if(psc->count==1)                 // если последний
                                              // пользователь,
            delete psc;                       // удалить strCount
            else                             // иначе
            (psc->count)--;                   // уменьшить счетчик
        }
//-----
        void display()                      // вывод String
        {
            cout << psc->str;                 // вывести строку
            cout << " (addr=" << psc << ")": // вывести адрес
        }
//-----
        void operator = (String& S)         // присвоение String
        {
            if(psc->count==1)                 // если последний
                                              // пользователь,

```

Листинг 11.18 (продолжение)

```
delete psc;           // удалить strCount
else                  // иначе
    (psc->count)--;    // уменьшить счетчик
psc = S.psc;          //использовать strCount
                      //аргумента
(psc->count)++;        //увеличить счетчик
}
}:
////////////////////////
int main()
{
    String s3 = "Муха по полю пошла, муха денежку нашла";
    cout << "\ns3="; s3.display(); //вывести s3

    String s1;          //определить объект String
    s1 = s3;            //присвоить его другому объекту
    cout << "\ns1="; s1.display(); //вывести его

    String s2(s3);      //инициализация
    cout << "\ns2="; s2.display(); //вывести
                          //инициализированное

    cout << endl;
    return 0;
}
```

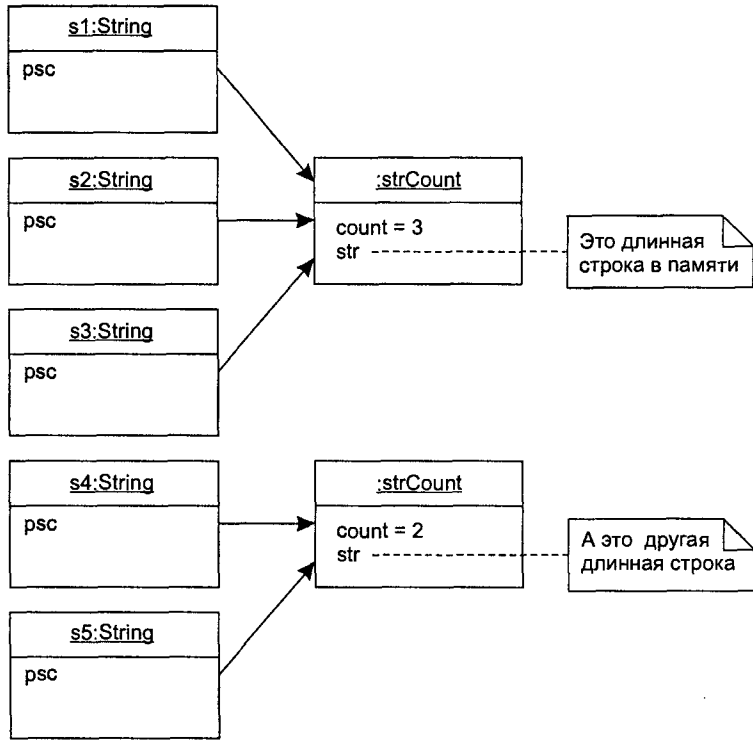


Рис. 11.6. Объекты классов String и strCount

В части `main()` данной программы мы определяем объект класса `String`, `s3`, содержащий строку из известного детского стишка: «Муха по полю пошла, муха денежку нашла». Затем определяем еще один объект, `s1`, и присваиваем ему значение `s3`. Определяем `s2` и инициализируем его с помощью `s3`. Полагание `s1` равным `s3` запускает перегружаемую операцию присваивания, а инициализация `s2` с помощью `s3` запускает перегружаемую операцию копирования. Выводим все три строки, а также адрес объекта класса `strCount`, на который ссылается указатель каждого объекта. Мы это делаем для того, чтобы показать, что все строки в действительности представлены одной и той же строкой. Ниже — результат работы программы `STRIMEM`:

