

Министерство образования и науки Российской Федерации

ТОМСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ
СИСТЕМ УПРАВЛЕНИЯ И РАДИОЭЛЕКТРОНИКИ (ТУСУР)

М. Ю. Катаев

ОБЪЕКТНО-ОРИЕНТИРОВАННОЕ ПРОГРАММИРОВАНИЕ

Учебное пособие

Томск
«Эль Контент»
2013

УДК 004.42.045(075.8)

ББК 32.973.2-018я73

К 290

Рецензенты:

Козин Е. С., ведущий программист Института химии нефти Сибирского
отделения Российской академии наук, г. Томск;

Кривенцов Д. М., ведущий инженер по тестированию,
компания «Unigine 3D», г. Томск.

Катаев М. Ю.

К 290 Объектно-ориентированное программирование : учебное пособие /
М. Ю. Катаев. — Томск : Эль Контент, 2013. — 212 с.

ISBN 978-5-4332-0100-2

Учебное пособие предназначено для ознакомления студентов с основами объектно-ориентированного программирования как современного подхода к программированию и подготовки их к решению различных экономических задач с использованием различных методов программирования. В результате изучения курса студенты должны получить начальные знания, необходимые для программирования на языке C++.

УДК 004.42.045(075.8)

ББК 32.973.2-018я73

ISBN 978-5-4332-0100-2

© Катаев М. Ю., 2013

© Оформление.

ООО «Эль Контент», 2013

ОГЛАВЛЕНИЕ

Введение	5
1 Возможности Си++	8
1.1 Локальные и глобальные переменные	8
1.2 Подпрограммы и их аргументы	9
1.3 Определение данных	10
1.4 Операторы динамического распределения памяти	14
1.5 Перегрузка функций и операций	15
2 Объекты и классы	22
2.1 Класс как обобщение структуры	23
2.2 Определение первичного класса	26
2.3 Перегрузка операций	29
2.4 Конструкторы	33
2.5 Список инициализации	35
2.6 Деструктор	37
2.7 Дружественные классы	40
2.8 Статические элементы класса	43
2.9 Шаблоны функций	45
3 Контейнерные классы	55
3.1 Шаблоны классов	55
3.2 Параметризованные очереди и стеки	59
3.3 Бинарные деревья	65
3.4 Определение класса множества	73
4 Производные классы	90
4.1 Определение производного класса	90
4.2 Доступ к полям и функциям базового класса	91
4.3 Класс дерева поиска	93
4.4 Параметризованный связный список	96
4.5 Множественное наследование	102
4.6 Виртуальные классы	105
5 Виртуальные функции	114
5.1 Переопределение составной функции	114
5.2 Организация списка объектов различного типа	116
5.3 Техническая реализация виртуальных функций	120

5.4	Виртуальные деструкторы	121
5.5	Абстрактные классы	122
6	Теоретические основы объектно-ориентированного программирования	135
6.1	Парадигмы программирования	135
6.2	От процедурного программирования к объектному	137
6.3	Основные принципы и этапы объектно-ориентированного программирования	143
6.4	Объектно-ориентированные языки программирования	149
6.5	Этапы разработки программных систем с использованием ООП . .	150
6.6	Объектная декомпозиция	152
6.7	Объекты и сообщения	156
6.8	Классы	158
6.9	Основные средства разработки классов	163
7	Технология ООП программирования	171
7.1	Технология программирования и информатизация общества	172
7.2	Общие принципы разработки программных средств	174
7.3	Жизненный цикл программного средств	175
7.4	Понятие качества программного средства	177
7.5	Надежность программных средств	178
7.6	Методы борьбы со сложностью программных средств	180
7.7	Контроль принимаемых решений	180
8	Объектный подход к разработке программных средств	182
8.1	Особенности объектного подхода к разработке программного средства	185
8.2	Особенности объектного подхода на этапе конструирования программного средства	190
8.3	Особенности объектного подхода на этапе кодирования программного средства	191
8.4	Качество ПО и методы его обеспечения	192
8.5	Методы контроля качества	198
8.6	Ошибки в ПО	200
	Заключение	205
	Литература	206
	Глоссарий	210

ВВЕДЕНИЕ

В настоящее время компьютеры используются во всех сферах человеческой деятельности. В связи с этим в последние годы резко возросли сложность и размер решаемых задач. Для увеличения эффективности их решения уже недостаточно использования традиционных подходов: лучшая организация труда, подбор кадров и т. д. Более важным сейчас является выбор инструментария, т. е. системы программирования, языка программирования, т. к. современные системы содержат приемы, позволяющие эффективно решать большие задачи, контролировать критические ситуации (исключения). Многие аспекты программирования из искусства перешли уже в разряд «технология программирования». В последние 10–15 лет наибольшее применение нашла технология объектно-ориентированного программирования (ООП).

В основу ООП положено понятие объекта. По своей сути это простое понятие. Объект — это данные и операции, их обрабатывающие, любого уровня сложности, объединенные вместе. Причем в ООП наибольшее внимание уделяется не реализации объектов, а связям между ними. Эти связи организованы в виде сложных иерархических структур, где новые типы объектов создаются на основе имеющихся. В языке C++ существует большой набор стандартных объектов, но при решении новых задач приходится создавать новые объекты, и профессиональный программист должен уметь это делать.

ООП базируется на 3 основных принципах:

- 1) *Инкапсуляция* — сокрытие информации. Этот принцип предполагает создание пользовательских типов данных, включающих как данные, так и операции и функции, их обрабатывающие. Никакие другие данные не могут использовать эти операции и функции, и наоборот. Контроль за санкционированным использованием данных и функций выполняет компилятор. Такие данные называются абстрактными, в отличие от стандартных (встроенных) типов данных (**int**, **char**, ...). Механизм создания абстрактных типов данных осуществляется через понятие класса.
- 2) *Наследование* — создание иерархии абстрактных типов данных. Определяется базовый класс, содержащий общие характеристики (прародительский класс), а из него по правилам наследования строятся порожденные классы, сохраняющие свойства и методы базового класса и дополненные своими характерными свойствами и методами.

3) *Полиморфизм* — множественность форм. Это принцип использования одинаковых имен функций и знаков операций для обозначения однотипных действий. В языке C++ полиморфизм используется в двух видах:

- а) для обычных функций и операций над стандартными и абстрактными типами данных. Это так называемая «перегрузка функций и операций»;
- б) для функций, определенных в иерархии наследования. Это так называемые «виртуальные функции».

Язык C++ был создан в лаборатории Bell Labs в начале 80-х гг. программистом Бьярном Страуструпом. Первые компиляторы появились в 1985 г.

Отметим, что в настоящее время имеется много учебников по данному вопросу, но большинство из них ориентировано на профессиональных программистов. Данное учебное пособие предназначено в первую очередь для начинающих изучать технологию ООП и не имеющих навыки программирования на языке C++. Примеры, приведенные в учебнике, написаны на языке C++ и отлажены в среде MS Visual Studio.

Соглашения, принятые в книге

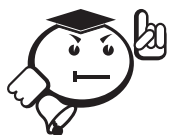
Для улучшения восприятия материала в данной книге используются пиктограммы и специальное выделение важной информации.



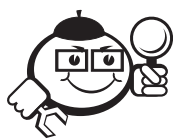
.....
Эта пиктограмма означает определение или новое понятие.



.....
 Эта пиктограмма означает внимание. Здесь выделена важная информация, требующая акцента на ней. Автор здесь может поделиться с читателем опытом, чтобы помочь избежать некоторых ошибок.



.....
 В блоке «На заметку» автор может указать дополнительные сведения или другой взгляд на изучаемый предмет, чтобы помочь читателю лучше понять основные идеи.



Пример

Эта пиктограмма означает пример. В данном блоке автор может привести практический пример для пояснения и разбора основных моментов, отраженных в теоретическом материале.



Контрольные вопросы по главе

Глава 1

ВОЗМОЖНОСТИ СИ++

Язык Си++ отличается от «обычного» языка программирования Си, прежде всего, тем, что он поддерживает объектно-ориентированное программирование. Но в нем есть и ряд полезных нововведений, а именно *использование*:

- подставляемых функций и значений параметров по умолчанию;
- ссылочных переменных и модификатора константы;
- новых операторов динамического распределения памяти;
- перегрузки функций и операций.

Перечисленные выше возможности не связаны с поддержкой объектно-ориентированного программирования, но существенно упрощают разработку программ. Поэтому и «обычные» программы часто удобнее разрабатывать на языке программирования Си++, чем на Си [1–7].

1.1 Локальные и глобальные переменные

В языке Си локальные переменные, а именно переменные, определяемые в подпрограмме (функции), должны быть описаны в начале подпрограммы.



.....
***Блоком** называется оператор или часть подпрограммы, ограниченная фигурными скобками. Блок может быть равен подпрограмме.*
.....

Локальная переменная может быть описана в начале любого блока перед первым исполняемым оператором. *Например,*

```
for (int i = 0; i < 10; i++)  
{ x[i]++; x[i]*=2; }
```

Переменная, которая не содержится ни в одном блоке (подпрограмме), называется глобальной. Если внутри блока нужно обратиться к глобальной переменной, то это делается с помощью оператора разрешения области видимости, который обозначается через «::». *Например,*

```
int i=0;    // глобальная переменная
int f();
{ int i=10;
  ::i++;    // глобальная переменная примет значение 11
}
```

Заметим, что признак начала комментария с символов «//» тоже относится к Си++.

1.2 Подпрограммы и их аргументы

Для реализации алгоритма применяется метод *процедурного* (модульного) программирования либо метод *структурного* программирования. При модульном программировании главная программа состоит из вызовов подпрограмм. При структурном программировании сначала разрабатывают схему, состоящую из блоков, а затем производят детализацию каждого блока.

Подставляемые функции

Одно из преимуществ второго подхода — экономия времени, расходуемого на вызов подпрограмм.

В Си++ можно превратить вызовы подпрограмм в генерацию кодов, составляющих эти подпрограммы, в местах вызова. Такие подпрограммы определяются модификатором **inline** и называются подставляемыми функциями. *Например,* вызов подпрограммы

```
inline int max(int a, int b)
{ return a > b ? a : b; }
```

эквивалентен действию макрокоманды

```
#define max(a, b) ((a) > (b) ? (a) : (b))
```

в том смысле, что в обоих случаях оператор

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

будет действовать одинаково, если не считать игнорирование типа аргументов во втором случае.

В общем случае макрокоманду определить нелегко и поэтому целесообразно применять подставляемые функции.

Значения параметров по умолчанию

При объявлении функции можно разместить константы, определяющие ее предполагаемые аргументы. Предполагаемый аргумент функции отделяется от объявления формального параметра знаком «=» (равно). Аргументы, заданные по умолчанию, должны быть последними в списке параметров.

Например, `int f(int n, int a = 1, char *txt = 0);`

— объявлена функция трех переменных, у которой значения переменной **a** и указателя **txt** определены по умолчанию. Рассмотрим функцию:

```
int f (int a, int b = 10)
{ return a + b; }
```

Если ее вызвать с помощью оператора `x = f(100)`, то переменная **x** примет значение, равное 110, а в случае `x = f(100, 20);` **x** будет равна 120.

Второе правило при определении параметров по умолчанию заключается в том, что при перегрузке функции вызываемая подпрограмма должна быть однозначно определена.

1.3 Определение данных

Передача параметров по ссылке



.....
Ссылочной переменной называется переменная, которая служит псевдонимом для другой переменной.

Например,

```
int a = 3;
int &x = a; // x — псевдоним для a
x = a * x;
```

— в результате `x = a = 9`.

Тип ссылочной переменной должен совпадать с типом переменной, для которой она служит псевдонимом, *например* объявления

```
float a = 3;
int &x = a; // ошибка
```

содержат ошибку, ибо тип переменной **x** — целый, а тип **a** — с плавающей точкой. Запрещаются также двойные ссылки, *например*

```
int &&x = a; // ошибка,
```

указатель-ссылка, *например*,

```
int &*p = &b; // ошибка,
```

массив-ссылка, а также ссылка-битовое поле, *например*

```
int &x : 1; // ошибка.
```

.....  Пример

```
float &Pi = 3.14;
```

— объявлена ссылочная переменная **Pi**, представляющая неявную вспомогательную переменную типа **float**, которой присвоено начальное значение 3.14.

.....  Пример

```
int a[] = {-1, 0, 1};
int *&p = a;
```

— объявлена ссылочная переменная **p**, как псевдоним имени массива.

.....

Ссылочные переменные применяются для изменения значений аргументов подпрограмм. Известно, что аргументы при вызове подпрограмм передаются через стек — перед вызовом они записываются в системный стек, а при возврате из подпрограммы восстанавливаются из стека, *например* если вызвать подпрограмму

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

с помощью оператора **swap(x, y)**, то значения переменных **x** и **y** не изменятся. Если же аргументы определить как ссылочные:

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

то вызов подпрограммы с помощью **swap(x, y)** приводит к перестановке переменных **x** и **y**, ибо через стек теперь будут передаваться не сами переменные, а их адреса.

Модификатор константы



.....
*Переменная, описанная как **const**, недоступна в других модулях программы, ее нельзя изменять во время выполнения программы. Единственная возможность присвоить ей значение — это инициализация при определении. Указатель, определенный с модификатором **const**, нельзя изменить, однако, может быть изменен объект, который адресуется этим указателем.*

Например, при объявлении и выполнении оператора:

```
char* const p = buffer;  
*p = 'x';
```

по адресу **buffer** будет записан символ «х». А если объявить **p** как указатель на строку констант

```
const char *p = buffer;
```

то аналогичная операция ***p = 'x';** будет ошибкой. Таким образом, модификатор **const** означает, что объект этого типа не может изменяться ни непосредственно, ни через указатель на него.



.....
*Адрес объекта, объявленного как **const**, не может быть присвоен указателю на переменные, которые могут изменяться, ибо в этом случае объект постоянного типа можно изменять через указатель.*

Например,

```
const char space = 'A';  
const char *p = &space; // верная запись  
char *q = &space;       // ошибка
```

Упражнение. Учитывая объявления

```
char c;           const char cc='a';  
char *pc;         const char *pcc;  
char *const cpc=&c;  const char *const cpcc=&cc;  
char *const *pcpc;
```

определить, какие из приведенных ниже присваиваний верные, а какие нет:

c=cc;	cc=c;	psc=&c;
psc=&cc;	pc=&c;	pc=&cc;
pc=psc;	pc=pc;	pc=pc;
pc=pc;	*pc=*pc;	pc=*pc;
**pc=pc;		*pc=*pc;

Ответ: Неверны присваивания *cc=c; pc=&cc; pc=psc; pc=pc;* (остальные присваивания верны).



*Модификатор **const** применяется в тех случаях, когда аргументы функции — ссылочные переменные, используемые для того, чтобы избежать копирования аргументов (которые могут быть достаточно большими), не предназначенных для модификации.*

Например, операция умножения объявляется как

```
complex operator*(const complex& z, const complex& w);
```

что приводит к передаче адресов объектов в подпрограмму умножения с сохранением всех остальных свойств передачи параметров как в Си.



*Модификатор **volatile** сообщает компилятору, что переменная может быть изменена некоторым фоновым процессом, например подпрограммой обработки прерываний или портом ввода-вывода.*

Это ключевое слово запрещает компилятору делать предположения относительно значения указанной переменной, ибо при вычислении выражений, включающих эту переменную, ее значение может измениться в любой момент и значение может находиться только в этой переменной. Компилятор должен брать значения только из этой переменной, а не использовать копию, находящуюся в регистре, что допустимо в других случаях. Рассмотрим, например, реализацию таймера, в котором переменная **ticks** модифицируется обработчиком временных прерываний:

```
Volatile int ticks;
Void timer()
{ ticks++; }
void wait(int interval)
{
    ticks=0;
    while(ticks<interval);
}
```

Предположим, что обработчик прерываний **timer()** надлежащим образом связан с аппаратным прерыванием от часов реального времени. Процедура **wait()** реализует цикл ожидания, пока значение **ticks** не станет равным интервалу времени, заданному параметром. Компилятор Си++ обязан перезагружать значение целой

переменной **ticks** типа **volatile** перед каждым сравнением внутри цикла, несмотря на то, что значение переменной внутри цикла не изменяется.

1.4 Операторы динамического распределения памяти

Для выделения и освобождения памяти в Си++ можно применять новые унарные операции — **new** и **delete**, соответствующие функциям **malloc** и **free** в Си.