```
s3= Муха по полю пошла. муха денежку нашла (addr=0x8f5410e00)
s2= Муха по полю пошла. муха денежку нашла (addr=0x8f5410e00)
s1= Муха по полю пошла. муха денежку нашла (addr=0x8f5410e00)
```

Остальные обязанности класса `String` мы поделили между классами `String` и `strCount`. Посмотрим, что каждый из них делает.

Класс `strCount`

Этот класс содержит указатель на реальную строку и считает, сколько объектов класса `String` на нее указывают. Его единственный конструктор рассматривает указатель на строку в качестве аргумента и выделяет для нее область памяти. Он копирует строку в эту область и устанавливает счетчик в единицу, так как только один объект `String` указывает на строку сразу после ее создания. Деструктор класса `strCount` освобождает память, занятую строкой. (Мы используем `delete[]` с квадратными скобками, так как строка — это массив.)

Класс `String`

У класса `String` есть три конструктора. При создании новой строки генерируется новый объект `strCount` для ее хранения, а указатель `psc` хранит ссылку на этот объект. Если копируется уже существующий объект `String`, указатель `psc` продолжает ссылаться на старый объект `strCount`, а счетчик в этом объекте увеличивается.

Перегружаемая операция присваивания должна наравне с деструктором удалять старый объект `strCount`, на который указывает `psc`, если счетчик равен единице. (Здесь нам квадратные скобки после `delete` уже не требуются, так как мы удаляем всего лишь единственный объект `strCount`.) Но почему вдруг оператор присваивания должен заботиться об удалении? Все дело в том, что объект `String`, стоящий в выражении слева от знака равенства (назовем его `s1`), указывал на некоторый объект `strCount` (назовем его `oldStrCnt`). После присваивания `s1` будет указывать на объект, находящийся справа от знака равенства. А если больше не существует объектов `String`, указывающих на `oldStrCnt`, он должен быть удален. Если же еще остались объекты, ссылающиеся на него, его счетчик просто должен быть уменьшен. Рисунок 11.7 показывает действия перегружаемой операции присваивания, а рис. 11.8 показывает конструктор копирования.

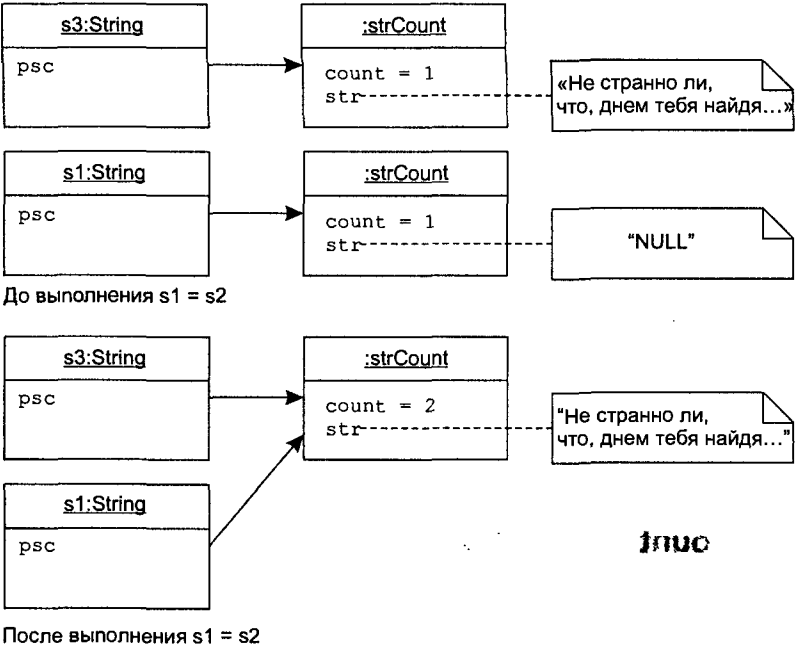


Рис. 11.7. Оператор присваивания в STRIMEM

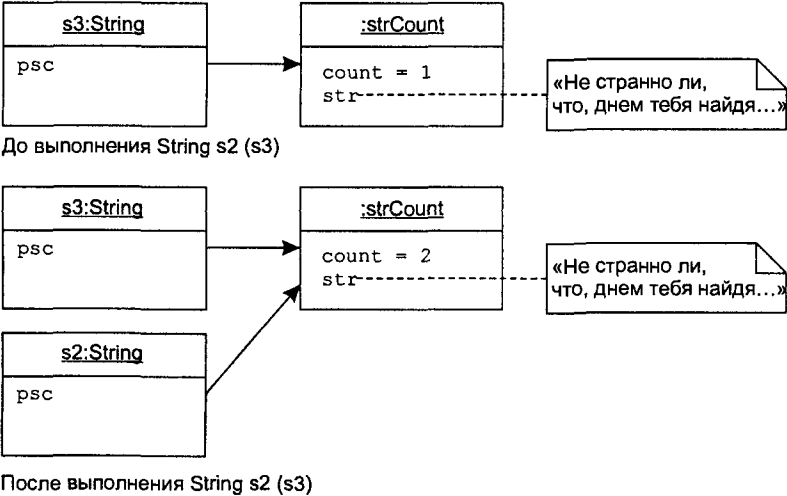


Рис. 11.8. Конструктор копирования в STRIMEM

Указатель this

Методы каждого объекта имеют доступ к некому волшебному указателю под названием `this`, который ссылается на сам объект. Таким образом, любой метод может узнать адрес, образно говоря, дома, в котором он прописан, то есть адрес сво-

его родного объекта. В следующем примере мы покажем механизм работы с этим загадочным указателем.

Листинг 11.19. Программа WHERE

```
// where.cpp
// указатель this
#include <iostream>
using namespace std;
////////////////////////////////////////
class where
{
private:
    char chararray[10]; //массив из 10 байтов
public:
    void reveal()
    { cout << "\nМой адрес - не дом и не улица. мой адрес - " << this; }
    // вывести адрес объекта
};
////////////////////////////////////////
int main()
{
    where w1, w2, w3; //создать три объекта
    w1.reveal();      //посмотреть, где они находятся
    w2.reveal();
    w3.reveal();
    cout << endl;
    return 0;
}
```

В функции `main()` данной программы создаются три объекта типа `where`. Затем печатаются их адреса с помощью метода `reveal()`. Этот метод просто выводит значение указателя `this`. Вот как это выглядит на экране:

```
Мой адрес - не дом и не улица. мой адрес - 0x8f4effec
Мой адрес - не дом и не улица. мой адрес - 0x8f4effe2
Мой адрес - не дом и не улица. мой адрес - 0x8f4effd8
```

Так как данные объектов хранятся в массивах, размер каждого из которых равен 10 байтов, то объекты в памяти отделены друг от друга десятью байтами ($EC-E2=E2-D8=10$ (дес)). Впрочем, некоторые компиляторы могут творчески подойти к расположению объектов и выделить под каждый чуть больше, чем 10 байтов.

Доступ к компонентным данным через указатель this

Когда вы вызываете какой-либо метод, значением указателя `this` становится адрес объекта, для которого этот метод вызван. Указатель `this` может интерпретироваться, как любой другой указатель на объект, соответственно, его можно использовать для получения доступа к данным объекта, на который он ссылается. Это продемонстрировано в программе `DOTTHIS`.

Листинг 11.20. Программа DOTTHIS

```
// dothis.cpp
// как указатель this ссылается на данные
#include <iostream>
using namespace std;
////////////////////////////////////
class what
{
private:
    int alpha;
public:
    void tester()
    {
        this->alpha = 11;    //то же. что alpha = 11;
        cout << this->alpha; //то же. что cout <<
                           //alpha;
    }
};
////////////////////////////////////
int main()
{
    what w;
    w.tester();
    cout << endl;
    return 0;
}
```

Эта программа просто выводит на экран значение 11. Обратите внимание, каким образом метод `tester()` получает доступ к переменной `alpha`:

```
this -> alpha
```

Это точно то же самое, что прямая ссылка на `alpha`. Такой синтаксис обрабатывается без каких-либо проблем, но в нем на самом деле никакого иного смысла нет, кроме единственного — показать, что указатель `this` действительно ссылается на объект.

Использование `this` для возврата значений

Более практичным применением указателя `this` является возврат значений из методов и перегружаемых операций в вызывающую программу.

Вспомните, что в программе `ASSIGN` мы никак не могли решить проблему возврата объекта из функции по ссылке, потому что объект был локальный по отношению к возвращающей функции и, следовательно, уничтожался во время выхода из функции. Нам необходим был более долговечный объект, чтобы возвращать результат по ссылке. Объект, чьим методом является данная функция, прочнее, нежели его собственные методы. Методы объекта создаются и уничтожаются при каждом вызове, в то время как сам объект может быть удален только извне (например, по `delete`). Таким образом, возврат по ссылке результата работы функции, являющейся методом данного объекта, является более профессиональным решением, чем возврат объекта, созданного данным методом. Объясним чуть более простыми словами: будет лучше, если в объект включен

метод, потому что в таком случае этот объект и возвращается в результате работы своего метода в вызывающую программу. Раньше у нас было наоборот — в методе содержался объект, который возвращался в вызывающую программу, но в результате того, что локальный объект при выходе из функции подлежит удалению, мы не могли передать его по ссылке. Новый подход очень легко осуществляется с помощью указателя `this`.

Ниже приведен листинг программы `ASSIGN2`, где функция `operator=()` возвращает по ссылке тот объект, в который она включена.

Листинг 11.21. Программа `ASSIGN2`

```
assign2.cpp
    возврат содержимого указателя this
#include <iostream>
using namespace std;
////////////////////////////////////
class alpha
{
private:
    int data;
public:
    alpha()                // конструктор без аргументов
    { }
    alpha(int d)           // конструктор с одним аргументом
    { data = d; }
    void display()         // вывести данные
    { cout << data; }
    alpha& operator = (alpha& a) // перегружаемая операция =
    {
        data = a.data;      // не делается автоматически
        cout << "\nЗапущен оператор присваивания";
        return *this;       // вернуть копию this alpha
    }
};
////////////////////////////////////
int main()
{
    alpha a1(37);
    alpha a2, a3;

    a3 = a2 = a1;           // перегружаемый =, дважды
    cout << "\na2="; a2.display(); // вывести a2
    cout << "\na3="; a3.display(); // вывести a3
    cout << endl;
    return 0;
}
```

В этой программе можно использовать описатель

```
alpha& operator = (alpha& a)
```

использующий возврат по ссылке, вместо

```
alpha operator = (alpha& a)
```

возвращающего результат по значению. Последним выражением функции является

```
return *this;
```

Так как `this` является указателем на объект, чей метод выполняется, то `*this` — это и есть сам объект, и вышеприведенное выражение возвращает его по ссылке. Вот результат работы программы `ASSIGN2`:

```
Запущен оператор присваивания
Запущен оператор присваивания
a2=37
a2=37
```

Всякий раз, когда в нашей программе встречается знак равенства

```
a3 = a2 = a1
```

запускается перегружаемая функция `operator=()`, которая выводит соответствующее сообщение. Все три объекта у нас имеют одно и то же значение.

Обычно описанный выше метод с возвратом результата работы перегружаемого `operator=()` по ссылке используется для того, чтобы избежать создания никому не нужных дополнительных копий объекта.

Исправленная программа `STRIMEM`

Используя указатель `this`, можно переделать функцию `operator=()` в программе `STRIMEM` и заставить ее возвращать результат по ссылке. Таким способом можно осуществить множественные присваивания для класса `String`, например

```
s1 = s2 = s3
```

В то же время мы избежим и создания ложных объектов в виде копий для возврата результата, как было, когда этот возврат осуществлялся по значению. Ниже представлен листинг программы `STRIMEM2`.

Листинг 11.22. Программа `STRIMEM2`

```
// strimem2.cpp
// Класс String с экономией памяти
// Перегружаемая операция присваивания и указатель this
#include <iostream>
#include <cstring> //для strcpy() и т. д.
using namespace std;
////////////////////////////////////
class strCount //Класс-счетчик уникальных строк
{private:
    int count;//собственно счетчик
    char* str;//указатель на строку
    friend class String; //сделаем себя доступными
    //методы скрытые
    strCount(char* s)//конструктор с одним аргументом
    {
        int length = strlen(s); //длина строкового //аргумента
        str = new char[length+1]; //занять память //под строку
        strcpy(str, s); //копировать в нее аргументы
        count=1; //считать с единицы
    }
    //-----
    ~strCount()//деструктор
    { delete[] str; } //удалить строку
};
```

```

////////////////////////////////////
class String //класс String
{
private:
strCount* psc: //указатель на strCount
public:
String() //конструктор без аргументов
{ psc = new strCount("NULL"); }
//-----
String(char* s) //конструктор с одним аргументом
{ psc = new strCount(s); }
//-----
String(String& S) //конструктор копирования
{
cout << "\nКОНСТРУКТОР КОПИРОВАНИЯ";
psc = S.psc;
(psc->count)++;
}
//-----
~String() //деструктор
{
if(psc->count==1) //если последний //пользователь.
delete psc; //удалить strCount
else//иначе
(psc->count)--://уменьшить счетчик
}
//-----
void display() //вывод String
{
cout << psc->str; //вывести строку
cout << " (addr=" << psc << ")"; //вывести //адрес
}
//-----
String& operator = (String& S)//присвоение String
{
cout << "\nПРИСВАИВАНИЕ";
if(psc->count==1) //если последний //пользователь.
delete psc; //удалить strCount
else//иначе
(psc->count)--://уменьшить счетчик
psc = S.psc; //использовать strCount //аргумента
(psc->count)++; //увеличить счетчик
return *this; //вернуть этот объект
}
};
////////////////////////////////////
int main()
{
String s3 = "Муха по полю пошла, муха денежку нашла";
cout << "\ns3=": s3.display();//вывести s3

String s1,s2: //определить объекты String
s1 = s2 = s3;//присваивания
//вывести их
cout << "\ns1=": s1.display();
cout << "\ns2=": s2.display();
cout << endl;//ожидать нажатия клавиши
return 0;
}

```

Теперь описателем оператора присваивания является

```
String& operator = (String& S)//возврат по ссылке
```

и, как в программе ASSIGN2, функция возвращает указатель на `this`. Результат работы программы:

```
s3=Муха по полю пошла, муха денежку нашла (addr=0x8f640d3a)
ПРИСВАИВАНИЕ
ПРИСВАИВАНИЕ
s2=Муха по полю пошла, муха денежку нашла (addr=0x8f640d3a)
s1=Муха по полю пошла, муха денежку нашла (addr=0x8f640d3a)
```

Результат работы программы показывает, что в соответствии с выражением присваивания все три объекта класса `String` указывают на один и тот же объект класса `strCount`.

Следует отметить, что указатель `this` нельзя использовать в статических методах, так как они не ассоциированы с конкретным объектом.

Остерегайтесь неправильных присваиваний

Один из основополагающих так называемых «законов Мёрфи» говорит о том, что все, что можно случайно сделать не так, обязательно будет кем-нибудь сделано. Это абсолютная правда в отношении программирования, поэтому будьте уверены: если вы переопределили оператор присваивания, кто-нибудь станет его использовать для присваивания объектов самим себе. Вот так:

```
alpha = alpha;
```

Перегружаемый = должен быть морально готов обработать такую ситуацию. В противном случае не ждите ничего хорошего. Например, в секции `main()` программы STRIMEM2, если вы положите объект `String` равным самому себе, программа зависнет (несмотря на то что другие объекты без проблем используют один и тот же объект `strCount`). Проблема заключается в том, что согласно коду перегружаемой операции присваивания последний удаляет объект `strCount`, если считает, что вызываемый объект является единственным, использующим `strCount`. Присваивание объекта самому себе убедит оператор в этом прискорбном факте, хотя на самом деле никто и не собирался удалять никакие объекты.

Чтобы привести все в порядок, нужно ввести проверку на присваивание самому себе в начале работы каждого перегружаемого =. В большинстве случаев это делается путем сравнения адресов объектов, стоящих справа и слева от знака равенства. Понятно, что если адреса равны, значит, осуществляется присваивание объекта самому себе и необходимо немедленно вернуться из этой функции. (Если кто-то не понял, в чем проблема, поясняем, что не нужно присваивать объекты самим себе — они и так уже равны.) Например, в STRIMEM2 с этой целью можно вставить следующие две строчки

```
if(this == &S)
    return *this;
```

в начало функции `operator=()`. Это должно решить возникшую проблему.

Динамическая информация о типах

Существует возможность узнавать информацию о классе объекта и даже изменять этот класс прямо во время выполнения программы. Мы вкратце рассмотрим два механизма, которые служат для этого: операция `dynamic_cast` и оператор `typeid`. Их можно рассматривать как дополнительные средства языка, но когда-нибудь они могут и пригодиться.

Эти возможности используются, когда базовый класс имеет ряд порожденных классов, созданных порой довольно непростым путем. Для того чтобы использовать динамический подход, базовый класс обязан быть полиморфным, то есть он должен содержать по крайней мере одну виртуальную функцию.

Чтобы заработали `dynamic_cast` и `typeid`, компилятор должен активизировать механизм, который позволяет определять и изменять тип объекта во время выполнения программы — RTTI (Run-Time Type Information). В системе Borland C++ Builder этот механизм включается по умолчанию, а в Microsoft Visual C++ нужно подключать его вручную. Также необходимо включить в программу заголовочный файл `TYPEINFO`.

Проверка типа класса с помощью `dynamic_cast`

Допустим, некая внешняя программа передает в вашу программу объект (так любит делать операционная система с функцией обратного вызова). Предполагается, что объект принадлежит какому-то известному классу, но вам хочется проверить это. Как это сделать? Как определить класс объекта? Оператор `dynamic_cast` позволяет это сделать, предполагая, что классы, чьи объекты вы хотите проверить, все вышли из гоголевской «Шинели», то есть имеют общий базовый класс. Этот подход демонстрируется в программе `DYNCAST1`:

Листинг 11.23. Программа `DYNCAST1`

```
//dyncast1.cpp
//динамическое приведение типов для проверки типа объекта
//RTTI должен быть включен
#include <iostream>
#include <typeinfo>           //для dynamic_cast
using namespace std;
////////////////////////////////////
class Base
{
    virtual void vertFunc()    //для dynamic cast
    { }
};
class Derv1 : public Base
{ };
class Derv2 : public Base
{ };
```

Листинг 11.23 (продолжение)

```

////////////////////////////////////
//проверка, указывает ли pUnknown на Derv1
bool isDerv1(Base* pUnknown) //неизвестный подкласс базового
{
    Derv1* pDerv1;
    if( pDerv1 = dynamic_cast<Derv1*>(pUnknown) )
        return true;
    else
        return false;
}
//-----
int main()
{
    Derv1* d1 = new Derv1;
    Derv2* d2 = new Derv2;

    if( isDerv1(d1) )
        cout << "d1 - компонент класса Derv1\n";
    else
        cout << "d1 - не компонент класса Derv1\n";

    if( isDerv1(d2) )
        cout << "d2 - компонент класса Derv1\n";
    else
        cout << "d2 - не компонент класса Derv1\n";
    return 0;
}

```

У нас есть базовый класс Base и два порожденных — Derv1 и Derv2. Есть также функция isDerv1(), возвращающая значение «истина», если указатель, переданный в качестве аргумента, указывает на объект класса Derv1. Наш аргумент принадлежит классу Base, поэтому объекты могут быть либо класса Derv1, либо Derv2. Оператор dynamic_cast пытается привести указатель неизвестного типа pUnknown к типу Derv1. Если результат ненулевой, значит, pUnknown указывал на объект Derv1. В противном случае он указывал на что-то другое.

Изменение типов указателей с помощью dynamic_cast

Оператор dynamic_cast позволяет осуществлять приведение типов «вверх» и «вниз» по генеалогическому дереву классов. Но его возможности имеют некоторые ограничения. Покажем в программе DYNCAST2 порядок работы с этим оператором.

Листинг 11.24. Программа DYNCAST2

```

//dyncast2.cpp
//Тестирование динамического приведения типов
//RTTI должен быть включен
#include <iostream>
#include <typeinfo>          //для dynamic_cast
using namespace std;
////////////////////////////////////
class Base

```