.....
Формат операций:

- **new TYPE** выделяет область памяти для переменной типа **TYPE** и возвращает его адрес.
 - **new TYPE (значение)** действует как предшествующая операция и инициализирует область памяти начальным значением.
 - **new TYPE [n]** выделяет область памяти для массива из **n** элементов и возвращает его адрес.
 - **delete p** освобождает область памяти, на которую ссылается указатель **p**.
 - **delete []p** освобождает область памяти, занятую массивом, на который ссылается указатель **p**.
-



Пример

Приведём подпрограмму конкатенации строк, возвращающую адрес строки, полученной объединением двух строк.

```
// peregr.cpp
#include <iostream.h>           // библиотека ввода-вывода
#include <conio.h>              // консольный ввод-вывод
#include <string.h>             // библиотека функций для работы со строками
char* conc(char* s, char* t)
{
    int i;
    char *res=new char[strlen(s)+strlen(t)+1]; // результирующая строка
    for (i=0; i<strlen(s); i++)
        res[i]=s[i]; // копируем в результат сначала первую строку
    for (i=strlen(s); i<strlen(s)+strlen(t); i++)
        res[i]=t[i-strlen(s)]; // а затем вторую
    res[i]=0; // строка должна завершаться нулём
```

```

    return res;
}
void main()
{
    char* p="abc",*q="1234";           // объявим две строки p и q
    clrscr();
    cout<<"Входные данные:\nПервая строка "<<p;
    cout<<"\nВторая      "<<q;
    cout<<"\n\nВыходные данные:\n";
    cout<<"Результат конкатенации первой и второй строк      ";
    cout<<conc(p,q)<<"\n";           // результат конкатенации p и q — "abc1234"
    cout<<"Результат конкатенации первой, второй и снова первой строк ";
    cout<<conc(conc(p,q),p)<<"\n";   // результат конкатенации (p и q)
                                     // и p — "abc1234abc"
    getch();                         // ожидание нажатия клавиши
}

```

Подпрограмма **conc()** учитывает, что размер строки, полученной после объединения двух строк, равен сумме размеров этих строк. Здесь функция **strlen()** — стандартная и возвращает количество первых символов строки, не равных 0. Длина строки, вместе с завершающим ее нулем, равна **strlen()+1**. В данном примере сначала переписываются символы первой строки в результирующую строку **res**, а затем символы второй строки. В конце строки добавляется 0. Возвращается указатель на результирующую строку **res**.

Замечание. В рассмотренном примере вывод производится с помощью операции **cout<<данные**. Аналогичным образом в Си++ можно осуществлять ввод: **cin>>данные**. Например, вместо операторов:

```

    Scanf("%d",&x);           printf("\nx=%d",x);

```

для переменной **x** целого типа, можно написать:

```

    cin>>x;                     cout<<"\n"<<x;

```

указав в начале программы **#include <iostream.h>**. Отметим, что ввод с помощью **cin** отличается тем, что не требуется символ **&**.

1.5 Перегрузка функций и операций



В Си++ различные функции могут иметь одинаковое имя. Такие функции называются **перегружаемыми**.

Цель перегрузки (присвоения одинаковых имен) функций состоит в том, чтобы функция выполнялась по-разному в зависимости от типа и количества ее аргументов. *Например*, функция вычисления модуля целого числа, числа с плавающей точкой и вектора будет выполняться по-разному:

```
#include <stdio.h>           // библиотека стандартного ввода-вывода
#include <math.h>             // библиотека математических функций
struct Vector3d {           // структура трёхмерного вектора
    double x,y,z;           // состоит из трёх координат в пространстве
};
double absl(double x)       // эта функция возвращает модуль double
{
    if(x<0) return -x;      // воспользуемся определением модуля
    else return x;
}
double absl(Vector3d v)     // эта функция возвращает модуль (длину)
                           // трёхмерного вектора
{ return sqrt(v.x*v.x+v.y*v.y+v.z*v.z); } // корень квадратный из суммы
                                           // квадратов координат
int absl(int i)             // эта функция возвращает модуль целого числа
{
    if(i<0) return -i;     // воспользуемся определением модуля
    return i;
}
main()
{
    Vector3d n={3.14159, 2.71828, -1}; // n — трёхмерный вектор
    printf("\nВходные данные:\n");
    printf("Трёхмерный вектор n={%f,%f,%f}\n",n.x,n.y,n.z);
    printf("\nВыходные данные:");
    printf("\nМодуль вектора n равен %f",absl(n));
    // найдём модуль n
    printf("\nМодуль целого числа -1 равен %d",absl(-1));
    // вызов функции для int
    printf("\nМодуль double числа -1 равен %f",absl(-1));
    // вызов функции для double
}
```

Результаты работы программы

Входные данные:

Трёхмерный вектор n={3.141590,2.718280,-1.000000}

Выходные данные:

Модуль вектора n равен 4.273012

Модуль целого числа -1 равен 1

Модуль double числа -1 равен 1.000000

Аналогичным образом перегружаются операции. Перегрузка операций осуществляется для типов данных, определяемых структурами. Операция перегружается как функция, имя которой состоит из слова **operator** с добавленным справа символом операции. *Например*, подпрограмму конкатенации строк, определяемых структурой

```
Struct String{
    int length; // длина строки
    char *p;    // указатель на строку
}
```

можно определить как операцию сложения строк

```
String operator+(String s,String t);
```



Пример

Приведём *пример программы*, в которой определена такая операция:

```
#include <stdio.h> // библиотека стандартного ввода-вывода
#include <string.h> // библиотека функций для работы со строками
#include <conio.h> // библиотека консольного ввода-вывода
struct string { // структура string
    int length; // содержит длину
    char *p; // и саму строку
};
string operator+(string s, string t) // перегрузка операции +
{
    int i;
    string res; // результирующая строка
    res.p=new char[s.length+t.length+1]; // выделим память для строки
    strcpy(res.p, s.p); // копируем первую строку
    strcpy(res.p+s.length, t.p); // копируем вторую строку
    res.length=s.length+t.length; // заполняем поле структуры — длина строки
    return res;
}
void main()
{
    string s1={3,"abc"}, s2={4,"1234"},s3; // строки s1,s2,s3
    clrscr();
    printf("Входные данные:\n");
    printf("\nПервая строка   %s\n",s1.p);
    printf("Длина первой строки  %d\n",s1.length);
    printf("Вторая строка    %s\n",s2.p);
    printf("Длина второй строки   %d\n",s2.length);
    s3=s1+s2; // используем перегруженную операцию
```

```
printf("\nВыходные данные:\n");
printf("Результат конкатенации первой и второй строк  %s\n",s3.p);
printf("Длина результирующей строки  %d\n",s3.length);
    // результат конкатенации s1, s2 — "abc1234"  длина — 7
}
```

Результаты работы программы

Входные данные:	
Первая строка	abc
Длина первой строки	3
Вторая строка	1234
Длина второй строки	4
Выходные данные:	
Результат конкатенации первой и второй строк	abc1234
Длина результирующей строки	7



Пример

Те же самые результаты могут быть получены при запуске следующей программы, отличающейся от приведённой выше способом копирования входных строк в результирующую:

```
#include <stdio.h>    // библиотека стандартного ввода-вывода
#include <string.h>    // библиотека функций для работы со строками
#include <conio.h>     // библиотека консольного ввода-вывода
struct string {       // структура string
    int length;       // содержит длину
    char *p;         // и саму строку
};
string operator+(string s, string t) // перегрузка операции +
{
    int i;
    string res;        // результирующая строка
    res.p=new char[s.length+t.length+1]; // выделим память для строки
    for (i=0; i<s.length; i++)
        res.p[i]=s.p[i]; // копируем первую строку
    for (i=s.length; i<s.length+t.length; i++)
        res.p[i]=t.p[i-s.length]; // копируем вторую строку
    res.p[i]=0;         // строка завершается 0
    res.length=s.length+t.length; // заполняем поле структуры — длина строки
    return res;
}
void main()
```

```

{
    string s1={3,"abc"}, s2={4,"1234"},s3; // строки s1,s2,s3
    clrscr();
    printf("Входные данные:",s3.p);
    printf("\nПервая строка   %s\n",s1.p);
    printf("Длина первой строки   %d\n",s1.length);
    printf("Вторая строка   %s\n",s2.p);
    printf("Длина второй строки   %d\n",s2.length);
    s3=s1+s2;      // используем перегруженную операцию +
    printf("\nВыходные данные:\n");
    printf("Результат конкатенации первой и второй строк   %s\n",s3.p);
    printf("Длина результирующей строки   %d\n",s3.length);
        // результат конкатенации s1, s2 — "abc1234"   длина — 7
}

```

Отметим, что невозможно определить эту операцию с помощью

char* operator+(char* s, char* t),

поэтому приходится объявлять структуру, содержащую строку.

.....

Правила составления перегружаемых функций и операций:

- для перегружаемых операций (над структурами) отсутствует возможность передачи параметров по умолчанию;
- перегрузка функций не должна приводить к конфликту с параметрами, заданными по умолчанию. *Например*, нельзя определить две функции:

```

int f(int x=0);
int f();

```

ибо неясно, к вызову какой из этих функций приводит оператор **y=f()**;

- перегружаемые функции и операции не могут различаться только по типу возвращаемого значения, *например* объявление функций:

```

Void f(int);
int f(int);

```

является ошибочным.

.....



Пример

.....

Рассмотрим структуру, реализующую двумерный вектор. Определим для него операции суммы, разности, унарного минуса, скалярного произведения.

```

#include <stdio.h>    // библиотека стандартного ввода-вывода
struct Vector {      // структура вектора на плоскости
    double x,y;      // состоит из координат x и y
};
Vector operator+(Vector v, Vector w) // перегрузим операцию сложения
{
    Vector t;
    t.x=v.x+w.x; t.y=v.y+w.y; // складываются координаты двух векторов
    return t;
}
Vector operator-(Vector v, Vector w) // перегрузим операцию вычитания
{
    Vector t;
    t.x=v.x-w.x; t.y=v.y-w.y; // находится разность соответствующих
                               // координат двух векторов
    return t;
}
Vector operator-(Vector v) // перегрузим операцию унарного минуса
{
    Vector t;
    t.x=-v.x; t.y=-v.y; // найдём вектор,противоположно направленный
                        // и имеющий ту же длину, для данного
    return t;
}
double operator*(Vector v, Vector w) // перегрузим операцию умножения
{
    return v.x*w.x+v.y*w.y; // найдём скалярное произведение двух векторов
}
int main()
{
    Vector a={1,0}, b={-1,1},c,d,e;
    printf ("\nВходные данные:\n");
    printf ("Вектор a={%f,%f},b={%f,%f}\n",a.x,a.y,b.x,b.y);
    c=a-b;
    printf("\nРезультат вычитания a-b={%f,%f}",c.x,c.y); // вычитание
    printf("\nРезультат скалярного произведения a*b=%f",a*b); // произведение
    d=a+b;
    printf("\nРезультат сложения a+b={%f,%f}",d.x,d.y); // сложение
    e=-a ;
    printf("\nВектор противоположный a это вектор e={%f,%f}",e.x,e.y);
    // унарный минус
}

```

Результаты работы программы

Входные данные:

Вектор a={1.000000,0.000000},b={-1.000000,1.000000}

Выходные данные:

Результат вычитания $a-b=\{2.000000,-1.000000\}$

Результат скалярного произведения $a*b=-1.000000$

Результат сложения $a+b=\{0.000000,1.000000\}$

Вектор, противоположный a , это вектор $e=\{-1.000000,-0.000000\}$

.....



Контрольные вопросы по главе 1

.....

- 1) Чем язык Си++ отличается от «обычного» языка программирования Си?
- 2) Что такое локальные и глобальные переменные?
- 3) Что такое подставляемые функции?
- 4) Как определяются значения параметров по умолчанию?
- 5) Что такое передача параметров по ссылке?
- 6) Что такое ссылочная переменная?
- 7) Можно ли изменить указатель, определенный с модификатором **const**?
- 8) Какие вы знаете операторы динамического распределения памяти?
- 9) Что такое перегрузка функций и операций?
- 10) Какие функции называются перегружаемыми?

Глава 2

ОБЪЕКТЫ И КЛАССЫ



.....
***Классом** называется набор типизированных данных и функций, объединенных в новый тип данных.*
.....

В отличие от структуры, типизированные данные и функции имеют различные уровни доступа.



.....
*Переменная, объявленная в программе, имеющая определенный классом тип данных, называется **объектом этого класса**.*
.....



.....
*Таким образом, **класс** — это структура, состоящая из некоторых переменных (и констант) и функций, а **объект** — это область памяти, которую занимает структура при ее объявлении.*
.....



.....
*Для создания объектов предусмотрены принадлежащие классу функции, заполняющие поля объектов. Эти функции называются **конструкторами**.*
.....



.....
*При удалении объектов вызываются функции, принадлежащие классу и предназначенные для освобождения памяти, — **деструкторы** [8–13].*
.....

2.1 Класс как обобщение структуры

Выше мы дали определение класса, как конструкции, состоящей из полей и функций. В частности, этому определению удовлетворяет структура. В действительности, структура в Си++ реализована как класс, все поля которой, по умолчанию, общедоступны в том смысле, что доступ к ним осуществляется через имена *имя_структуры.поле*, *имя_структуры.функция(аргументы)* или указатели *указатель->поле*, *указатель->функция(аргументы)*.

Простейшим образом класс можно определить с помощью конструкции:

$$\left\{ \begin{array}{l} class \\ struct \\ union \end{array} \right\} \text{ имя } \{ \text{список компонент} \};$$

где первая пара фигурных скобок обозначает альтернативный выбор одного из ключевых слов, а вторая пара включает в себя поля и имена функций, которые принадлежат классу. Такие функции называются *составными функциями* класса. Заключенный в фигурные скобки список компонент называется *телом класса*. Определение тела класса заканчивается точкой с запятой.



Пример

Будем использовать ключевое слово **struct** для определения класса двумерного вектора, для которого определены функции ввода и вывода данных, составляющих объект.

```
#include <iostream.h>
#include <conio.h>
// Класс вектор
struct Vector
{
    double x, y;    // Координаты вектора
    // Функция вывода на экран координат вектора
    void get()
    { cout<<"x="<<x<<" "<<"y="<<y<<"\n"; }
    // Функция ввода с клавиатуры координат вектора
    void put()
    {
        cout<<"Введите через пробел координаты вектора (x и y): ";
        cin>>x>>y;
    }
};

void main()
{
    clrscr(); // Очистка экрана
    Vector v,w[2]; // Определение векторов
```

```

v.put();w[0].put();w[1].put(); // Ввод координат векторов
// Вывод координат векторов
cout<<"\nКоординаты вектора v: ";
v.get();
cout<<"Координаты вектора w[0]: ";
w[0].get();
cout<<"Координаты вектора w[1]: ";
w[1].get();
getch(); // Ожидание нажатия клавиши
}

```

Результаты работы программы

```

Введите через пробел координаты вектора (x и y): 12.4 3.56
Введите через пробел координаты вектора (x и y): 2.34 5.6
Введите через пробел координаты вектора (x и y): 7 8.02
Координаты вектора v: x=12.4 y=3.56
Координаты вектора w[0]: x=2.34 y=5.6
Координаты вектора w[1]: x=7 y=8.02

```

.....

Тело составной функции может быть определено вне класса. В таком случае для этой функции указывается имя класса, членом которой она является:

тип_возвращаемого_значения имя_класса::функция(аргументы) {...}

Следует помнить, что указатель на объект, к которому принадлежит составная функция, определяется с помощью ключевого слова **this**. В частности, в предшествующем примере в функциях **put()** и **get()** переменные **x** и **y** будут равны **(*this).x** и **(*this).y**.



Пример

.....

Рассмотрим подпрограмму перегрузки операции присваивания для структуры, состоящей из строки и ее длины. В теле класса эта функция объявлена как

```
str&operator = (const str&);
```

она будет возвращать адрес объекта, полученного после присваивания. Это позволит применять цепочки присваиваний, например `str1 = str2 = str3`. Аргумент функции сделаем ссылочным, чтобы избежать копирования всего объекта в стек при вызове операции присваивания. В стек теперь будет сохраняться адрес объекта.

```

#include <string.h>
#include <iostream.h>
#include <conio.h>
// Класс строка
struct Str

```

```

{
char *s; // Указатель на строку
int len; // Длина строки
void init(const char*); // Функция инициализации строки
Str operator = (const Str); // Перегрузка операции =
};

// Перегрузка операции =
Str Str::operator = (const Str st)
{
len = st.len; // Выяснение длины новой строки
delete s; // Удаление старого содержимого
s = new char[len + 1]; // Выделение памяти под новую строку
strcpy(s, st.s); // Копирование строки
return *this; // Возвращение полученной строки по значению
}

// Функция инициализации строки
void Str::init(const char* s)
{
len = strlen(s); // Выяснение длины строки
Str::s = new char[len + 1]; // Выделение памяти под строку
strcpy(Str::s, s); // Копирование строки
}

void main()
{
clrscr(); // Очистка экрана
Str str1, str2, str3; // Создание строк
str1.init("Пирамида"); // Инициализация первой строки
str3 = str2 = str1; // Присваивание значения первой строки остальным
// двум строкам
cout<<"Объект str3 = " << str3.s << "\n"; // Вывод третьей строки
getch(); // Ожидание нажатия клавиши
}

```

Результаты работы программы

Объект str3 = Пирамида

.....

В этом примере мы столкнулись со следующей проблемой: подпрограмма **init** имеет формальный параметр с именем **s**, совпадающим с именем строки **s** в классе **Str**. Для того чтобы отличать имя строки в классе, применяется модификатор расширения области видимости «**::**». В данном случае к строке класса применяется обращение **Str::s**.

2.2 Определение первичного класса

Мы определили класс как тип данных, состоящий из полей (типов данных) и составных функций. Слияние данных и функций, работающих с этими данными, называется *инкапсуляцией*. Таким образом,

класс = данные + составные функции.

Помимо инкапсуляции в объектно-ориентированном программировании реализован механизм наследования, позволяющий строить иерархию типов, и полиморфизм, позволяющий объектам отвечать на запросы в соответствии с их типом.

Наследование реализовано с помощью понятия производного класса. Класс называется *производным* от другого класса, если он содержит все поля и функции этого другого класса. Если класс не является производным, то он называется *первичным*. В предшествующем параграфе фактически было дано определение первичного класса. Тело класса разбито на три части, соответствующие трем атрибутам:

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



.....
Атрибут **private** имеют члены класса, доступные только для составных и дружественных функций этого класса. Эти члены класса называются **закрытыми**.
.....



.....
Атрибут **protected** имеют члены класса, доступные для составных и дружественных функций классов, которые являются производными от этого класса или совпадают с ним. Эти члены класса называются **защищенными**.
.....



.....
Атрибут **public** имеют члены класса, обращение к которым осуществляется как к полям структуры. Эти члены называются **открытыми**.
.....

Если первичный класс объявлен с ключевым словом **class**, то первые его члены будут закрытыми по умолчанию, если как **struct**, то открытыми.

В случае **union** члены класса могут быть только открытыми. Например, если класс объявлен как

```
class Vector
{
    double x,y;
public:
    double getx() {return x;}
    double gety() {return y;}
};
```

то элементы класса **x** и **y** будут закрыты по умолчанию и обращение к ним, как к открытым членам, приведет к ошибке. Эти элементы можно будет читать с помощью функции **getx()** и **gety()**:

```
void main()
{
    Vector a;
    int z;
    z=a.x; // ошибка!
    z=a.getx(); // верно
}
```

Произвольная внешняя функция, прототип которой объявлен в теле класса и имеет модификатор **friend**, называется *дружественной* функцией этого класса.

Составные функции, определяемые внутри тела класса, будут подставляемыми (**inline**). Составные функции, определенные как внешние, с помощью оператора разрешения области видимости **“::”**, тоже можно сделать подставляемыми, указав для них модификатор **inline**. Но такое определение необходимо поместить перед первым использованием этой функции.



Определим класс, объектом которого является стек целых чисел. Для инициализации стека определим дружественную функцию. Для записи элемента в стек и для чтения элемента из стека определим функции **Push()** и **Pop()**.

```
#include <iostream.h>
#include <conio.h>
// Описание класса — целочисленный стек
class IntStack
{
    // Закрытые элементы
    int *v;    // У нас стек будет реализован в виде массива
    int size, top; // Размер стека и положение вершины
public: // Общедоступные элементы
    friend IntStack init(int size); //Дружественная функция // инициализации стека
```

```

    int pop();    // Извлечение числа из вершины стека
    void push(int x); // Занесение числа в стек
};
// Инициализации стека
IntStack init(int size)
{
    IntStack res; // Создаём новый стек
    res.v=new int [size]; // Выделяем память под массив
    res.size=size;      // Указываем размер стека
    res.top=size;       // Устанавливаем вершину стека
    return res;        // Возвращаем созданный стек
}
// Занесение числа в стек
inline void IntStack::push(int x)
    {if(top>0) v[--top]=x;}
// Извлечение числа из стека
inline int IntStack::pop()
    {
        if(top<size) return v[top++];
        else return 0;
    }
void main()
{
    clrscr(); // Очистка экрана
    IntStack s1, s2; // Создание стеков
    s1=init(10); s2=init(20); // Инициализация стеков
    cout<<"Заносим в стек s1 число -3\n";
    s1.push(-3);
    cout<<"Заносим в стек s2 число 1\n";
    s2.push(1);
    cout<<"Заносим в стек s1 число -2\n\n";
    s1.push(-2);
    cout<<"Извлекаем из стека s1 первое число "<<s1.pop();
    cout<<", затем второе "<<s1.pop()<<"\n";
    cout<<"Извлекаем из стека s2 число"<<s2.pop()<<"\n";
    getch(); // Ожидание нажатия клавиши
}

```

Результаты работы программы

```

Заносим в стек s1 число -3
Заносим в стек s2 число 1
Заносим в стек s1 число -2
Извлекаем из стека s1 первое число -2, затем второе -3
Извлекаем из стека s2 число 1

```

.....

Если функцию **Pop()** или функцию **Push()** определить за текстом главной программы, то модификатор **inline** приведет к ошибке, ибо компилятор при генерации кода главной программы использовал команды **call** и при встрече модификатора **inline** не может эти команды заменить на подставляемые функции.

2.3 Перегрузка операций

Операции могут быть перегружены с помощью составных и дружественных функций. Имя функции, соответствующей операции, получается добавлением символа операции к слову **operator**. Например,

```
class Bits
{
    char *b;
    int size;
public:
    Bits operator+(const Bits&);    // сложение
    Bits operator-(const Bits&);    // вычитание
    Bits operator-();               // унарный минус
    Friend Bits&operator ^ (const Bits&, const Bits&); // XOR
};
```

Если операция определяется с помощью составной функции, то эта функция имеет на один аргумент меньше, чем в том случае, когда операция определяется с помощью дружественной функции. Для составной функции первый аргумент предполагается равным ***this**. Например, для класса строки операцию сравнения относительно лексикографического (алфавитного) порядка можно определить с помощью приведённой ниже составной функции:

```
#include <string.h>
class String
{
    char *s;
    int len;
public:
    int operator<(String st)
    {
        return strcmp(s,st,s)<0;
    }
};
```

То же самое с помощью дружественной функции определяется следующим образом:

```
#include <string.h>
class String
{
    char *s;
    int len;
```

```

public:
    friend int operator<(String, String);
};
operator<(String str1, String str2)
{
    return strcmp(str1.s, str2.s)<0;
}

```



.....
 Перегрузка операций позволяет определить для классов значения любых операций, исключая «.», «::», «.*», «?:», **sizeof**.

Отсюда вытекает, что разрешено определять операции для класса, символы которых равны:

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

Здесь стандартная операция `->*` обозначает косвенное обращение к элементу класса (элементу структуры) через указатель на объект и указатель на этот элемент, *например*

```

class C
{
    int *d;
    friend int f(C *p);
};
int f(C *p) {return p->*d;}

```

Аналогично, операция `.*` обозначает прямое обращение к элементу класса по имени объекта и указателю на элемент. Приведем пример перегрузки операции индексации `[]`, обычно используемую для возвращения ссылки, что позволяет применять ее в операции присваивания с обеих сторон.



Пример

Пусть класс определен как строка символов. Определим операцию индексации, позволяющую читать и записывать *i*-й символ строки:

```

#include <string.h>
#include <conio.h>
#include <iostream.h>

```

```
// Класс строки
class String
{
    // Закрытые элементы
    char *s; // Сама строка
    int len; // Её длина
public: // Общедоступные элементы
    // Перегрузка операции []
    char& operator[](int pos)
    {
        return s[pos];
    }
    // Инициализация строки
    void init(char *s)
    {
        len=strlen(s); // Определение длины
        String::s=new char[len+1]; // Выделение памяти под строку
        strcpy(String::s, s); // Присваивание
    }
    // Вывод строки на экран
    void show()
    {
        cout<<s<<'\n';
    }
};

void main()
{
    clrscr(); // Очистка экрана
    String a; // Создаём строку
    a.init("abc"); // Инициализируем её
    cout<<"Начальное содержимое строки: ";
    a.show(); // Выводим строку на экран
    a[1]='c';
    cout<<"Содержимое строки после операции a[1]='\c\': ";
    a.show(); // Выводим строку на экран
    a[0]='b';
    cout<<"Содержимое строки после операции a[0]='\b\': ";
    a.show(); // Выводим строку на экран
    cout<<"Содержимое строки после операции a[0]=a[2]: ";
    a[0]=a[2];
    a.show(); // Выводим строку на экран
    getch(); // Ожидание нажатия клавиши
}
```

Результаты работы программы

Начальное содержимое строки: abc

Содержимое строки после операции `a[1]='c'`: acc

Содержимое строки после операции `a[0]='b'`: bcc

Содержимое строки после операции `a[0]=a[2]`: ccc

Вызов операции возвращает адрес `a[i]`. Присваивание `a[i]=x` записывает в этот адрес `x`.

.....

Операции `++` и `--` могут быть как префиксными и записываться `++x` или `--x`, так и постфиксными — `x++`, `x--`. Если определять префиксные операции через составные функции, то следует указать обычным образом тип возвращаемого значения.

Например,

```
class A
{
    A& operator++();
    A& operator--();
};
```

Префиксные операции `++` и `--` можно определить как дружественные функции от одной переменной, представляющие собой ссылку на объект данного класса.

Например,

```
class A
{
    friend A& operator++(A&);
    friend A& operator--(A&);
};
```

Постфиксные операции `++` и `--` определяются с помощью функций, имеющих дополнительный аргумент типа `int`, который на самом деле не используется. *Например:*

```
class A
{
    int x;
public:
    void operator++() {x=x+2;}
    void operator++(int) {x=x+1;}
};

void main()
{
    A b;
    b++; // b.x увеличили на 1
    ++b; // b.x увеличили на 2
}
```

2.4 Конструкторы

Для заполнения полей объекта при его инициализации мы вынуждены определять функцию **init()**. Составные функции не позволяют устанавливать значения констант и переменных ссылочного типа, и необходимость вызова этих функций усложняет программирование и делает программу громоздкой. Для заполнения полей объекта при его создании и для освобождения полей при его удалении в Си++ предусмотрено специальное средство, которое состоит из составных функций, называемых *конструктором* и *деструктором*.



.....
*Составная функция класса, вызываемая при создании объекта, называется **конструктором**.*

Поскольку объект создается при его появлении в области видимости и с помощью оператора **new**, то в этих случаях будет вызываться конструктор.



.....
Основные свойства конструктора:

- Класс может иметь несколько конструкторов. Имя каждого конструктора совпадает с именем класса, для которого этот конструктор определен. Конструкторы могут быть перегружены и иметь параметры по умолчанию.
 - Конструктор не имеет возвращаемого значения.
 - Конструктор может быть объявлен и в закрытой, и в защищенной, и в открытой части класса. Если конструктор объявлен в закрытой части, то объекты этого класса могут создаваться только дружественными функциями, а если в защищенной — дружественными функциями производного класса.
 - Конструктор может быть определен вне тела класса.
-



..... **Пример**

Определим класс двумерного вектора. Будем инициализировать его с помощью полярных координат:

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

```

class Vector
{
    double x, y;
public:
    Vector(double rho, double phi);
    void show()
    {
        cout << "Вектор = (" << x << ", " << y << ")\\n";
    }
};

Vector::Vector(double rho, double phi = 0)
{
    x = rho*cos(phi);
    y = rho*sin(phi);
}

void main()
{
    clrscr();
    Vector v(1), w(-1, 0.5);
    v.show(); w.show();
    getch();
}

```

Результаты работы программы

```

Вектор = (1, 0)
Вектор = (-0.877583, -0.479426)

```

Обращение к конструктору осуществляется одним из трех способов:

- 1) **имя объект(параметры);**
- 2) **имя объект = имя(параметры);**
- 3) **имя объект = параметр;**

где **имя** обозначает имя класса. Второй способ называется *явным*, третий — *сокращенным*.



Пример

Определим класс, объектом которого является стек заданного размера. Продемонстрируем способы вызова конструктора:

```

class IntStack
{
    int *v, size, top;
public:

```

```

    IntStack(int size);
};
IntStack::IntStack(int size)
{
    v = new int[IntStack::size = size];
    top = size;
}
void main()
{
    IntStack s1(1000);
    IntStack s2 = IntStack(1000); // явный
    IntStack s3 = 1000;           // сокращенный
}

```

.....

В данном примере будут определены 3 стека, по 1000 элементов в каждом.

2.5 Список инициализации



.....

*Конструктор имеет дополнительное средство для инициализации полей. Это средство называется **списком инициализации**.*

.....

Обычные составные функции не имеют этого средства. Список инициализации находится в заголовке определения конструктора и отделяется от прототипа конструктора двоеточием. Он состоит из слов **поле(выражение)**, инициализирующих поле арифметических выражений над аргументами.

Например:

```

class Arr // массив чисел с плавающей точкой
{
    int n; // максимальное число элементов
    double *p; // указатель на массив
public:
    Arr(int size, double *a): n(size), p(a) {}
};

```

В данном примере список инициализации означает то же самое, что и присваивания **n = size** и **p = a**.

Преимущество списка инициализации заключается в том, что он позволяет задавать начальные значения констант и псевдонимов (ссылок). *Например:*

```

class Pair
{
    const int n;
}

```

```

    int& adr;
public:
    Pair(int n, int a): n(n), adr(a) {}
};
void main()
{
    int p = 10;
    Pair b(5, p); // в результате b.n = 5, b.adr = 10
}

```

В этом примере присваивания **Pair::n = n; adr = a;** — не допускаются.



Пример

Список инициализации и определение двумерного массива с помощью перегрузки операции скобок.

```

#include <conio.h>
#include <iostream.h>
// Класс — двумерный массив
class twomas
{
    // Закрытые элементы
    int *p;          // Массив
    const int m, n;  // Размерность массива
public: // Общедоступные элементы
    int& operator () (int i, int j) // Перегрузка операции ()
    {
        return p[(i-1)*n + j - 1];
    }
    twomas(int m0, int n0):m(m0), n(n0) // Конструктор
    {
        p = new int [m*n]; // Выделяем память под массив
    }
};
void main()
{
    int i, j; // Переменные для циклов
    twomas t(10, 15); // Создаём двумерный массив t размером 10 × 15
    clrscr(); // Очистка экрана
    cout<<"Содержимое двумерного массива:\n";
    // Заполнение ячеек массива и вывод его содержимого на экран
    for (i=1; i<=10; i++)
    {
        for (j=1; j<=15; j++)
        {

```

```

    t(i, j) = 100*i+j;
    cout<<' ' <<t(i,j);
}
cout<<'\n'; // Перевод строки
}
getch(); // Ожидание нажатия клавиши
}

```

Результаты работы программы

Содержимое двумерного массива:

```

101 102 103 104 105 106 107 108 109 110 111 112 113 114 115
201 202 203 204 205 206 207 208 209 210 211 212 213 214 215
301 302 303 304 305 306 307 308 309 310 311 312 313 314 315
401 402 403 404 405 406 407 408 409 410 411 412 413 414 415
501 502 503 504 505 506 507 508 509 510 511 512 513 514 515
601 602 603 604 605 606 607 608 609 610 611 612 613 614 615
701 702 703 704 705 706 707 708 709 710 711 712 713 714 715
801 802 803 804 805 806 807 808 809 810 811 812 813 814 815
901 902 903 904 905 906 907 908 909 910 911 912 913 914 915
1001 1002 1003 1004 1005 1006 1007 1008 1009 1010 1011 1012 1013 1014 1015

```

2.6 Деструктор



Деструктором называется составная функция класса, которая вызывается перед разрушением объекта.

Это означает, что деструктор вызывается в следующих случаях:

- при выходе из области видимости;
- при выполнении операции **delete** для объектов, размещенных в динамической памяти;
- непосредственно, как составная функция.

Уже обсуждалось, что класс может иметь несколько конструкторов, все эти конструкторы имеют одинаковое имя, совпадающее с именем класса. Приведём правила определения деструкторов:

- класс имеет ровно один деструктор;
- имя деструктора совпадает с именем класса с добавленным впереди символом тильды «~»;
- деструктор не имеет аргументов и не имеет возвращаемого значения.

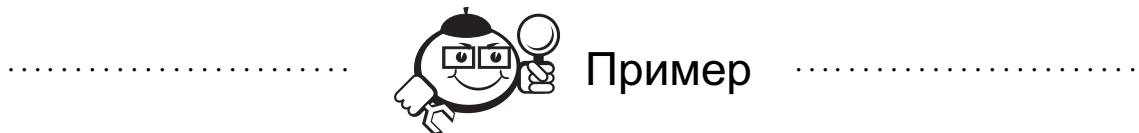

```

IntStack s; // Создание стека s (по умолчанию 10 элементов)
for(int i = 0; i < 10; i++)
    s++ = i;    // запись чисел 0, 1, 2,... в стек
cout<<"Содержимое стека:\n";
for(int j = 0; j < 10; j++)
    cout << --s << '\n';    // Чтение из стека
(*ps)++ = 100; // Пример занесения в стек ps числа 100
s.~IntStack(); // Явное разрушение стека s
delete ps; // Разрушение стека ps
getch(); // Ожидание нажатия клавиши
return 0;    // Выход из программы
}

```

В результате работы этой программы на экран будут выведены числа 9, 8, 7, 6, 5, 4, 3, 2, 1, 0, каждое с новой строки.

.....



Определим класс трассировочного типа, позволяющий отслеживать вход в блок и выход из блока программы, для которого определен объект этого класса. Воспользуемся с этой целью свойством запуска конструктора при вхождении в блок, содержащий объект, инициализируемый этим конструктором. При выходе из блока будет вызываться деструктор.

```

#include <stdio.h>
#include <conio.h>
#include <iostream.h>
// Трассировочный класс
class trace
{
    const char* msg;
public:
    trace(char *m): msg(m) // Конструктор
    {
        // Вывод сообщения о входе в блок
        fprintf(stderr, "Входим в %s\n", msg);
    }
    ~trace() // Деструктор
    {
        // Вывод сообщения о выходе из блока
        fprintf(stderr, "Выходим из %s\n", msg);
    }
};
void subr()

```

```
{
    trace t("subr");
}
int main()
{
    clrscr(); // Очистка экрана
    trace t("main");
    cout<<"\n"; // Перевод строки
    subr();
    cout<<"\n"; // Перевод строки
    for(int i = 0; i < 5; i++)
    {
        trace t("internal");
    }
    cout<<"\n"; // Перевод строки
    return 0; // Выход из программы
}
```

Результаты работы программы

Входим в main
Входим в subr
Выходим из subr
Входим в internal
Выходим из internal
Входим в internal
Выходим из internal
Входим в internal
Выходим из internal
Входим в internal
Выходим из internal
Входим в internal
Выходим из internal
Выходим из main

.....