```

{
protected:
    int ba;
public:
    Base() : ba(0)
    { }
    Base(int b) : ba(b)
    { }
    virtual void vertFunc() //для нужд dynamic_cast
    { }
    void show()
    { cout << "Base: ba=" << ba << endl; }
};
////////////////////////////////////
class Derv : public Base
{
private:
    int da;
public:
    Derv(int b, int d) : da(d)
    { ba = b; }
    void show()
    { cout << "Derv: ba=" << ba << ", da=" << da << endl; }
};
////////////////////////////////////
int main()
{
    Base* pBase = new Base(10); //указатель на Base
    Derv* pDerv = new Derv(21, 22); //указатель на Derv

    //приведение к базовому типу: восхождение по дереву -
    //указатель поставлен на подобъект Base класса Derv
    pBase = dynamic_cast<Base*>(pDerv);
    pBase->show(); // "Base: ba=21"

    pBase = new Derv(31, 32); //обычное нисходящее
    //приведение типов -- pBase должен указывать на Derv)
    pDerv = dynamic_cast<Derv*>(pBase);
    pDerv->show(); // "Derv: ba=31, da=32"
    return 0;
}

```

В этом примере у нас есть базовый и порожденный классы. В каждый из них мы ввели элемент данных для лучшей демонстрации эффекта динамического приведения типов.

При восходящем приведении типов производится попытка изменить объект порожденного класса на объект базового класса. Все, что вы при этом можете получить, — это только унаследованная из базового класса часть объекта порожденного. В нашем примере мы создали объект класса `Derv`. Ему в наследство от базового класса досталась переменная `ba`, которая, будучи в базовом классе, имела значение 21, а в порожденном классе имела значение 22. После приведения типов указатель `pBase` стал ссылаться на базовую часть объекта класса `Derv1`, поэтому на экране мы видим: `Base:ba=21`. Восходящее приведение типов целесообразно использовать, когда вам нужно получить базовую составляющую объекта.

При нисходящем приведении типов мы переносим ссылку на объект порожденного класса из указателя базового класса в указатель порожденного.

Оператор typeid

Иногда требуется более полная информация об объекте, нежели простая верификация его классовой принадлежности. Можно узнать, например, имя типа неопознанного объекта, используя оператор typeid. Рассмотрим в связи с этим программу TYPEID.

Листинг 11.25. Программа TYPEID

```
// typeid.cpp
// демонстрация функции typeid()
// RTTI должен быть активизирован
#include <iostream>
#include <typeinfo>          // для typeid()
using namespace std;
////////////////////////////////////
class Base
{
    virtual void virtFunc() // для нужд typeid
    { }
};
class Derv1 : public Base
{ };
class Derv2 : public Base
{ };
////////////////////////////////////
void displayName(Base* pB)
{
    cout << "указатель на объект класса ": // вывести имя класса
    cout << typeid(*pB).name() << endl;   //на который
                                           //указывает pB
}
//-----
int main()
{
    Base* pBase = new Derv1;
    displayName(pBase); //указатель на объект класса Derv1"
    pBase = new Derv2;
    displayName(pBase); //" указатель на объект класса Derv2"
    return 0;
}
```

В этом примере функция displayName() отображает имя класса переданного ей объекта. Для этого она использует переменную name класса type_info наряду с оператором typeid. В main() мы передаем этой функции два объекта классов Derv1 и Derv2 соответственно. На экране видим следующее:

```
указатель на объект класса Derv1
указатель на объект класса Derv2
```

Используя typeid, кроме имени можно получить и некоторую другую информацию о классе. Например, используя перегружаемую операцию ==, можно

осуществить проверку на равенство классов. Пример этого мы покажем в программе EMPL_IO в главе 12 «Потоки и файлы». Несмотря на то что в этом разделе в примерах использовались указатели, `dynamic_cast` и `typeid` работают одинаково хорошо со ссылками.

Резюме

Виртуальные функции позволяют решать прямо в процессе выполнения программы, какую именно функцию вызывать. Без их использования это решение принимается на этапе компиляции программы. Виртуальные функции дают большую гибкость при выполнении одинаковых действий над разнородными объектами. В частности, они разрешают использование функций, вызванных из массива указателей на базовый класс, который на самом деле содержит указатели (или ссылки) на множество порожденных классов. Это пример *полиморфизма*. Обычно функция объявляется виртуальной в базовом классе, а другие функции с тем же именем объявляются в порожденных классах.

Использование по крайней мере одной чистой виртуальной функции делает весь класс, содержащий ее, *абстрактным*. Это означает, что с помощью этого класса нельзя создавать никакие объекты.

Дружественная функция имеет доступ к скрытым данным класса, по отношению к которому она объявлена таковой. Это бывает полезно, когда одна функция должна иметь доступ к двум и более не связанным между собой классам и когда перегружаемая операция должна, со своей стороны, использовать данные «чужого» класса, методом которого она не является. Дружественность также используется для упрощения записи функций.

Статической называется функция, работающая с классом в целом, а не над отдельными его объектами. В частности, она может обрабатывать статические переменные. Она может быть вызвана с использованием имени класса и оператора явного задания функции.

Оператор присваивания = может быть перегружен. Это необходимо, когда он должен выполнять более сложную работу, нежели простое копирование содержимого одного объекта в другой. Конструктор копирования, создающий копии объектов во время инициализации и во время передачи аргументов и возврата результата по значению, также может быть перегружен. Это необходимо, когда он должен выполнять более сложную работу, чем простое копирование объекта.

Указатель `this` может использоваться в функции для указания на объект, чьим методом он является. Он пригодится, если в качестве результата возвращается объект, чьим методом является данная функция.

Оператор `dynamic_cast` умеет многое. С его помощью можно определить, на объект какого типа ссылается указатель, можно в некоторых ситуациях изменять тип указателя. Оператор `typeid` позволяет узнать кое-что о классе объекта, например его название.

Объектная диаграмма UML показывает отношения внутри группы объектов в определенный момент работы программы.

Вопросы

Ответы на эти вопросы можно найти в приложении Ж.

1. Виртуальные функции позволяют:
 - а) создавать массивы типа «указатель на базовый класс», которые могут содержать указатели на производные классы;
 - б) создавать функции, к которым никогда не будет доступа;
 - в) группировать объекты разных классов так, чтобы они все были доступны с помощью одного и того же вызова функции;
 - г) использовать один и тот же вызов функции для выполнения методов объектов, принадлежащих разным классам.
2. Истинно ли утверждение о том, что указатель на базовый класс может ссылаться на объекты порожденного класса?
3. Пусть указатель `p` ссылается на объекты базового класса и содержит адрес объекта порожденного класса. Пусть в обоих этих классах имеется невиртуальный метод `ding()`. Тогда выражение `p->ding();` поставит на выполнение версию функции `ding()` из _____ класса.
4. Напишите описатель для виртуальной функции `dang()`, возвращающей результат типа `void` и имеющей аргумент типа `int`.
5. Принятие решения о том, какая именно функция будет выполняться по конкретному вызову функции, называется _____.
6. Пусть указатель `p` ссылается на объекты базового класса и содержит адрес объекта порожденного класса. Пусть в обоих этих классах имеется виртуальный метод `ding()`. Тогда выражение `p->ding();` поставит на выполнение версию функции `ding()` из _____ класса.
7. Напишите описатель чистой виртуальной функции `aragorn`, не возвращающей значений и не имеющей аргументов.
8. Чистая виртуальная функция — это виртуальная функция, которая:
 - а) делает свой класс абстрактным;
 - б) не возвращает результата;
 - в) используется в базовом классе;
 - г) не имеет аргументов.
9. Напишите определение массива `par1`, содержащего 10 указателей на объекты класса `dong`.
10. Абстрактный класс используется, когда:
 - а) не планируется создавать порожденные классы;
 - б) есть несколько связей между двумя порожденными классами;
 - в) с его помощью запрещено создавать какие-либо объекты;
 - г) вы хотите отложить объявление класса.

11. Истинно ли утверждение о том, что дружественная функция имеет доступ к скрытым данным класса, даже не являясь его методом?
12. Дружественная функция может быть использована для того, чтобы:
 - а) разные классы могли пользоваться одними аргументами;
 - б) разрешать доступ к классам, исходный код которых недоступен;
 - в) разрешать доступ к несвязанному классу;
 - г) увеличить многосторонность применения перегружаемой операции.
13. Напишите описатель дружественной функции `harry()`, возвращающей результат типа `void` и имеющей один аргумент класса `george`.
14. Ключевое слово `friend` появляется в:
 - а) классе, разрешающем доступ к другому классу;
 - б) классе, требующем доступа к другому классу;
 - в) разделе скрытых компонентов класса;
 - г) разделе общедоступных компонентов класса.
15. Напишите описатель, который в том классе, где он появится, сделает каждый компонент класса `harry` дружественной функцией.
16. Статическая функция:
 - а) должна вызываться, когда объект уничтожается;
 - б) сильно связана с индивидуальным объектом класса;
 - в) может быть вызвана с использованием имени класса и имени функции;
 - г) используется для создания простого объекта.
17. Объясните, что делает по умолчанию оператор присваивания `=` в применении к объектам.
18. Напишите описатель перегружаемой операции присваивания для класса `zeta`.
19. Оператор присваивания может быть перегружен с целью:
 - а) хранения информации о количестве одинаковых объектов;
 - б) присваивания идентификационного номера каждому объекту;
 - в) проверки того, что все компонентные данные скопировались без ошибок;
 - г) уведомления о том, что имело место присваивание.
20. Истинно ли утверждение о том, что пользователь всегда должен определять операцию для конструктора копирования?
21. Операции, выполняемые оператором присваивания и конструктором копирования:
 - а) похожи, за исключением того, что конструктор копирования создает новый объект;
 - б) похожи, за исключением того, что оператор присваивания копирует компонентные данные;

- в) различны, за исключением того, что оба создают новый объект;
 - г) различны, за исключением того, что оба копируют компонентные данные.
22. Напишите описатель конструктора копирования для класса Bertha.
23. Истинно ли утверждение о том, что конструктор копирования может быть переопределен с целью копирования только части данных объекта?
24. Продолжительность жизни переменной, которая является:
- а) локальной по отношению к методу, совпадает с продолжительностью жизни функции;
 - б) глобальной, совпадает с продолжительностью жизни класса;
 - в) нестатическим компонентным данным объекта, совпадает с продолжительностью жизни объекта;
 - г) статической внутри метода, совпадает с продолжительностью жизни этой функции.
25. Истинно ли утверждение о том, что возврат по значению локальной переменной метода не создает никаких проблем?
26. Объясните разницу в выполнении следующих двух выражений:
- ```
person p1(p0):
person p1 = p0;
```
27. Конструктор копирования запускается, когда:
- а) функция возвращается по значению;
  - б) аргумент передается по значению;
  - в) функция возвращается по ссылке;
  - г) аргумент передается по ссылке.
28. На что ссылается указатель this?
29. Если в заданном классе переменная da является компонентными данными, присвоит ли выражение this.da=37; значение 37 переменной da?
30. Напишите выражение, с помощью которого функция может возвращать весь объект, методом которого она является, без создания временных объектов.
31. Прямоугольник в объектной диаграмме означает:
- а) общую группу объектов;
  - б) класс;
  - в) экземпляр класса;
  - г) все объекты класса.
32. Линии между объектами в объектной диаграмме UML называются \_\_\_\_\_.
33. Истинно ли утверждение о том, что объект А может быть связанным с объектом В только в данный момент и более ни в какой иной?

34. Объектные диаграммы показывают, какие объекты:
- а) существуют в данный момент;
  - б) взаимодействуют в данный момент;
  - в) принимают участие в каких-то действиях в данный момент;
  - г) имеют операции (методы), вызывающие объекты других классов.

## Упражнения

Решения к упражнениям, помеченным знаком \*, можно найти в приложении Ж.

- \*1. Пусть имеется та же издательская компания, которая описана в упражнении 1 главы 9, которая продает и книги, и аудио версии печатной продукции. Как и в том упражнении, создайте класс `publication`, хранящий название (фактически, строку) и цену (типа `float`) публикации. Создайте два порожденных класса: `book`, в котором происходит изменение счетчика страниц (типа `int`), и `tape`, в котором происходит изменение счетчика записанных на кассету минут. Каждый из классов должен иметь метод `getdata()`, запрашивающий информацию у пользователя, и `putdata()` для вывода данных на экран.
2. Напишите `main()`, где создавался бы массив указателей на класс `publication`. Это очень похоже на то, что мы делали в текущей главе на примере `VIRTPERS`. В цикле запрашивайте у пользователя данные о конкретной книге или кассете, используйте `new` для создания нового объекта `book` или `tape`. Сопоставляйте указатель в массиве с объектом. Когда пользователь закончит ввод исходных данных, выведите результат для всех введенных книг и кассет, используя цикл `for` и единственное выражение `pubarr[j]->putdata()`: для вывода данных о каждом объекте из массива
- \*3. В классе `Distance`, как показано в примерах `FRENGL` и `FRISQ` из этой главы, создайте перегружаемую операцию умножения `*`, чтобы можно было умножать два расстояния. Сделайте эту функцию дружественной, тогда можно будет использовать выражение типа `Wdist=7.5*dist2`. Вам понадобится конструктор с одним аргументом для перевода величин из формата чисел с плавающей запятой в формат `Distance`. Напишите какой-либо `main()` на свое усмотрение для того, чтобы несколькими способами проверить работу этой перегружаемой операции.
- \*4. Как уже говорилось, классы можно заставлять вести себя как массивы. Пример `CLARRAY` показывает один из способов создания такого класса.

### Листинг 11.26. Программа CLARRAY