2.7 Дружественные классы

Напомним, что для того, чтобы функция, определенная обычным образом, получила доступ ко всем членам класса, включая закрытые, ее следует объявить *дружественной*, указав в теле класса ее прототип с ключевым словом **friend**. Функцией, дружественной классу, может быть как произвольная внешняя функция, так и составная функция другого класса, который должен быть уже определен. *Например*, если мы хотим перегрузить операцию вывода на экран элементов двумерного

массива, определенного выше классом **twomas**, то нам будет нужен доступ к закрытым членам класса. Для обеспечения этого доступа в теле класса следует объявить прототип функции

```
friend ostream& operator << (ostream& o, twomas& d);
```

и определить операцию вывода с помощью внешней подпрограммы (функции):

```
ostream& operator << (ostream& o, twomas& d)
{
    int i, j;
    for(i = 1; i <= d.n; i++)
    {
        for(j = 1; j <= d.n; j++)
            o << d(i, j) << ' ';
        o << "\n";
    }
    return o;
}
```

Если класс содержит объекты другого класса в качестве полей, то его составным функциям необходим дополнительный доступ к закрытым членам этих полей. Например, в классе вектора, состоящего из **n** комплексных чисел

```
class Vectorc
{
    Complex *z;
public:
    double norma();
};
```

составная функция **norma()**, возвращающая максимум абсолютных величин вещественных и мнимых компонент вектора, не имеет доступа к этим компонентам. Чтобы обеспечить этот доступ, в классе **Complex** следует указать функцию **Vectorc::norma()** как дружественную. Поскольку к моменту объявления составной функции класс **Vectorc** должен быть определен, то следует указать перед определением класса **Complex** этот класс **Vectorc** как внешний (глобальный):

```
class Vectorc;
class Complex
{
    double Re, Im;
    friend double Vectorc::norma();
    ...
};
```

а затем определить класс **Vectorc**.

Существует возможность сделать доступными все члены класса **A** для каждой из составных функций класса **B**. Для реализации этой возможности достаточно класс **B** объявить дружественным для класса **A**. К моменту объявления класс **B**

должен быть определен или объявлен как внешний. Члены класса *A* не становятся доступными для дружественных функций класса *B*.



Пример

Определим класс графического окна, в которое будет выводиться график линейной функции. Линейная функция определяется как класс.

```
#include <graphics.h>
#include <conio.h>
class Wnd; // Прототип класса Wnd
// Класс — функция
class Func
{
    // Закрытые элементы
    double k, b; //  $y = kx + b$ 
    friend class Wnd; // Объявление дружественного класса
    public: // Общедоступные элементы
        Func(double k1, double b1=0): k(k1), b(b1) {} // Конструктор
};
// Класс окна
class Wnd
{
    // Закрытые элементы
    int xleft, xright, ytop, ybot; // Реальные координаты окна
    double xmin, ymin, xmax, ymax; // Относительные координаты окна
    public: // Общедоступные элементы
        // Конструктор
        Wnd(double x0, double y0, double x1, double y1,
            int xl=0, int yt=0, int xr=639, int yb=479):
            xmin(x0), ymin(y0), xmax(x1), ymax(y1),
            xleft(xl), ytop(yt), xright(xr), ybot(yb)
        Wnd& operator << (Func); // Перегрузка операции <<
};
// Перегрузка операции <<
Wnd& Wnd::operator << (Func f)
{
    double xkof, ykof; // Коэффициенты перевода относительных
                        // координат в реальные
    xkof = (xright-xleft)/(xmax-xmin);
    ykof = (ybot-ytop)/(ymax-ymin);
    rectangle(xleft, ytop, xright, ybot); // Рамка
    line(xleft,
        ytop+(ymax-ymin)*ykof/2,
        xright,
```

```

    ytop+(ymax-ymin)*ykof/2); // Ось x
line(xleft+(xmax-xmin)*xkof/2,
    ytop,
    xleft+(xmax-xmin)*xkof/2,
    ybot); // Ось y
line((xright - xleft)/2 + xmin*xkof,
    (ybot - ytop)/2 - (xmin*f.k+f.b)*ykof,
    (xright - xleft)/2 + xmax*xkof,
    (ybot - ytop)/2 - (xmax*f.k+f.b)*ykof); // Вывод функции
return (*this);
}
void main()
{
    int gd=DETECT, gm;
    Wnd w(-5, -3, 5, 3); // Определение окна
    Func phi(1, 1); // Определение функции
    initgraph(&gd, &gm, " "); // Инициализация графики
    w<<phi; // Вывод функции phi в окно w
    getch(); // Ожидание нажатия клавиши
    closegraph(); // Закрытие графики
}

```

2.8 Статические элементы класса



.....

*Определяя класс в тексте программы, мы создаем новый тип данных, состоящий из полей и составных функций, объединенных в структуру. **Объектом** класса называется переменная этого типа.*

.....

Объект отличается от класса тем, что он занимает некоторую область памяти, тогда как класс описывает структуру полей, составляющих эту область. Поля двух различных объектов класса не связаны между собой. Чтобы получить область, которая является общей для всех объектов класса, достаточно описать поля этой области с атрибутом **static**. Описанные таким образом поля называются *статическими*. Более точно, статическими называются элементы класса, которые являются общими для всех объектов этого класса.



Пример

.....

Определим класс строки, содержащий подпрограмму, возвращающую количество объектов этого класса, находящихся в области видимости. Определим статическую переменную **many** целого типа. Конструктор объекта будет увеличивать

эту переменную на 1, а деструктор — уменьшать на 1. Распределение области памяти, занимаемой объектами класса, приведено в таблице 2.1.

Таблица 2.1 – Распределение области памяти, занимаемой объектами класса

Int many				— поле, содержащееся в каждом объекте — поля объектов, содержащие указатели на строки и длины строк
Объект char *s int length	Объект char *s int length	...	Объект char *s int length	

Приведём текст программы:

```
#include <iostream.h>
#include <conio.h>
#include <string.h>
class Vstring
{
// Закрытые элементы
static int many; // Количество объектов Vstring
char *s; // Строка
int length; // Длина строки
public: // Общедоступные элементы
Vstring(char *text) // Конструктор
{
length = strlen(text); // Вычисление длины
s = new char[length+1]; // Выделение памяти
strcpy(s, text); // Копирование строки
many++; // Увеличение числа объектов
}
~Vstring() // Деструктор
{
delete s; // Освобождение памяти
many--; // Уменьшение числа объектов
}
static int Number() { return many; } // Статическая функция
// Общая функция
void get()
{
cout << s << "\n";
}
};
int Vstring::many = 0; // Установка начального числа объектов
void main()
{
clrscr(); // Очистка экрана
cout << "Количество объектов Vstring: " << Vstring::Number() << "\n";
```

```

Vstring u("12345");
cout << "Количество объектов Vstring: " << Vstring::Number() << '\n';
Vstring v("12345");
cout << "Количество объектов Vstring: " << Vstring::Number() << '\n';
cout << "Значение объекта v: ";
v.get();
cout << '\n';
for(int i = 0; i < 3; i++)
{
    cout << "Количество объектов Vstring: " << Vstring::Number() << '\n';
    Vstring v("12345");
    cout << "Количество объектов Vstring: " << Vstring::Number() << '\n';
    getch();
}
}

```

Результаты работы программы

```

Количество объектов Vstring: 0
Количество объектов Vstring: 1
Количество объектов Vstring: 2
Значение объекта v: 12345
Количество объектов Vstring: 2
Количество объектов Vstring: 3
Количество объектов Vstring: 2
Количество объектов Vstring: 3
Количество объектов Vstring: 2
Количество объектов Vstring: 3

```

.....

2.9 Шаблоны функций

Многие алгоритмы не зависят от типа данных, которые они обрабатывают. Например, перестановка двух переменных:

```

Type x, y, temp;
...
temp = x; x = y; y = temp;

```

будет работать для **Type = int**, для **Type = double** и для любого нового типа, определенного в программе с помощью класса. Логика алгоритма одинакова для всех типов данных. Эту ситуацию можно обобщить. Многие алгоритмы допускают отделение метода от данных. Программы, реализующие такие алгоритмы, можно отлаживать для данных одного типа, а затем применять для обработки данных других типов. Функции, реализующие эту возможность, называются параметризованными. Аргументы этих функций определяются с помощью ключевого слова

template. Тип, который определяет это ключевое слово, называется шаблоном. Общая форма определения шаблона функции **template** приведена ниже:

```
template <class T> прототип функции (аргументы)
{
    тело функции;
}
```

Здесь символ **T** обозначает тип данных, используемый функцией, прототип функции состоит из типа возвращаемого значения и имени функции. *Например*, алгоритм перестановки значений двух аргументов можно реализовать следующим образом:

```
template <class Tswp> void swap(Tswp& x, Tswp& y)
{
    Tswp temp;
    temp = x; x = y; y = temp;
}
void main()
{
    int a = 1, b = 2;
    double c = 1.1, d = 2.2;
    swap(a, b); // Перестановка целых чисел
    swap(c, d); // Перестановка чисел с плавающей точкой
}
```

Таким образом, для объявления шаблона функции, функция описывается стандартным образом, но перед ее прототипом ставится ключевое слово **template**, за которым следует заключенный в угловые скобки список параметров. Каждый из этих параметров определяется с помощью ключевого слова **class**, за которым следует *идентификатор*. Идентификатор служит для замещения имени типа. При вызове функции предполагается, что ключевое слово **class** в контексте шаблонов означает любой тип, а не только класс.

Например, максимум двух значений типа **T** можно вычислять с помощью функции:

```
template <class T> // Ключевое слово и параметр
const T& Max(const T& a, const T& b) { return a>b? a:b; }
void main()
{
    int i = 1, j = 2;
    float r = 1.1, s = 1.2;
    int k = Max(i, j);
    float t = Max(r, s);
}
```

Параметризованные функции могут быть перегружены другими функциями, которые тоже могут быть параметризованы, например:

```
template <class T> const T& Max(const T&, const T&);
```

```
template <class T> const T& Max(const T*, int);
int Max(int, int);
```

Для определенных типов эти функции могут быть перегружены (и переопределены) для того, чтобы выполнять (или не выполнять) какие-либо действия, которые функции-шаблоны не выполняют (или выполняют), *например*:

```
const char* Max(const char* c, const char* d)
{
    // Выполнить действия, специфичные для char*
}
```



Пример

Параметризованная функция бинарного поиска в отсортированном массиве.

```
#include <iostream.h>
#include <conio.h>
template <class Type>
int binsearch(Type* x, int count, Type key)
{
    int low, high, mid; // Левый, правый и средний элементы
    low = 0; high = count - 1;
    while (low <= high)
    {
        mid = (low+high)/2; // Вычисление середины массива
        if(key < x[mid]) high = mid - 1; // Если нужный элемент
                                   // находится слева от середины
        else if(key > x[mid]) low = mid + 1; // Если справа
        else return mid; // Нашли
    }
    return -1; // Не нашли
}
void main()
{
    clrscr(); // Очистка экрана
    int nums[] = 1, 2, 3, 5, 7, 11, 13, 17; // Массив, в котором ищем
    cout << "5 находится на " << binsearch(nums, 8, 5)
    cout << "месте в массиве.";
    getch(); // Ожидание нажатия клавиши
}
```

Результаты работы программы

5 находится на 3 месте в массиве.



Пример

Приведём параметризованную подпрограмму (функцию) сортировки методом пузырька и применим ее для упорядочения массива, состоящего из целых чисел, записанных в неупакованном BCD-формате. Ниже следует подпрограмма (функция) сортировки и программа тестирования.

```
#include <iostream.h>
#include <conio.h>
template <class Type>
void bubble (Type *x, int n) // Сортировка методом пузырька
{
    int i, j;
    Type t;
    for(i = 1; i < n; i++)
        for(j = n-1; j >= i; --j)
        {
            if(x[j-1] > x[j])
            {
                t = x[j-1]; x[j-1] = x[j]; x[j] = t;
            }
        }
}
void main()
{
    clrscr(); // Очистка экрана
    int i;
    int nums[] = 10, 12, 11, 3, 5, 12, 10; // Исходный массив
    cout << "Исходный массив: ";
    for(i = 0; i < 7; i++) cout << nums[i] << " ";
    cout << "\n";
    bubble (nums, 7); // Сортировка
    cout << "Отсортированный массив: ";
    for(i = 0; i < 7; i++) cout << nums[i] << " ";
    getch(); // Ожидание нажатия клавиши
}
```

Результаты работы программы

Исходный массив: 10 12 11 3 5 12 10

Отсортированный массив: 3 5 10 10 11 12 12

В неупакованном BCD-формате старшим байтом представляется знак, младшая цифра числа записывается как нулевой элемент массива, следующая цифра — первый элемент массива и т. д.

Например, число **123** представляется байтами:

$a[0] = '3', a[1] = '2', a[2] = '1', a[n-1] = '-'$.

Здесь n — количество разрядов числа.



Пример

Приведём *пример программы* для сортировки чисел, представленных в формате BCD. Для этого к параметризированной подпрограмме сортировки добавим определение класса чисел BCD и необходимые операции присваивания и сравнения. Получим следующий текст:

```
#include <iostream.h>
#include <string.h>
#include <conio.h>
template <class Type>
void bubble (Type *x, int n) // Сортировка методом пузырька
{
    int i, j;
    Type t;
    for(i = 1; i < n; i++)
        for(j = n-1; j >= i; --j)
        {
            if(x[j-1] > x[j])
            { t = x[j-1]; x[j-1] = x[j]; x[j] = t; }
        }
}
// Класс BCD чисел
class Bcd
{
    // Недоступные элементы класса
    static int n; // Максимальный размер BCD чисел
    char *a; // Массив под BCD число
public:
    // Общедоступные элементы класса
    // Перегрузка оператора =
    void operator = (char *b);
    // Перегрузка оператора >
    int operator > (Bcd x);
    void show(); // Вывод BCD числа на экран
};
// Перегрузка оператора =
void Bcd::operator = (char *b)
{
    int i;
    a = new char[n]; // Выделение памяти под BCD число
    for(i = 0; i < n; i++)
        a[i] = '0'; // Инициализация его нулями
    i = strlen(b); // Определение длины присваиваемого числа
```

```

int k = i - 1;    // Запоминаем её
// Копирование знака числа
if(b[0] == '+' || b[0] == '-')
{
    i--;
    a[n - 1] = b[0];
}
else a[n - 1] = '+';
// Копирование самого числа
for(int j = 0; j < i; j++) a[j] = b[k - j];
}
// Перегрузка оператора >
int Bcd::operator > (Bcd x)
{
    int i = 0;
    // Если первое число положительное, а второе — отрицательное, то
    // первое больше
    if(this->a[n-1] == '+' && x.a[n-1] == '-') return 1;
    // Если первое число отрицательное, а второе — положительное, то
    // первое меньше
    if(this->a[n-1] == '-' && x.a[n-1] == '+') return 0;
    // Сравнение по отдельным цифрам
    for(i = 1; i < n; i++)
    {
        if(this->a[n - 1 - i] > x.a[n - 1 - i])
        {
            if(x.a[n - 1] == '+') return 1;
            else return 0;
        }
        else if(this->a[n - 1 - i] < x.a[n - 1 - i])
        {
            if(x.a[n - 1] == '+') return 0;
            else return 1;
        }
    }
    return 0;
}
// Вывод BCD числа на экран
void Bcd::show()
{
    // Создание вспомогательной строки
    char *str;
    str = new char[n+1]; // Выделение под неё памяти
    str[0] = a[n-1];    // Копирование знака
    str[n] = '\0';    // Постановка конечного нуля
    // Копирование цифр
    int i;

```

```

for(i=n-2; i>=0; i--) str[n-i-1] = a[i];
// Вывод строки на экран
cout << str;
delete str; // Освобождение памяти
}

```

Теперь вызываем параметризованную функцию для сортировки массива Bcd:

```

int Bcd::n = 15; // Максимальная длина BCD числа
void main()
{
    clrscr(); // Очистка экрана
    Bcdx[10]; // Создание массива BCD чисел
    // Инициализация BCD чисел
    x[0] = "1234"; x[1] = "924"; x[2] = "-92"; x[3] = "0";
    x[4] = "-1"; x[5] = "10"; x[6] = "12"; x[7] = "1";
    x[8] = "-2"; x[9] = "12345";
    // Вывод неотсортированного массива
    cout << "Неотсортированные BCD числа:\n";
    for(int i = 0; i < 10; i++)
    {
        x[i].show();
        cout << "\n";
    }
    bubble(x, 10); // Сортировка методом пузырька
    // Вывод отсортированного массива
    cout << "Отсортированные BCD числа:\n";
    for(i = 0; i < 10; i++)
    {
        x[i].show();
        cout << "\n";
    }
    getch(); // Ожидание нажатия клавиши
}

```

Результаты работы программы

Неотсортированные BCD числа:

```

+000000000001234
+000000000000924
-000000000000092
+000000000000000
-000000000000001
+000000000000010
+000000000000012
+000000000000001
-000000000000002
+000000000012345

```

Отсортированные BCD числа:

```
000000000000092
000000000000002
000000000000001
+000000000000000
+000000000000001
+000000000000010
+000000000000012
+0000000000000924
+000000000001234
+000000000012345
```



Пример

Рассмотрим параметризованную подпрограмму (функцию) сортировки методом выбора. Сначала находим наименьший элемент массива и переставляем его с первым элементом. Затем из оставшихся элементов находим наименьший и переставляем его со вторым элементом и т. д. Для того чтобы не переставлять элемент сам с собой в том случае, когда он уже является наименьшим среди элементов подмассива, определим переменную **exchange**, которая будет равна **0**, если перестановки не нужны. Получим следующую подпрограмму:

```
template <class Type>
void select(Type *x, int count)
{
    int i, j, k, exchange;
    Type t;
    for(i=0; i<count-1; i++)
    {
        exchange=0;
        k=i; t=x[i];
        for(j=i+1; j<count; j++)
        {
            if(x[j]<t)
            { k=j; t=x[j]; exchange=1; }
        }
        if(exchange)
        { x[k]=x[i]; x[i]=t; }
    }
}
```

Вызов подпрограммы осуществляется следующим образом:

```
int nums[] = {1, 3, 8, -1, 12, -1, 15};
Select(nums, 7);
```



Пример

Рассмотрим параметризованную функцию быстрой сортировки Хоара. Алгоритм быстрой сортировки опирается на идею разбиения массива на две части с последующим рекурсивным применением подпрограммы сортировки к каждой из этих частей. Перед разбиением выбирается некоторый элемент массива, значение которого называется *компарандом*.

Все элементы, значения которых меньше компаранда, переносятся в левую часть массива, а элементы, имеющие большее значение, — в правую часть. Затем этот же процесс повторяется для каждой из частей. И так до тех пор, пока массив не будет отсортирован. Ниже приведён текст программы.

```
#include <iostream.h> // библиотека потокового ввода-вывода
#include <conio.h> // библиотека консольного ввода-вывода
// параметризованная функция быстрой сортировки Хоара
template <class Type>
void qs(Type *a, int left, int right)
{
    int i, j; // левая и правая границы массива
    Type x, y;
    i = left; j = right;
    x = a[(left+right)/2]; // определим «центр» массива
    do
    { // произведём поиск элементов для перестановки
        while(a[i]<x && i<right) i++;
        while(x<a[j] && j>left) j--;
        if(i<=j)
        { // выполняем перестановку
            y=a[i]; a[i]=a[j]; a[j]=y;
            i++; j--;
        }
    }
    while(i<=j);
    if(left<j) qs(a, left, j); // рекурсивный вызов для левой части
    if(i<right) qs(a, i, right); // рекурсивный вызов для правой части
}
// основная программа
void main()
{
    int i;
    int nums[]={5, 10, 12, 3, 8, 9, 2, 1}; // массив чисел для сортировки
    clrscr();
    cout<<"Входные данные (неотсортированный массив):\n";
    for(i=0; i<8; i++) cout << nums[i] << " ";
    qs(nums, 0, 7); // вызов подпрограммы сортировки
```

```
cout<<"\nВыходные данные (отсортированный массив):\n";  
for(i=0; i<8; i++) cout << nums[i] << " "; // вывод результатов на экран  
}
```

Результаты работы программы

Входные данные (неотсортированный массив):

5 10 12 3 8 9 2 1

Выходные данные (отсортированный массив):

1 2 3 5 8 9 10 12

.....



Контрольные вопросы по главе 2

.....

- 1) Что называется классом?
- 2) В чем отличие класса от структуры?
- 3) Что такое объект класса?
- 4) Что называется инкапсуляцией?
- 5) Что такое полиморфизм?
- 6) Какой класс называется первичным?
- 7) На какие три части разбито тело класса?
- 8) Что такое перегрузка операций?
- 9) Какие типы обращения к конструктору вы знаете?
- 10) Что такое шаблоны функции?

Глава 3

КОНТЕЙНЕРНЫЕ КЛАССЫ



.....
***Контейнерными** классами в общем случае называются классы, в которых хранятся организованные данные.*
.....

Например, массивы и связанные списки являются контейнерами.

Аналогично параметризованным функциям можно создавать параметризованные контейнерные классы. Благодаря этому, один раз разработанный и отлаженный контейнерный класс можно использовать для различных типов данных.

В данной главе мы будем рассматривать параметризованные контейнерные классы [14, 15].

3.1 Шаблоны классов

Общая форма объявления шаблона класса следующая:

```
Template <class Type>  
Class имя_класса  
{  
    тело класса;  
}
```

В теле класса может участвовать тип **Type**, а перед определением класса, как для шаблонов функции, в угловых скобках указывается название этого типа. В угловых скобках можно указать список типов, разделенных запятыми. В теле класса названия указанных типов можно использовать в любом месте. Конкретная реализация определенного таким образом класса создается с помощью следующей общей формы:

имя_класса < тип > объект;

где **тип** — тип переменной, которая будет параметром класса.



Пример

Определим параметризованный контейнерный класс — *массив*. Этот массив защищен в том смысле, что при записи и чтении его элементов контролируется выход за границы массива. Такой массив называется *ограниченным*.

```
#include <conio.h>
#include <iostream.h> // библиотека потокового ввода-вывода
#include <stdlib.h>    // стандартная библиотека
template <class Atype> class array
{
    Atype *a; // элементы массива
    int length; // число элементов
public:
    array(int size);        // конструктор
    ~array() {delete [] a;} // деструктор
    Atype& operator[] (int i); // получение элемента массива
};
template <class Atype>
array <Atype>::array (int size) // конструктор
{
    int i; length = size;
    a = new Atype[size];    // выделение памяти
    if(!a) {cout << "\nнет памяти для массива";
        exit(1);
    }
    for(i=0; i<size; i++) a[i] = 0; // запись нулей
}
template <class Atype>
Atype& array <Atype>::operator[](int i)
{
    if(i<0 || i > length-1)
    {
        cout << "\nзначение с индексом " << i;
        cout << "выходит за пределы массива";
        exit(1);
    }
    return a[i];
}
main()
{
    array <int> ix(20);    // массив целых чисел
```

```

array <double> dx(20);    // массив чисел с плавающей точкой
int i;
clrscr();
for(i=0; i < 20; i++) ix[i] = i;
cout << "\nmассив целых чисел" << "\n";
for(i=0; i < 20; i++) cout << ix[i] << " ";
for(i=0; i < 20; i++) dx[i] = (double) i;
cout << "\nmассив чисел с плавающей точкой: \n";
for(i=0; i < 20; i++) cout << dx[i] << " ";
ix[20] = 1;    // генерирует ошибку
return 0;
}

```

Результат работы программы

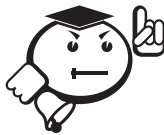
массив целых чисел:

0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19

массив чисел с плавающей точкой:

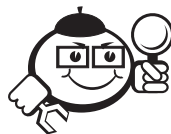
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19

значение с индексом 20 выходит за пределы массива



Правила определения шаблонов классов:

- шаблоны классов не могут быть вложены в другие классы;
- шаблоны классов могут иметь нетипизированные параметры; значения, указанные для этих параметров, должны быть константами.



Пример

Например, определим параметризованный класс стека. В данном примере иллюстрируется преимущество использования нетипизированных параметров.

```

template <class T, int size> // здесь size нетипизированный параметр
class
{
    T v[size];
    int top;
public:
    stack(): top(-1){}
    ~stack() {}
    void Push(const T& x)

```

```

    {
        if(top < size-1) {v[++top] = x;}
    }
    T& Pop() {if (top > -1) return v[top--];}
};
main()
{
    stack <int, 20> tiny;
    stack <int, 1000> huge;
    tiny.Push(-25);
}

```

.....

Параметр **size** используется для создания полей массива **v[]** без применения операции **new**, которая может быть выполнена неудачно. При таком определении типы **stack <int, 20>** и **stack <int, 1000>** будут различными типами. В частности, оператор объявления

```
stack <int, 20> *s = &tiny;
```

будет верным, а оператор

```
stack <int, 1000> *s = &tiny;
```

будет ошибочным.

- Шаблоны для определенных типов могут быть переопределены для того, чтобы выполнять (или не выполнять) какие-либо действия.

Поясним на *примере*. Пусть определён класс стека:

```

template <class T>
class Stack
{
    T *v;
    int size, top;
public:
    stack (int n); // n — размер стека
    ~stack();
    void Push (const T&); // записать T в стек
    T& Pop(); // извлечь T из стека
    ...
}

```

Переопределим его для **T = char***:

```

class Stack <char *>
{
    char ** v; // указатель на char*
    int size, top;
public:

```

```

Stack(int n);
~stack();
void Push(const char*&);
char* Pop();
...
// далее следуют новые функции Push и Pop
};

```

Шаблоны классов могут быть использованы структурами или объединениями, например:

```
Template <class T> struct S {T *x; ...}
```

- Статические члены параметризованного класса являются общими для *каждого конкретного экземпляра* этого класса.
- Шаблоны составных функций класса определяются вне класса, при помощи описания **template**. Например, определим для класса стека функцию **Push**:

```

Template <class T>
Void stack <T>:: Push(const T& element)
{
    if(top == size-1) error("stack overflow");
    else v[++top] = element;
}

```

- Шаблоны составных функций могут быть переопределены для отдельных типов. Например:

```

Void Stack <char > :: Push(const char& P)
{
    ...
}

```

3.2 Параметризованные очереди и стеки

Очередь представляет собой линейный список, доступ к элементам которого осуществляется по принципу **FIFO** («первым вошел — первым вышел»). Рассмотрим две функции обслуживания очереди: **qstore()** и **qretrieve()**. Функция **qstore()** помещает элемент в конец очереди, а функция **qretrieve()** извлекает из очереди первый элемент и возвращает его значение.



Приведем *пример* программы, создающей параметризованную очередь, размеры которой ограничены и задаются конструктором.

```

// очередь элементов типа Qtype
#include <conio.h>    // библиотека консольного ввода-вывода
#include <iostream.h> // библиотека потокового ввода-вывода
#include <stdlib.h>   // стандартная библиотека
template <class Qtype> class queue
{
    Qtype *q; // массив элементов очереди
    int sloc, rloc; // последний записанный элемент и последний прочитанный
    int length; // размер очереди
public:
    queue(int size); // конструктор
    ~queue() { delete [] q; } // деструктор
    void qstore(Qtype i); // запись в конец очереди
    Qtype qretrieve(); // получение первого элемента очереди
};

// конструктор
template <class Qtype>
queue <Qtype>:: queue(int size)
{
    size++; // размер на 1 больше
    q = new Qtype[size]; // выделим память
    if(!q) // если не удалось выделить память
    { cout << "\nнет памяти для очереди"; exit(1); }
    length = size; // длина очереди
    sloc = rloc = 0; // начало и конец очереди
}

// запись в очередь
template <class Qtype>
void queue <Qtype>:: qstore(Qtype i)
{
    if(sloc+1 == length) // если нельзя сместить указатель на конец
    {
        cout << "Очередь переполнена\n";
        return;
    }
    sloc++; // смещаем указатель
    q[sloc] = i; // записываем элемент
}

// чтение и удаление из очереди
template <class Qtype>
Qtype queue <Qtype> :: qretrieve()
{
    if(rloc == sloc) // если указатель на конец равен указателю на начало
    {
        cout << "\nОчередь пуста ";
        return 0;
    }
}

```

```

rloc++; return q[rloc];
}
// пример использования очереди
main()
{
    clrscr();           // очистка экрана
    queue <int> a(10), b(10); // две очереди с целыми числами
    queue <double> c(10); // одна очередь с числами с плавающей точкой
    a.qstore(100);
    a.qstore(200);
    b.qstore(300);
    b.qstore(400);
    cout<<"Первый элемент очереди a "<<a.qretrieve()<<"\n"; // выведет 100
    cout<<"Второй элемент очереди a "<<a.qretrieve()<<" "; // выведет 200
    cout << a.qretrieve() << "\n"; // "очередь пуста"
    c.qstore(-1);
    c.qstore(-2);
    cout<<"Первый элемент очереди c "<<c.qretrieve()<<" "; // выведет -1
    return 0;
}

```

Результат работы программы

```

Первый элемент очереди a 100
Второй элемент очереди a 200
Очередь пуста 0
Первый элемент очереди c -1

```

В данном примере после записи и чтения десяти элементов очереди индекс **sloc** будет указывать на конец массива, в результате чего дальнейшее использование очереди станет невозможным. Чтобы исправить этот недостаток, индекс последнего записанного элемента при достижении конца массива устанавливается на начало. При такой реализации программы в очередь можно будет поместить любое количество элементов, при условии, что элементы не только помещаются в очередь, но и извлекаются из нее. Такая реализация очереди называется *циклической очередью*, поскольку она использует массив, в котором хранятся элементы очереди, как циклический список.



Пример

Приведём текст программы, имеющей те же самые результаты, что и в предыдущем примере, но использующей в своей работе циклическую очередь.

```

// очередь элементов типа Qtype
#include <conio.h>           // библиотека консольного ввода-вывода

```

```

#include <iostream.h>    // библиотека потокового ввода-вывода
#include <stdlib.h>      // стандартная библиотека
template <class Qtype> class queue
{
    Qtype *q; // массив элементов очереди
    int sloc, rloc; // последний записанный элемент и последний прочитанный
    int length; // размер очереди
public:
    queue(int size); // конструктор
    ~queue() { delete [] q; } // деструктор
    void qstore(Qtype i); // запись в конец очереди
    Qtype qretrieve(); // получение первого элемента очереди
};
// конструктор
template <class Qtype>
queue <Qtype>:: queue(int size)
{
    size++; // размер на 1 больше
    q = new Qtype[size]; // выделим память
    if(!q) // если не удалось выделить память
        { cout << "\nнет памяти для очереди"; exit(1); }
    length = size; // длина очереди
    sloc = rloc = 0; // начало и конец очереди
}
// запись в очередь
template <class Qtype>
void queue <Qtype>::qstore(Qtype i) // добавление
{
    if((sloc+1 == rloc) || (sloc+1 == length) && !rloc)
        { cout << "\nОчередь переполнена"; return; }
    q[sloc] = i; sloc++;
    if(sloc == length) sloc = 0; // циклический список
}
// чтение и удаление из очереди
template <class Qtype>
Qtype queue <Qtype>:: qretrieve()
{
    if(rloc == length) rloc = 0;
    if(rloc == sloc)
        { cout << "\nОчередь пуста"; return 0; }
    rloc++;
    return q[rloc-1];
}
// пример использования очереди
main()
{

```

```

clrscr();           // очистка экрана
queue <int> a(10), b(10); // две очереди с целыми числами
queue <double> c(10);   // одна очередь с числами с плавающей точкой
a.qstore(100);
a.qstore(200);
b.qstore(300);
b.qstore(400);
cout<<"Первый элемент очереди а "<<a.qretrieve()<<"\n"; // выведет 100
cout<<"Второй элемент очереди а "<<a.qretrieve()<<" "; // выведет 200
cout << a.qretrieve() << "\n"; // "очередь пуста"
c.qstore(-1);
c.qstore(-2);
cout<<"Первый элемент очереди с "<<c.qretrieve()<<" "; // выведет -1
return 0;
}

```

Результат работы программы

```

Первый элемент очереди а 100
Второй элемент очереди а 200
Очередь пуста 0
Первый элемент очереди с -1

```

.....

Стек представляет собой линейный список, доступ к элементам которого осуществляется по принципу **LIFO** («последним вошел — первым вышел»).

Две основные операции традиционно называются **pop()** (извлечение) и **push()** (добавление).

Приведенная ниже программа реализует параметризованный класс стека.

```

#include <conio.h>           // библиотека консольного ввода-вывода
#include <iostream.h>        // библиотека потокового ввода-вывода
#include <stdlib.h>          // стандартная библиотека
template <class Stype>
class stack                // класс стека
{
    Stype *s;              // массив, содержащий элементы стека
    int tos;               // число записанных элементов
    int length;            // размер стека
public:
    stack(int size); // конструктор
    ~stack() {delete [] s;} // освобождение памяти
    void push(Stype i); // запись элемента в стек
    Stype pop(); // извлечение элемента из стека
};
// конструктор
template <class Stype>
stack <Stype>:: stack(int size)

```

```

{
    s = new Stype[size];    // захват памяти
    if(!s)                 // если s=0
    {
        cout << "\nНет памяти для стека";
        exit(1);
    }
    length = size; tos = 0;    // вершина стека
}
// запись объекта в стек
template <class Stype>
void stack <Stype>:: push(Stype i)
{
    if(tos == length) // если длина стека — максимум
    {
        cout << "\nСтек переполнен";
        return;
    }
    s[tos++] = i;           // сохраняем элемент в стеке
}
// чтение и удаление объекта из стека
template <class Stype>
Stype stack <Stype>:: pop()
{
    if(tos == 0)
    { cout << "\nСтек пуст"; return 0; }
    return s[--tos]; // получение элемента из стека
}
// примеры использования стека
main()
{
    stack <int> a(10); // стек целых чисел
    stack <double> b(10); // стек чисел с плавающей точкой
    clrscr();          // очистка экрана
    a.push(10); // запись в стек a
    b.push(-1); // запись в стек b
    a.push(20); // запись в стек a
    cout << "Последний элемент стека a " << a.pop() << "\n"; // вывод числа 20
    cout << "Последний элемент стека b " << b.pop(); // вывод числа -1
    return 0;
}

```

Результат работы программы

Последний элемент стека a 20

Последний элемент стека b -1

3.3 Бинарные деревья



По определению **бинарного дерева** каждый его элемент содержит собственную информацию и ссылки на левое и правое поддеревья, растущие из этого элемента.