```
// clarray.cpp
// создает класс-массив
#include <iostream>
using namespace std;
////////////////////////////////////
class Array //моделирует обычный массив C++
{
```

**Листинг 11.26 (продолжение)**

```

private:
 int* ptr; //указатель на содержимое Array
 int size; //размер Array
public:
 Array(int s) //конструктор с одним аргументом
 {
 size = s; //аргумент - размер Array
 ptr = new int[s]; //выделить память под Array
 }
 ~Array() //деструктор
 { delete[] ptr; }
 int& operator [] (int j) //перегружаемая операция
 //списка индексов
 { return *(ptr+j); }
};
////////////////////////////////////
int main()
{
 const int ASIZE = 10; //размер массива
 Array arr(ASIZE); //создать массив

 for(int j=0; j<ASIZE; j++) //заполнить его j^2
 arr[j] = j*j;

 for(j=0; j<ASIZE; j++) //вывести его содержимое
 cout << arr[j] << ' ';
 cout << endl;
 return 0;
}

```

Результат работы программы:

```
0 1 4 9 16 25 36 49 64 81
```

Взяв за основу приведенную программу, добавьте перегружаемое присваивание и перегружаемый конструктор копирования к классу `Array`. Затем добавьте к `main()` выражение `Array arr2(arr1);` и `arr3=arr1;` для проверки того, что перегружаемые операции работают. Конструктор копирования должен создать новый объект `Array` со своим собственным местом в памяти, выделенным для хранения элементов массива. И конструктор копирования, и оператор присваивания должны копировать содержимое старого объекта класса `Array` в новый. Что будет, если вы присвоите объект `Array` одного размера объекту `Array` другого размера?

- Взяв за основу программу из упражнения 1 этой главы, добавьте метод типа `bool`, называющийся `isOveersize()`, к классам `book` и `tape`. Допустим, книга, в которой больше 800 страниц, или кассета со временем проигрывания более 90 минут, будут считаться объектами с превышением размера. К этой функции можно обращаться из `main()`, а результат ее работы выводить в виде строки «Превышение размера!» для соответствующих книг и кассет. Допустим, объекты классов `book` и `tape` должны быть доступны через указатели на них, хранящиеся в массиве типа `publication`.

Что в этом случае вам нужно добавить в базовый класс `publication`? Вы можете привести примеры компонентов этого базового класса?

6. Возьмите за основу программу из упражнения 8 главы 8, где было перегружено пять арифметических операций для работы с денежным форматом. Добавьте два оператора, которые не были перегружены в том упражнении:

```
long double * bMoney //умножать число на деньги
long double / bMoney //делить число на деньги
```

Эти операции требуют наличия дружественных функций, так как справа от оператора находится объект, а слева — обычное число. Убедитесь, что `main()` позволяет пользователю ввести две денежные строки и число с плавающей запятой, а затем корректно выполняет все семь арифметических действий с соответствующими парами значений.

7. Как и в предыдущем упражнении, возьмите за основу программу из упражнения 8 главы 8. На этот раз от вас требуется добавить функцию, округляющую значение `bMoney` до ближайшего доллара:

```
mo2 = round(mo1);
```

Как известно, значения, не превышающие \$0.49, округляются вниз, а числа от \$0.50 и более округляются вверх. Можно использовать библиотечную функцию `modfl()`. Она разбивает переменную типа `long double` на целую и дробную части. Если дробная часть меньше 0.50, функция просто возвращает целую часть числа. В противном случае возвращается увеличенная на 1 целая часть. В `main()` проверьте работоспособность функции путем передачи в нее последовательно значений, одни из которых меньше \$0.49, другие — больше \$0.50.

8. Помните программу `PARSE` из главы 10? Попробуйте доработать ее, чтобы она могла вычислять значения математических выражений с рациональными числами, например типа `float`, а не только с одноразрядными числами:

```
3.14159 / 2.0 + 75.25 * 3.333 + 6.02
```

Во-первых, нужно развить стек до такой степени, чтобы он мог хранить и операторы (типа `char`), и числа (типа `float`). Но как, спрашивается, можно хранить в стеке значения двух разных типов? Ведь стек — это, по сути дела, массив. Надо еще учесть, что типы `char` и `float` даже не совпадают по размеру! Даже указатели на разные типы данных (`char*` и `float*`) компилятор не позволит хранить в одном массиве, несмотря на то, что они одинакового размера. Единственный способ хранить в массиве два разных типа указателей — сделать эти типы наследниками одного и того же базового класса. При этом базовому классу даже нет нужды иметь какие-то собственные данные, это может быть абстрактный класс, из которого никакие объекты создаваться не будут.

Конструкторы могут хранить значения в порожденных классах обычным способом, но должна иметься специальная чистая виртуальная функция

для того, чтобы извлечь эти значения. Представляем возможный сценарий работы над этим вопросом:

```
class Token // Абстрактный базовый класс
{
public:
 virtual float getNumber()=0; // чистая виртуальная
 // функция
 virtual char getOperator()=0;
};
class Operator : public Token
{
private:
 char oper; // Операторы +, -, *, /
public:
 Operator(char): // конструктор устанавливает значение
 char getOperator(): // получить значение
 float getNumber(): // просто некая функция
};
class Number : public Token
{
private:
 float fnum; // число
public:
 Number(float): // конструктор устанавливает значение
 float getNumber(): // получить значение
 char getOperator(): // просто некая функция
};
Token* atoken[100]: //содержит типы Operator* и Number*
```

Виртуальные функции базового класса должны быть реализованы во всех порожденных классах, в противном случае классы становятся абстрактными. Таким образом, классу `Operand` нужна функция `getNumber()`, несмотря на то, что она фиктивная. Классу `Number` нужна функция `getOperand()`, несмотря на то, что она тоже фиктивная.

Поработайте над этим каркасом, сделайте его реально работающей программой, добавив класс `Stack`, содержащий объекты класса `Token`, и функцию `main()`, в которой бы заносились в стек и извлекались из него разные арифметические операторы и числа в формате с плавающей запятой.

9. Внесем некоторое разнообразие в пример `HORSE` из главы 10, создав класс для внесения в него лошадей экстракласса. Предположим, что любая лошадь, которая на скачках к середине дистанции находится впереди всех, становится практически непобедимой. Относительно класса лошадей создадим порожденный класс `comhorse` (для конкурентоспособной лошади). Перегрузим функцию `horse_tick()` в этом классе таким образом, чтобы каждая лошадь могла проверять, является ли она ведущей и нет ли поблизости соперников (скажем, ближе, чем на 0,1 форлонг (1/80 часть мили или 20,1 м.)). Если есть, то ей следует немного ускориться. Может быть, не настолько, чтобы побеждать на всех скачках, но в достаточной мере для того, чтобы оставаться конкурентоспособной.

Как каждая лошадь узнает, где находятся остальные? Моделирующий ее объект должен иметь доступ к области памяти, в которой хранятся дан-

ные о соперниках. В программе HORSE это hArray. Будьте внимательны: вы создадите класс для передовой лошади, он должен быть наследником класса всех лошадей. Поэтому классу comhorse потребуются перегрузить hArray. Вам может потребоваться создать еще один производный класс, comtrack, для отслеживания позиции лошади.

Можно непрерывно проверять, лидирует ли ваша лошадь, и если она впереди всех, но лишь незначительно, следует ее немного ускорить.

10. Упражнение 4 в главе 10 включало в себя добавление к классу linklist перегружаемого деструктора. Допустим, мы заполняем объект этого класса данными, а затем присваиваем один класс целиком другому, используя стандартный оператор присваивания:

```
list2 = list1;
```

Допустим, что впоследствии мы удалим объект класса list1. Можем ли мы все еще использовать list2 для доступа к введенным данным? Увы, нет, так как при удалении list1 все его ссылки были удалены. Единственное, что было известно объекту linklist про удаленный объект, это указатель на него. Но его удалили, указатель в list2 стал недействительным, и все попытки получить доступ к данным приведут к получению мусора вместо данных, а в худшем случае — к зависанию программы.

Один из способов избежать этих проблем — перегрузить оператор присваивания, чтобы он вместе с объектом копировал бы все его ссылки. Но тогда придется пройти по всей цепочке, поочередно копируя все ссылки. Как отмечалось ранее, следует также перегружать конструктор копирования. Чтобы была возможность удалять объекты linklist в main(), можно создавать их с помощью указателя и new. В таком случае проще будет проверять работу новых операций. Не переживайте, если обнаружите, что в процессе копирования порядок следования данных изменился.

Понятно, что копирование всех данных не является самым эффективным решением проблемы с точки зрения экономии памяти. Сравните этот подход с представленным в примере STRIMEM (глава 10), где использовался только один набор данных для всех объектов, и хранилась информация о том, сколько объектов указывали на эти данные.

11. Выполните изменения в соответствии с упражнением 7, применив их к программе PARSE главы 10. То есть заставьте программу анализировать выражения, содержащие числа в формате с плавающей запятой. Совместите классы, предложенные в упражнении 7, с алгоритмами из PARSE. Вам придется работать с указателями на символы вместо работы с самими символами. Это потребует выражений, подобных следующим:

```
Number* ptrN = new Number(ans);
s.push(ptrN);
and
Operator* ptrO = new Operator(ch);
s.push(ptrO);
```

Р. ЛАФОРЕ

# ОБЪЕКТНО-ОРИЕНТИРОВАННОЕ ПРОГРАММИРОВАНИЕ В C++

4-Е ИЗДАНИЕ

КНИГИ, КОТОРЫЕ НЕ СТАРЕЮТ!

КЛАССИКА COMPUTER SCIENCE

Изучение объектно-ориентированной парадигмы программирования обычно ассоциируется с многочасовым курсом лекций и утомительными лабораторными занятиями, во время которых вас бросает из крайности в крайность — от тупого отчаяния до ложного ощущения полного понимания. Книга популярного среди американских студентов писателя Роберта Лафоре призвана помочь изучающему C++ избежать таких нагрузок на психику и без чьей-либо помощи пройти путь от застенчивого ученика до уверенного в своих силах программиста.

Столь значительный объем этого учебника обусловлен как сложностью предмета, так и стремлением автора не оставить белых пятен в программистском образовании читателя: любой элемент теории немедленно иллюстрируется небольшим примером кода, для приобретения практических навыков даются детально прокомментированные тексты полномасштабных приложений, а для закрепления полученных знаний читателю предлагается выполнить множество самостоятельных упражнений.

ISBN 5-94723-302-9



9 785947 123302 5

Обсудите наш веб-сайт: [www.piter.com](http://www.piter.com)

 ПИТЕР  
WWW.PITER.COM

 SAMS