Напомним, что *корнем* (root) называется первый элемент дерева. Элементы данных называются *узлами* (node). Узлы, не имеющие наследников, называются *листьями* (leaf), фрагмент дерева называется *поддеревом* (subtree) или *ветвью*. *Высота дерева* (height) определяется количеством уровней, на которых располагаются его узлы. Для определения параметризованного класса бинарного дерева будет использоваться следующий шаблон:

```
template <class DataT>
class tree
{
    DataT info; // информ. часть
    tree *left, *right; // указатели на поддеревья
public:
    tree <DataT> *root; // корень дерева
    tree() { root = NULL;} // конструктор
    // добавление элемента в дерево
    void stree(tree <DataT>*, tree <DataT>*, DataT);
    // удаление из дерева
    tree <DataT> * dtree(tree <DataT>*r, DataT key);
    void preorder(tree <DataT>*r); // обход прямой
    void inorder(tree <DataT>*r); // симметричный обход
    void postorder(tree <DataT>*r); // обратный обход
    void print_tree(tree <DataT>*r, int l); // вывод дерева
    // поиск в дереве
    tree <DataT> * search_tree(tree <DataT>*r, DataT key);
};
```

Рассмотрим функции для работы с бинарными деревьями. Каждый узел дерева будет объектом класса **tree**. Наличие указателя ***root** позволяет определить корень всего дерева, находясь в любом из его узлов.

Для включения новых элементов в дерево используется функция, описанная в теле класса как **stree**:

```
// параметризованная функция добавления элемента в дерево
template <class DataT>
void tree <DataT>::stree (tree <DataT> *r, tree <DataT> *previous, DataT info)
{
    if (!r)
    {
```

```

// если в качестве дерева для вставки передан NULL то
r = new tree <DataT>; // выделим память
if (!r)
{ cout<< "\nНедостаточно памяти"; exit(1); }
// создаётся новое дерево
r -> left = r -> right = NULL; // левое и правое поддеревья пусты
r -> info = info;
if (!root) root = r; // корень
else // если корень первоначального дерева не пуст
{
    // вставляем элемент на его место в дереве поиска
    if (info<previous -> info) previous -> left = r;
    else previous -> right = r;
}
return;
}
// если передано в качестве формального параметра не пустое дерево,
// то вставим элемент либо
if (info < r -> info)
    stree (r -> left, r, info); // в левое поддерево
else
    stree (r -> right, r, info); // либо в правое
}

```

Эта функция вставляет объект в бинарное дерево, отслеживая ссылки на элементы дерева, перемещаясь влево или вправо, в зависимости от содержимого поля **info**, до тех пор, пока для элемента не будет найдено соответствующее ему место в иерархии дерева. Функция **stree()** рекурсивна, как и большинство функций, связанных с бинарным деревом. При вызове функции первый аргумент будет равен указателю на корень дерева, или поддерева, в которое будет добавляться элемент, второй аргумент указывает на предшествующий корню элемент и может быть равен нулю.

Для прохождения дерева с выводом на экран значений его узлов можно использовать функцию обхода в симметричном порядке, описанную в теле класса как **inorder**:

```

// параметризованная функция обхода дерева в симметричном порядке
template <class DataT>
void tree <DataT>::inorder(tree <DataT> *r)
{
    if(!r) return; // если дерево пусто
    inorder (r -> left); // посетим левое поддерево
    if(r -> info) cout << r -> info << " "; // вывод элемента
    inorder (r -> right); // посетим правое поддерево
}

```

Эту функцию следует использовать, применяя в качестве аргумента указатель на корень созданного дерева. Значения элементов будут выведены в неубываю-

щем порядке. Приведем соответствующие функции для обхода дерева в прямом и обратном порядках, описанные в теле класса как **preorder** и **postorder** соответственно:

```
// параметризованная функция обхода дерева в прямом порядке
template <class DataT>
void tree <DataT>::preorder(tree <DataT>*r)
{
    if(!r) return;    // если дерево пусто
    if(r -> info) cout << r -> info << " ";    // вывод элемента
    preorder (r -> left);    // посетим левое поддерево
    preorder (r -> right);    // посетим правое поддерево
}
// параметризованная функция обхода дерева в обратном порядке
template <class DataT>
void tree <DataT>::postorder(tree <DataT>*r)
{
    if(!r) return;    // если дерево пусто
    postorder (r -> left);    // посетим левое поддерево
    postorder (r -> right);    // посетим правое поддерево
    if (r -> info) cout << r -> info << " "; // вывод элемента
}
}
```

Для вывода дерева на экран можно применить следующую подпрограмму, описанную в теле класса как **print_tree**:

```
// параметризованная функция печати дерева на экране
template <class DataT>
void tree <DataT>::print_tree(tree <DataT>*r, int l)
{
    int i;
    if(!r) return;    // если дерево пусто
    print_tree(r -> right, l+1);    // распечатаем правое поддерево на l+1 уровне
    for (i = 0; i < l; i++) cout << " "; // вывод необходимого количества пробелов
    cout << r -> info << endl;    // вывод информационной части
    print_tree(r -> left, l+1);    // распечатаем левое поддерево на l+1 уровне
}
}
```

Для полноты приведём подпрограммы поиска элемента (**search_tree**) и удаления (**dtree**) элемента бинарного дерева.

```
// параметризованная функция поиска поддерева с корнем, равным key
template <class DataT>
tree <DataT> *tree<DataT>::search_tree(tree <DataT>*r, DataT key)
{
    if (!r) return r;    // если дерево пусто
    while (r -> info != key) // цикл пока не найдено поддерево
    {
```

```

        if (key < r -> info) r = r -> left; // ищем слева
        else r = r -> right;           // ищем справа
        if (r == NULL ) break;        // если не нашли
    }
    return r;
}

```

Если элемент не найден, то эта подпрограмма возвратит **NULL**.

Подпрограмма удаления элемента реализуется рекурсивно:

// параметризованная функция получения нового дерева из имеющегося
 // путём удаления некоторого узла

```

template <class DataT>
tree <DataT>* tree <DataT>::dtree(tree <DataT> *r, DataT key)
{
    tree <DataT> *p;
    tree <DataT> *p2;           // r — корень дерева
    if (!r) return r;          // если элемент не найден
    if (r -> info == key)       // элемент это корень?
    {
        if (r -> left == r -> right) // если 1 элемент
        {
            // вернём пустое дерево
            if (root==r) root = NULL;
            return NULL;        // пустое дерево
        }
        else if(r -> left == NULL) // если левое поддереву пусто
        {
            p = r -> right;      // нет левого поддереву
            if (root == r) root = p;
            return p;
        }
        else if (r -> right == NULL) // если правое поддереву пусто
        {
            p = r -> left;       // нет правого поддереву
            if (r == root) root = p;
            return p;
        }
    }
    else                        // в противном случае
    {
        p2 = r -> right; // как минимум дерево из правого поддереву
        p = r -> right; // правые поддеревья
        while (p -> left) p = p -> left;
        p -> left = r -> left;
        if (r == root) root = p2;
        return p2;           // вернём новое дерево
    }
}

```

```

// если не корень, двигаемся
if (r -> info < key) r -> right = dtree(r -> right, key);    // вправо
else r -> left = dtree (r -> left, key);                    // и влево
// пока искомый элемент не станет корнем
return r;
}

```



Пример

Приведём *пример* программы, использующей вышеописанные функции для работы с бинарным деревом, информационной частью элементов которого являются символы, вводимые пользователем с клавиатуры.

```

#include <iostream.h> // библиотека потокового ввода-вывода
#include <conio.h> // библиотека консольного ввода-вывода
#include <process.h> // необходимо для использования функции exit
template <class DataT>
class tree
{
    DataT info;    // информ. часть
    tree *left, *right; // указатели на поддеревья
public:
    tree <DataT> *root;    // корень дерева
    tree() { root = NULL;} // конструктор
    // добавление элемента в дерево
    void stree(tree <DataT>*, tree <DataT>*, DataT);
    // удаление из дерева
    tree <DataT> * dtree(tree <DataT>*r, DataT key);
    void preorder(tree <DataT>*r);    // обход прямой
    void inorder(tree <DataT>*r);    // симметричный обход
    void postorder(tree <DataT>*r);    // обратный обход
    void print_tree(tree <DataT>*r, int l); // вывод дерева
    // поиск в дереве
    tree <DataT> * search_tree(tree <DataT>*r, DataT key);
};
// параметризованная функция добавления элемента в дерево
template <class DataT>
void tree <DataT>::stree (tree <DataT> *r, tree <DataT> *previous, DataT info)
{
    if (!r)
    {
        // если в качестве дерева для вставки передан NULL то
        r = new tree <DataT>; // выделим память
        if (!r)
        {

```

```

        cout<< "\nНедостаточно памяти"; exit(1);
    }
    // создаётся новое дерево
    r -> left = r -> right = NULL; // левое и правое поддеревья пусты
    r -> info = info;
    if (!root) root = r; // корень
    else // если корень первоначального дерева не пуст
    {
        // вставляем элемент на его место в дереве поиска
        if (info<previous -> info) previous -> left = r;
        else previous -> right = r;
    }
    return;
}

// если передано в качестве формального параметра не пустое дерево, то
// вставим элемент либо
    if (info < r -> info)
        stree (r -> left, r, info); // в левое поддерево,
    else
        stree (r -> right, r, info); // либо в правое
}

// параметризованная функция обхода дерева в симметричном порядке
template <class DataT>
void tree <DataT>::inorder(tree <DataT> *r)
{
    if(!r) return; // если дерево пусто
    inorder (r -> left); // посетим левое поддерево
    if(r -> info) cout << r -> info << " "; // вывод элемента
    inorder (r -> right); // посетим правое поддерево
}

// параметризованная функция обхода дерева в прямом порядке
template <class DataT>
void tree <DataT>::preorder(tree <DataT>*r)
{
    if(!r) return; // если дерево пусто
    if(r -> info) cout << r -> info << " "; // вывод элемента
    preorder (r -> left); // посетим левое поддерево
    preorder (r -> right); // посетим правое поддерево
}

// параметризованная функция обхода дерева в обратном порядке
template <class DataT>
void tree <DataT>::postorder(tree <DataT>*r)
{
    if(!r) return; // если дерево пусто
    postorder (r -> left); // посетим левое поддерево
    postorder (r -> right); // посетим правое поддерево
}

```

```

        if (r -> info) cout << r -> info << " "; // вывод элемента
    }
    // параметризованная функция печати дерева на экране
    template <class DataT>
    void tree <DataT>::print_tree(tree <DataT>*r, int l)
    {
        int i;
        if(!r) return;          // если дерево пусто
        print_tree(r -> right, l+1); // распечатаем правое поддерево на l+1 уровне
        for (i = 0; i < l; i++) cout << " "; // вывод необходимого количества пробелов
        cout << r -> info << endl;      // вывод информационной части
        print_tree(r -> left, l+1); // распечатаем левое поддерево на l+1 уровне
    }
    // параметризованная функция поиска поддерева с корнем, равным key
    template <class DataT>
    tree <DataT> *tree<DataT>::search_tree(tree <DataT>*r, DataT key)
    {
        if (!r) return r;      // если дерево пусто
        while (r -> info != key) // цикл пока не найдено поддерево
        {
            if (key < r -> info) r = r -> left;      // ищем слева
            else r = r -> right;                      // ищем справа
            if (r == NULL ) break;                    // если не нашли
        }
        return r;
    }
    // параметризованная функция получения нового дерева из имеющегося
    // путём удаления некоторого узла
    template <class DataT>
    tree <DataT>* tree <DataT>::dtree(tree <DataT> *r, DataT key)
    {
        tree <DataT> *p;
        tree <DataT> *p2;      // r — корень дерева
        if (!r) return r;      // если элемент не найден
        if (r -> info == key)   // элемент это корень?
        {
            if (r -> left == r -> right) // если 1 элемент
            {
                // вернём пустое дерево
                if (root==r) root = NULL;
                return NULL;    // пустое дерево
            }
            else if(r -> left == NULL) // если левое поддерево пусто
            {
                p = r -> right;    // нет левого поддерева
                if (root == r) root = p;
                return p;
            }
        }
    }

```

```

    }
    else if (r -> right == NULL) // если правое поддереву пусто
    {
        p = r -> left;    // нет правого поддерева
        if (r == root) root = p;
        return p;
    }
    else // в противном случае
    {
        p2 = r -> right; // как минимум дерево из правого поддерева
        p = r -> right; // правые поддеревья
        while (p -> left) p = p -> left;
        p -> left = r -> left;
        if (r == root) root = p2;
        return p2; // вернём новое дерево
    }
}
// если не корень, двигаемся
if (r -> info < key) r -> right = dtree(r -> right, key); // вправо
else r -> left = dtree (r -> left, key); // и влево
// пока искомый элемент не станет корнем
return r;
}
int main(void)
{
    char stemp[80]; // символьная строка
    int i=0; // счётчик
    tree <char> ch; // дерево
    tree <char> *ch1; // указатель на дерево
    ch1=new tree <char>; // выделим память для дерева
    clrscr(); // очистим экран
    cout << "Введите строку (она должна завершаться точкой):";
    cin >> stemp;
    do
    {
        // пока не встретилась точка, вставляем каждый элемент строки в дерево ch
        if (stemp[i]!='.') ch.stree(ch.root, NULL, stemp[i]);
        i++;
    }
    while (stemp[i] != '.');
    cout << "Обход первоначального дерева в прямом порядке:\n";
    ch.preorder(ch.root);
    cout << "\n";
    cout << "Обход первоначального дерева в обратном порядке:\n";
    ch.postorder(ch.root);
    cout << "\n";
}

```

```

cout <<"Обход первоначального дерева в симметричном порядке:\n";
ch.inorder(ch.root);
ch1->root=ch.dtree(ch.root,stemp[0]); // получение нового дерева
cout <<"\n";
cout <<"Обход дерева в прямом порядке после удаления первого введённого
элемента:\n";
ch1->preorder(ch1->root);
cout <<"\n";
cout <<"Печать окончательного вида дерева:\n";
ch.print_tree(ch.root,0);
return 0;
}

```

Результат работы программы

Введите строку (она должна завершаться точкой):123321453754.

Обход первоначального дерева в прямом порядке:

1 2 1 3 2 3 4 3 5 4 7 5

Обход первоначального дерева в обратном порядке:

1 2 3 4 5 7 5 4 3 3 2 1

Обход первоначального дерева в симметричном порядке:

1 1 2 2 3 3 3 4 4 5 5 7

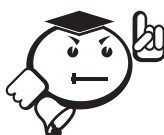
Обход дерева в прямом порядке после удаления первого введённого элемента:

2 1 3 2 3 4 3 5 4 7 5

Печать окончательного вида дерева:

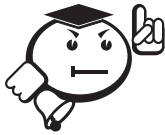
7
5
5
4
4
3
3
3
3
2
2
1

3.4 Определение класса множества



В дальнейшем множество будем представлять в виде *массива элементов*, среди которых нет одинаковых.

Если один массив получается из другого массива перестановкой элементов, то множества, которые представляются этими массивами, считаются равными.



.....
 Для множеств определены операции *объединения*, *пересечения*,
разности.

Класс множества определим следующим образом:

```
// параметризованный класс множества
template <class Stype>
class Set
{
    Stype *SetPtr;    // указатель на первый элемент
    int MaxSize;      // максимальное число элементов
    int NumMembers;   // количество элементов множества
    void insert (Stype member); // добавление элемента
    void remove (Stype member); // удаление элемента
    int find(Stype member);    // поиск элемента
    int ismember (Stype member); // принадлежность элемента
public:
    Set();           // конструкторы
    Set(int size);
    Set(const Set &ob);    // конструктор копирования
    ~Set() { delete SetPtr; } // деструктор
    Set <Stype> &operator = (Set <Stype> &ob); // присваивание
    Set <Stype> operator + (Stype member);    // добавление элемента
    friend Set<Stype> operator + (Stype member, Set <Stype> ob);
    Set <Stype> operator + (Set <Stype> &ob); // объединение
    Set <Stype> operator - (Stype member);    // удаление элемента
    Set <Stype> operator - (Set <Stype> &ob); // разность
    Set <Stype> operator & (Set <Stype> &ob); // пересечение
    // операции сравнения
    int operator == (Set <Stype> &ob);    // 1 если равно
    int operator != (Set <Stype> &ob);    // 1 если не равно
    int operator < (Set <Stype> &ob);    // 1 если подмножество
    friend int operator < (Stype member, Set <Stype> ob);    // 1 если элемент
    // множества
    operator int() return NumMembers; // преобразование в целое
    // ВВОД-ВЫВОД
    friend istream &operator >> (istream &stream, Set<Stype> &ob);
    friend ostream &operator << (ostream &stream, Set<Stype> & ob);
};
```

Класс **Set** можно использовать для построения множеств объектов любого типа. Класс содержит три поля, содержащих данные: **SetPtr**, **MaxSize** и **NumMembers**.

Область памяти для хранения элементов множества выделяется конструктором. Максимальное число элементов множества хранится в **MaxSize**. Количество элементов, содержащихся во множестве в текущий момент, равно **NumMembers**. Эти поля закрыты и доступны лишь для составных и дружественных функций класса, объявленных в теле класса.

Конструкторы. Первый конструктор без аргументов резервирует память для массива, состоящего из элементов, количество которых равно **DEFSET**. Значение **DEFSET** определяется с помощью внешней константы, например:

```
const int DEFSET = 100;
```

оно используется в конструкторе:

```
// параметризованный конструктор класса, вызываемый по умолчанию
template <class Type>
Set <Type>::Set()
{
    SetPtr = new Type [DEFSET];          // выделим память
    if(!SetPtr){ cout << "Нет памяти\n";
                exit(1);
            }
    NumMembers = 0; MaxSize = DEFSET;
}
```

Для построения множества заданного размера будем использовать конструктор:

```
// параметризованный конструктор с заданным числом элементов
template <class Type>
Set <Type>::Set(int size)
{
    SetPtr = new Type[size];             // выделим память
    if(!SetPtr){                         // не удалось
        cout << "Нет памяти\n"; exit(1);
    }
    NumMembers = 0; MaxSize = size;
}
```

Поиск элемента. Приведём подпрограмму **find** поиска элемента и тест на принадлежность элемента множеству:

```
// закрытый член класса, обеспечивающий поиск элемента в множестве
template <class Type>
int Set <Type>::find(Type member)
{
    int i;
    for (i = 0; i < NumMembers; i++)    // поиск во всем множестве
        if(SetPtr[i] == member) return i;
    return -1; // если такого элемента нет
}
```

```
// закрытый член класса, дающий ответ на вопрос:
// принадлежит ли переданное ему значение множеству
template <class Stype>
int Set <Stype>::ismember(Stype member)
{
    if (find(member) != -1) return 1;    // произведём поиск
    else return 0;                      // не нашли
}
```

Функция **find()** возвращает индекс указанного элемента, если этот элемент принадлежит множеству. В противном случае она возвращает -1 .

Добавление и удаление элементов. Приведём подпрограмму добавления (**insert()**) и удаления (**remove()**) элементов множества.

```
// закрытый член класса, обеспечивающий добавление элемента во множество
template <class Stype>
void Set<Stype>::insert(Stype member) // добавление
{
    if(NumMembers == MaxSize) // проверим не переполнено ли множество
        { cout << "\nПереполнение множества"; exit(1); }
    if(!ismember(member))    // если нет такого элемента
    {
        SetPtr[NumMembers] = member;    // добавить
        NumMembers++;                  // элементов стало на один больше
    }
}
```

Аргументом этой подпрограммы служит новый элемент множества. Для удаления будем использовать закрытую функцию **remove()**:

```
// закрытый член класса, обеспечивающий удаление заданного элемента множества
template <class Stype>
void Set<Stype>::remove(Stype member)
{
    int loc = find(member); // найдём элемент множества
    if(loc != -1)           // если элемент найден
    {
        for(loc=0; loc < NumMembers -1; loc++)
            SetPtr[loc] = SetPtr[loc+1]; // сдвигаем множество
        NumMembers--;                  // элементов на один стало меньше
    }
}
```

Если такой элемент множества существует, то он удаляется путем сдвига элементов массива на одну позицию влево.

Конструктор копирования. Конструктор копирования класса вызывается каждый раз, когда выполняется копирование объектов, принадлежащих этому классу. В частности, он вызывается, когда объект передается функции по значению, при построении временного объекта как возвращаемого значения функции, а также при

использовании объекта для инициализации другого объекта. Если класс не содержит конструктора копирования, определенного явным образом, то в этом случае при возникновении одной из трех перечисленных выше ситуаций используется *побитовое копирование* объекта. Побитовое копирование не во всех случаях бывает адекватным. Именно для таких случаев необходимо определять собственный конструктор копирования.

Необходимость использования этого конструктора для класса **Set** вызвана тем, что область памяти для каждого множества выделяется с помощью операции **new**, а указатель на эту область памяти хранится в указателе **SetPtr**. При побитовом копировании указатель, содержащийся в переменной **SetPtr** копии, будет показывать на ту же область памяти, что и указатель **SetPtr** оригинала. Оба объекта для хранения будут показывать на одну и ту же область памяти. Поэтому изменение одного из объектов может повлечь за собой изменение другого. Если один из объектов будет удален, то область памяти будет освобождена, а эта же область памяти используется и другим объектом. Такие ситуации, как правило, приводят к аварийному завершению программы.

Для того чтобы этого избежать, выделим различные области памяти для копии и оригинала:

```
// Конструктор копирования
template <class Stype>
Set<Stype>::Set(const Set<Stype> &ob)
{
    int i;
    MaxSize = ob.MaxSize;
    SetPtr = new Stype[MaxSize];          // выделим память
    if(!SetPtr)                          // если не удалось
    {
        cout << "\nНет памяти для копирования";
        exit(1);
    }
    NumMembers = 0;
    for(i=0; i < ob.NumMembers; i++)
        insert(ob.SetPtr[i]);           // производим копирование
}
```

Конструктор копирования выделяет область памяти для копии, а затем копирует в новый объект каждый элемент исходного объекта. Указатели **SetPtr** исходного объекта и копии будут указывать на разные области памяти.

Операция присваивания. Если не определять операцию присваивания, то она будет осуществляться с помощью побитового копирования. В результате может получиться два объекта, использующих одну и ту же область памяти для хранения данных.

Во избежание этой ситуации операция присваивания должна просто переписать содержимое одного множества в другое, не изменяя при этом значения **SetPtr** каждого из этих множеств:

```
// операция присваивания
template <class Stype>
```

```

Set<Stype> &Set<Stype>::operator = (Set<Stype> &ob)
{
    int i;
    // обработка случая s = s
    if(SetPtr == ob.SetPtr) return *this;
    // проверяем число элементов
    if(ob.NumMembers > MaxSize)
    {
        delete SetPtr;           // сначала удалим множество
        SetPtr = new Stype[ob.NumMembers]; // затем выделим память
                                   // под новое множество
        if(!SetPtr)               // если нет памяти
        {
            cout << "\nНет памяти для копирования"; exit(1);
        }
        MaxSize = ob.NumMembers;
    }
    NumMembers = 0;               // удаляем старое множество
    for (i = 0; i < ob.NumMembers; i++)
        insert(ob.SetPtr[i]);     // производим копирование всех элементов
    return *this;                 // возврат указателя на текущий экземпляр
    // класса
}

```

Добавление элемента и построение объединения. Перегрузим операцию сложения для двух случаев. В первом случае эта операция обрабатывает ситуацию «множество плюс элемент».

```

//Операция добавления нового элемента в множество
template <class Stype>
Set<Stype> Set<Stype>::operator+(Stype member)
{
    int i;
    Set<Stype> temp(NumMembers+1);
    // копирование элементов во временное множество
    for(i = 0; i < NumMembers; i++)
        temp.insert(SetPtr[i]);
        temp.insert(member);
    return temp; // возврат нового множества
}

```

Во втором случае эта операция обрабатывает ситуацию «элемент плюс множество». Она определяется с помощью дружественной функции:

```

//Ещё одна сигнатура операции добавления
template <class Stype>
Set<Stype> operator+(Stype member, Set<Stype> ob)
{

```

```

    int i;
    Set<Stype> temp(ob.NumMembers + 1);
    // копирование элементов во временное множество
    for(i = 0; i < ob.NumMembers; i++)
        temp.insert(ob.SetPtr[i]);
    // вставка нового элемента
    temp.insert(member);
    return temp; // возврат нового множества
}

```

Перегрузим операцию «+» для объединения множеств:

```

// операция объединения двух множеств
template <class Stype>
Set<Stype> Set<Stype>::operator+(Set<Stype> &ob)
{
    int i;
    Set<Stype> temp(NumMembers+ob.NumMembers);
    for(i = 0; i < NumMembers; i++)
        temp.insert(SetPtr[i]); // во временное множество копируем
// сначала первое множество
    for(i = 0; i < ob.NumMembers; i++)
        temp.insert(ob.SetPtr[i]); // а затем второе
    return temp; // возврат нового множества
}

```

Эта операция используется для выполнения операторов следующего типа:

```
set1 = set2 + set3;
```

где **set1**, **set2**, **set3** — объекты класса **set**.

Удаление элемента и разность множеств. Для удаления элемента определим операцию «-»:

```

// операция удаления элемента из множества
template <class Stype>
Set<Stype> Set<Stype>::operator-(Stype member)
{
    int i;
    Set<Stype> temp = *this;
    temp.remove(member); // удаление элемента
    return temp; // возврат множества
}

```

Эта функция позволяет вычислять выражения **set1 = set2 - item**, где **set1** и **set2** — объекты класса **set**, а **item** — элемент из **set2**.

Перегрузим операцию вычитания для вычисления разности множеств:

```

// операция разности двух множеств
template <class Stype>

```

```

Set<Stype> Set<Stype>::operator-(Set<Stype> &ob)
{
    int i;
    Set<Stype> temp = *this;
    // удаляем элементы из *this, принадлежащие ob
    for(i = 0; i < NumMembers; i++)
        if(ob.ismember(SetPtr[i]))
            temp.remove(SetPtr[i]);
    return temp; // возврат результата
}

```

Например, после выполнения оператора **set1 = set2 - set3**, множество **set1** будет состоять из элементов **set2**, не принадлежащих **set3**.

Пересечение множеств. Для обозначения пересечения будем использовать знак конъюнкции:

```

//Операция пересечения множеств
template <class Stype>
Set<Stype> Set<Stype>::operator& (Set<Stype> &ob)
{
    int i, j;
    Set<Stype> temp(NumMembers);
    for(i = 0; i < NumMembers; i++)
        if(ob.ismember(SetPtr[i]))
            temp.insert(SetPtr[i]);    // вставляем в результат только
// те элементы, которые принадлежат и первому множеству, и второму
    return temp; // возврат результата
}

```

После выполнения операции **set1 = set2 & set3** множество **set1** будет содержать элементы из **set2**, одновременно принадлежащие **set3**.

Сравнение множеств. Равенство и неравенство для класса **Set** реализованы перегрузкой операций «==» и «!=»:

```

// 1 — если множества равны
template <class Stype>
int Set<Stype>::operator == (Set<Stype> &ob)
{
    if(NumMembers != ob.NumMembers) return 0;
    // множества должны содержать одинаковое число элементов
    return *this < ob; // если первое содержится во втором, то равны
}
// проверка на неравенство
template <class Stype>
int Set<Stype>::operator !=(Set<Stype> &ob)
{ return !(*this == ob); }
// проверка на включение
template <class Stype>

```

```
int Set<Stype>::operator < (Set<Stype> &ob)
{
    int i;
    for(i = 0; i < NumMembers; i++)
        if(!ob.ismember(SetPtr[i])) return 0;
        // если SetPtr[i] не принадлежит ob
    return 1;
}
```

Проверка принадлежности. Операцию «<» перегрузим для определения принадлежности элемента множеству:

```
// 1 — если принадлежит
template <class Stype>
int operator < (Stype member, Set<Stype> ob)
{
    if ( ob.ismember(member) ) return 1; // если есть такой элемент
    return 0;
}
```

Преобразование в целое. Преобразование объекта класса **set** в целое число возвращает число, равное количеству элементов, содержащихся в множестве на текущий момент. Если множество пусто, то возвращается нуль. Функция преобразования нужна для автоматического преобразования к другому, обычно встроенному, типу. Ее текст определен как **inline**-функция:

```
operator int() return NumMembers;
```

Эта функция позволяет выполнять действия, подобные приведенным ниже:

```
if (set) cout << "Множество не пустое";
cout << "set1 содержит" << (int) set1 << "\n элементов"
```

Перегрузка операторов ввода-вывода. Определим операции ввода и вывода с помощью «>>» и «<<», как дружественные функции:

```
// ВВОД
template <class Stype>
istream& operator >>(istream& stream, Set <Stype> &ob)
{
    Stype member;
    stream >> member; // ввод элемента
    ob = ob + member; // запись элемента в множество
    return stream; // возврат результата
}
// ВЫВОД
template <class Stype>
ostream &operator << (ostream &stream, Set<Stype> &ob)
{
    int i;
```

```

for(i = 0; i < ob.NumMembers; i++) // для всех элементов
stream << ob.SetPtr[i] << ' '; // вывод
stream << endl; // после вывода всех элементов перевод строки
return stream;
}

```



Пример

Приведём *пример* программы, использующей параметризованный класс множества:

```

#include <iostream.h>           // библиотека потокового ввода-вывода
#include <conio.h>              // библиотека консольного ввода-вывода
#include <process.h>            // необходимо для функции exit
const int DEFSET = 100;
template <class Stype>
class Set
{
    Stype *SetPtr;             // указатель на первый элемент
    int MaxSize;               // максимальное число элементов
    int NumMembers;            // количество элементов множества
    void insert (Stype member); // добавление элемента
    void remove (Stype member); // удаление элемента
    int find(Stype member);    // поиск элемента
    int ismember (Stype member); // принадлежность элемента
public:
    Set();                     // конструкторы
    Set(int size);
    Set(const Set &ob);        // конструктор копирования
    ~Set() { delete SetPtr; }  // деструктор
    Set <Stype> &operator = (Set <Stype> &ob); // присваивание
    Set <Stype> operator + (Stype member);      // добавление элемента
    friend Set<Stype> operator + (Stype member, Set <Stype> ob);
    Set <Stype> operator + (Set <Stype> &ob); // объединение
    Set <Stype> operator - (Stype member);      // удаление элемента
    Set <Stype> operator - (Set <Stype> &ob); // разность
    Set <Stype> operator & (Set <Stype> &ob); // пересечение
    // операции сравнения
    int operator == (Set <Stype> &ob);          // 1 если равно
    int operator != (Set <Stype> &);            // 1 если не равно
    int operator < (Set <Stype> &ob);           // 1 если подмножество
    friend int operator < (Stype member, Set <Stype> ob);
                                           // 1 если элемент множества
    operator int() return NumMembers; // преобразование в целое
    // ВВОД-ВЫВОД

```

```

    friend istream &operator >> (istream &stream, Set<Stype> &ob);
    friend ostream &operator << (ostream &stream, Set<Stype> & ob);
};
// параметризованный конструктор класса, вызываемый по умолчанию
template <class Stype>
Set <Stype>::Set()
{
    SetPtr = new Stype [DEFSET]; // выделим память
    if(!SetPtr){ cout << "Нет памяти\n";
                exit(1);
            }
    NumMembers = 0; MaxSize = DEFSET;
}
// параметризованный конструктор с заданным числом элементов
template <class Stype>
Set <Stype>::Set(int size)
{
    SetPtr = new Stype[size];      // выделим память
    if(!SetPtr){                  // не удалось
        cout << "Нет памяти\n"; exit(1);
    }
    NumMembers = 0; MaxSize = size;
}
// закрытый член класса, обеспечивающий поиск элемента в множестве
template <class Stype>
int Set <Stype>::find(Stype member)
{
    int i;
    for (i = 0; i < NumMembers; i++)    // поиск во всем множестве
        if(SetPtr[i] == member) return i;
    return -1; // если такого элемента нет
}
// закрытый член класса, дающий ответ на вопрос:
// принадлежит ли переданное ему значение множеству
template <class Stype>
int Set <Stype>::ismember(Stype member)
{
    if (find(member) != -1) return 1;    // произведём поиск
    else return 0;                      // не нашли
}
// закрытый член класса, обеспечивающий добавление элемента в множество
template <class Stype>
void Set<Stype>::insert(Stype member) // добавление
{
    if(NumMembers == MaxSize)    // проверим, не переполнено ли множество
    { cout << "\nПереполнение множества"; exit(1); }
}

```



```

        if(!SetPtr)           // если нет памяти
        { cout << "\nНет памяти для копирования"; exit(1); }
        MaxSize = ob.NumMembers;
    }
    NumMembers = 0;           // удаляем старое множество
    for (i = 0; i < ob.NumMembers; i++)
        insert(ob.SetPtr[i]); // производим копирование всех элементов
    return *this;             // возврат указателя на текущий экземпляр
    // класса
}
//Операция добавления нового элемента в множество
template <class Stype>
Set<Stype> Set<Stype>::operator+(Stype member)
{
    int i;
    Set<Stype> temp(NumMembers+1);
    // копирование элементов во временное множество
    for(i = 0; i < NumMembers; i++)
        temp.insert(SetPtr[i]);
    temp.insert(member);
    return temp; // возврат нового множества
}
//Ещё одна сигнатура операции добавления
template <class Stype>
Set<Stype> operator+(Stype member, Set<Stype> ob)
{
    int i;
    Set<Stype> temp(ob.NumMembers + 1);
    // копирование элементов во временное множество
    for(i = 0; i < ob.NumMembers; i++)
        temp.insert(ob.SetPtr[i]);
    // вставка нового элемента
    temp.insert(member);
    return temp; // возврат нового множества
}
// операция объединения двух множеств
template <class Stype>
Set<Stype> Set<Stype>::operator+(Set<Stype> &ob)
{
    int i;
    Set<Stype> temp(NumMembers+ob.NumMembers);
    for(i = 0; i < NumMembers; i++)
        temp.insert(SetPtr[i]); // во временное множество копируем
    // сначала первое множество
    for(i = 0; i < ob.NumMembers; i++)
        temp.insert(ob.SetPtr[i]); // а затем второе
}

```

```

        return temp;                // возврат нового множества
    }
    // операция удаления элемента из множества
    template <class Stype>
    Set<Stype> Set<Stype>::operator-(Stype member)
    {
        int i;
        Set<Stype> temp = *this;
        temp.remove(member); // удаление элемента
        return temp; // возврат множества
    }
    // операция разности двух множеств
    template <class Stype>
    Set<Stype> Set<Stype>::operator-(Set<Stype> &ob)
    {
        int i;
        Set<Stype> temp = *this;
        // удаляем элементы из *this, принадлежащие ob
        for(i = 0; i < NumMembers; i++)
            if(ob.ismember(SetPtr[i]))
                temp.remove(SetPtr[i]);
        return temp; // возврат результата
    }
    //Операция пересечения множеств
    template <class Stype>
    Set<Stype> Set<Stype>::operator& (Set<Stype> &ob)
    {
        int i, j;
        Set<Stype> temp(NumMembers);
        for(i = 0; i < NumMembers; i++)
            if(ob.ismember(SetPtr[i]))
                temp.insert(SetPtr[i]); // вставляем в результат только
        // те элементы, которые принадлежат и первому множеству, и второму
        return temp; // возврат результата
    }
    // 1 — если множества равны
    template <class Stype>
    int Set<Stype>::operator == (Set<Stype> &ob)
    {
        if(NumMembers != ob.NumMembers) return 0;
        // множества должны содержать одинаковое число элементов
        return *this < ob; // если первое содержится во втором, то равны
    }
    // проверка на неравенство
    template <class Stype>
    int Set<Stype>::operator !=(Set<Stype> &ob)

```

```

{ return !(*this == ob); }
// проверка на включение
template <class Stype>
int Set<Stype>::operator < (Set<Stype> &ob)
{
    int i;
    for(i = 0; i < NumMembers; i++)
        if(!ob.ismember(SetPtr[i])) return 0;
        // если SetPtr[i] не принадлежит ob
    return 1;
}
// 1 — если принадлежит
template <class Stype>
int operator < (Stype member, Set<Stype> ob)
{
    if ( ob.ismember(member) ) return 1; // если есть такой элемент
    return 0;
}
// ВВОД
template <class Stype>
istream& operator >>(istream& stream, Set <Stype> &ob)
{
    Stype member;
    stream >> member; // ввод элемента
    ob = ob + member; // запись элемента в множество
    return stream;    // возврат результата
}
// ВЫВОД
template <class Stype>
ostream &operator << (ostream &stream, Set<Stype> &ob)
{
    int i;
    for(i = 0; i < ob.NumMembers; i++)    // для всех элементов
        stream << ob.SetPtr[i] << ' ';    // вывод
    stream << endl;    // после вывода всех элементов перевод строки
    return stream;
}
void main (void)
{
    Set <char> a(5);
        Set <char> b(5);
        Set <char> d(5);
        Set <char> temp(5);

    clrscr();
    cout << "Введите первое множество :\n";
    cin >> a;

```

```

    cin >> a;
    cin >> a;
    cin >> a;
    cin >> a;
    cout << "Введите второе множество :\n";
    cin >> b;
    cin >> b;
    cin >> b; cin >> b;
    cin >> b;
    cout << "Первое множество:"<<a;
    cout << "Количество его элементов: "<<int(a)<<"\n";
    cout << "Второе множество:"<<b;
    cout << "Объединение множеств: "<<a+b;
    cout << "Разность множеств: "<<a-b;
    temp=a;
    d=a&b;
    cout << "Пересечение множеств: "<<d;
    temp=temp-'a';
    cout << "Первое множество после удаления элемента 'a': "<<temp;
    cout << "Проверка на принадлежность элемента 'f' второму множеству:\n";
    if (b<'f')
        { cout <<"Элемент принадлежит множеству\n"; }
    else
        { cout <<"Элемент не принадлежит множеству\n"; }
    cout << "Проверка на равенство двух данных множеств:\n";
    temp=temp+'a';
    if (b==temp)
        { cout <<"Множества равны\n"; }
    else
        { cout <<"Множества не равны\n"; }
    return;
}

```

Результат работы программы

Введите первое множество :

a
b
c
d
e

Введите второе множество :

e
f
g
h
i

Первое множество: a b c d e

Количество его элементов: 5

Второе множество: e f g h i

Объединение множеств: a b c d e f g h i

Разность множеств: a b c d

Пересечение множеств: e

Первое множество после удаления элемента 'a': b c d e

Проверка на принадлежность элемента 'f' второму множеству:

Элемент принадлежит множеству

Проверка на равенство двух данных множеств:

Множества не равны

.....



Контрольные вопросы по главе 3

.....

- 1) Что такое контейнерные классы?
- 2) Что такое контейнер?
- 3) Что такое шаблоны классов?
- 4) Какие вы знаете правила определения шаблонов классов?
- 5) Можно ли использовать шаблоны классов в структурах или объединениях?
- 6) Что такое параметризованные очереди и стеки?
- 7) Что называется циклической очередью?
- 8) Что такое бинарные деревья?

Глава 4

ПРОИЗВОДНЫЕ КЛАССЫ



.....
*Класс, полученный из другого класса добавлением новых полей и составных функций, называется **производным классом**.*
.....



.....
***Базовым** называется класс, из которого получается производный класс.*
.....

Производный класс может быть сформирован на основе нескольких базовых классов. В этом случае говорят о *множественном наследовании*. При наследовании важную роль играет статус доступа к компонентам класса [16–18].

4.1 Определение производного класса

Объект производного класса должен быть частным случаем объекта базового класса. Поэтому для его определения нужна дополнительная информация. Например, прямоугольник, стороны которого параллельны осям координат, можно задать координатами левого верхнего угла, шириной и высотой. Произвольный четырёхугольник можно задать структурой следующего вида:

```
Struct poly4  
{ float x[4], y[4]; }
```

Следовательно, прямоугольник задается как структура:

```
Struct rect  
{  
    float x[4], y[4]; // координаты углов
```

```
float a, b; // ширина и высота
}
```

Конструктор прямоугольника можно определить как подпрограмму, устанавливающую координаты $x[0]$, $x[1]$, $x[2]$, $x[3]$ и $y[0]$, $y[1]$, $y[2]$, $y[3]$. Для того чтобы эти два класса выстроить в иерархию, класс прямоугольника определяется следующим образом:

```
Struct rect : poly4
{ float a,b; }
```

Такое определение равносильно перечислению всех полей базового класса. В некоторых случаях оно обладает преимуществами, позволяющими, в частности, создавать не только иерархические деревья, но и более сложные иерархические структуры. Формат определения производного класса:

$$\text{Class имя} : \left\{ \begin{array}{l} \text{private} \\ \text{public} \end{array} \right\} \text{ имя_1, имя_2, \dots, имя_N}$$

```
{
  дополнительные поля и функции производного класса
}
```

Если атрибут **private** или **public** не указан, то будет установлен **private**. При создании объекта производного класса сначала вызываются конструкторы базовых классов, а потом — конструктор производного. Деструкторы вызываются в обратном порядке. Вместо ключевого слова **class** можно применять **struct**.

4.2 Доступ к полям и функциям базового класса



.....
 Функция или поле базового класса могут быть определены заново в производном классе. Такой член базового класса называется **переопределенным**.

При обращении к переопределенной функции будет вызвана ближайшая по иерархии функция. Если в этом случае требуется вызов функции из базового класса или вызов внешней функции, то применяется *оператор разрешения области видимости ::*, например:

```
#include <iostream.h> // библиотека потокового ввода-вывода
void f()
{ cout << "\nВнешняя функция" }
struct Base1 // первый класс
{ void f() { cout << "\nФункция из Base1"; }
};
struct Base2 // второй класс
{ void f() { cout << "\nФункция из Base2;"} };
```

```

struct Deriv: Base1, Base2 // класс, производный от двух данных классов
{ void f() {::f();} } // вызов внешней функции
int main()
{
    Deriv obj; // создаём объект класса Deriv
    f();
    obj.Base1::f();
    obj.Base2::f();
    obj.f();
    return 0;
}

```

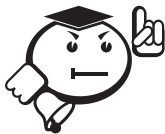
В результате работы программы будут выведены следующие строки:

```

Внешняя функция
Функция из Base1
Функция из Base2
Внешняя функция

```

В последнем случае будет вызвана функция **f()** объекта производного класса, которая, в свою очередь, вызовет внешнюю функцию.



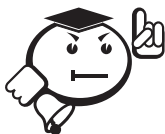
.....
 При наследовании статус доступа к полям и собственным функциям базового класса изменяется следующим образом: если перед базовым классом указан атрибут **private**, то:

открытые члены базового класса переходят в закрытые в производном классе,

защищенные — в закрытые,

закрытые — в недоступные.

.....



.....
 Если перед базовым классом используется атрибут **public**, то:

открытые члены базового класса переходят в открытые в производном классе,

защищенные — в защищенные,

закрытые — в недоступные.

.....

Более подробно эти преобразования приведены в табл. 4.1.

В таблице отражено преобразование статуса доступа как для классов, определенных с помощью ключевого слова **struct**, так и для классов, заданных с помощью **class**. Например:

```

Class A: protected B { };
Struct A: B { };

```

Таблица 4.1 – Таблица преобразований типов при наследовании

Доступ в базовом классе	Атрибут доступа перед базовым классом	Доступ в производном классе	
		struct	class
public	отсутствует	public	private
protected	отсутствует	public	private
private	отсутствует	недоступны	недоступны
public	public	public	public
protected	public	protected	protected
private	public	недоступны	недоступны
public	protected	protected	protected
protected	protected	protected	protected
private	protected	недоступны	недоступны
public	private	private	private
protected	private	private	private
private	private	недоступны	недоступны

4.3 Класс дерева поиска

Произвольный список определяется адресом одного из узлов. Для двоичного дерева этот узел будет *корневым*, для циклического списка удобно использовать *последний* узел. Следовательно, список можно задать с помощью класса

```
class LIST
{
protected:
    NODE *root;
public:
    LIST() { root = NULL; }
};
```

В этом примере структура основного узла имеет статус доступа **protected**, ибо поля со статусом доступа **private** для производных классов становятся недоступными. Здесь учтено, что пустой список, как правило, содержит указатель, равный нулю. Остальные списки можно определить как производные от класса **LIST**. Определим класс дерева как производный от класса **LIST**:

```
class TREE : LIST
{
public:
    void insert (int x);
    void show();
}
```

В этом примере функция **insert()** — служит для добавления нового элемента к дереву, а подпрограмма **show()** выводит содержимое дерева в симметричном порядке. Аналогичным образом можно дополнить эти составные функции подпро-

граммой удаления элемента. Возможность доступа к внешним функциям позволяет определить подпрограмму включения элемента следующим образом:

```
void TREE :: insert (int x)
{ root = :: insert (root, x); };
```

Аналогичным образом определяется подпрограмма вывода элементов в симметричном порядке. При выводе элементы дерева отображаются в неубывающем порядке.

Ниже приведён текст программы, реализующей класс дерева поиска:

```
#include <stdio.h>          // стандартная библиотека ввода-вывода
#include <conio.h>          // библиотека консольного ввода-вывода
struct NODE                // структура узла дерева
{
    int info;               // информационное поле
    NODE *left, *right;    // указатели на левое и правое поддеревья
};
class LIST                 // базовый класс списка
{
protected:
    NODE *root;
public:
    LIST() { root = NULL;}
};
class TREE: LIST           // класс дерева поиска, производный от класса списка
{
public:
    void insert(int x);     // добавление элемента
    void show();           // обход в симметричном порядке
};
NODE* insert(NODE* root, int x)
{
    if (!root)              // если дерево пусто, то
    {
        root = new NODE;    // создаём новое дерево
        root -> info = x;    // заполняем информационную часть
        root -> left = root -> right = NULL;
    }
    else
    {
        // дерево не пусто если значение добавляемого элемента меньше чем значение
        // информационной части корня, то его следует добавлять в левое поддерево
        if (x < root -> info) root -> left = insert(root -> left, x);
        // в противном случае его следует добавлять в правое поддерево
        else root -> right = insert(root -> right, x);
    }
    return root;
};
```

```
void TREE :: insert(int x)
{ root = ::insert(root, x); }; void display(NODE* p)
{
    if(p)
    {
        display(p -> left);      // переходим в левое поддерево
        printf("\n%d", p -> info); // отображаем содержимое информационного поля
        display(p -> right);     // переходим в правое поддерево
    } };
void TREE :: show()
{
    display(root);
};
int main()
{
    TREE a;          // создадим объект класса дерево
    clrscr();        // очистим экран
    // добавим в дерево произвольные элементы
    a.insert(2);
    a.insert(3);
    a.insert(1);
    a.insert(12);
    a.insert(21);
    a.insert(14);
    a.insert(20);
    a.insert(3);
    printf ("Обход дерева в симметричном порядке:");
    a.show();        // отобразим дерево на экране
    getch();         // ожидание нажатия любой клавиши (пауза)
    return 0;
}
```

Результаты работы программы

Обход дерева в симметричном порядке:

1
2
3
3
12
14
20
21

4.4 Параметризованный связный список



.....
Связным списком называется последовательность элементов данных, связанных ссылками.

Связные списки могут иметь *одиночные* или *двойные* связи. В списке с одиночными связями каждый элемент содержит ссылку на следующий элемент данных. В списке с двойными связями каждый элемент содержит ссылки на предшествующий и последующий элементы. Хотя списки с одиночными связями встречаются достаточно часто, но списки с двойными связями распространены наиболее широко. Основную роль в этом играют следующие три фактора:

- список с двойными связями можно читать в обоих направлениях: и от начала к концу, и от конца к началу. Список с одиночными связями можно читать только в одном направлении;
- поврежденный список с двойными связями проще перестраивать, так как с каждым из членов списка ассоциированы две ссылки;
- некоторые типы операций над списками (например, удаление) проще выполняются над списками с двойными связями.

Рассмотрим метод построения параметризованного списка с двойными связями. Список организовывается с помощью двух классов, первый из которых, **listob**, определяет природу элементов списка, а второй, **dlist**, реализует механизм списка с двойными связями. Первый из этих классов определяется следующим образом:

```
template <class DataT>
class listob // класс элемента списка
{
    public:
        DataT info;      // информационная часть
        listob <DataT> *next; // указатель на следующий элемент
        listob <DataT> *prior; // указатель на предшествующий элемент
        listob() // конструктор
        { info = 0; next = prior = NULL; };
        listob (DataT c) // конструктор
        { info = c; next = prior = NULL; };
        listob <DataT> *getnext() { return next; }
        listob <DataT> *getprior() { return prior; }
        void getinfo (DataT& c) { c = info; }
        void change (DataT c) { info = c; } // изменение элемента
        friend ostream &operator << (ostream &stream, listob <DataT> o);
        friend ostream &operator << (ostream &stream, listob <DataT> *o);
        friend istream &operator >> (istream &stream, listob <DataT> &o);
};
// перегрузка операции << для объекта listob
```

```

template <class DataT>
ostream &operator << (ostream &stream, listob <DataT> o)
{ stream << o.info << endl; return stream; }
template <class DataT>
ostream &operator << (ostream &stream, listob <DataT> *o)
{ stream << o -> info << endl; return stream; }
// Перегрузка операции >>
template <class DataT>
istream &operator >> (istream &stream, listob <DataT> &o)
{
    cout << "Введите информацию: ";
    stream << o.info; return stream;
}

```

Оператор << перегружается как для объектов типа **listob**, так и для указателей на объекты этого типа. Это связано с тем, что при использовании связных списков широко распространена практика получения доступа к элементам списка через указатель. Поэтому оператор << полезно перегружать, с тем чтобы он мог оперировать с переданным ему указателем на объект.

Механизм построения связного списка реализуется классом, приведенным ниже. Этот класс является производным от класса **listob** и оперирует с объектами класса **listob**.

```

template <class DataT> // параметризованный класс объекта списка
class dlist: public listob<DataT>
// класс списка — производный от класса узла
{
    listob<DataT> *start, *end;
// указатели на первый и последний элементы
public:
    dlist(){start=end=NULL;} // конструктор
    ~dlist(); // деструктор
    void store(DataT c); // запись в список
    void remove(listob<DataT> *ob); // удаление элемента
    void frwdlist(); // чтение в прямом направлении
    void bkwdlist(); // чтение в обратном направлении
    listob<DataT> *find(DataT c); // поиск
    listob<DataT> *getstart(){return start;} // начало поиска
    listob<DataT> *getend(){return end;} // указатель на конец списка
}

```

Поскольку каждый узел списка содержит указатели на следующий и предшествующий узлы, то список является *двухсвязным*. Объектами класса **dlist** будут двухсвязные списки. Каждый объект этого класса содержит указатель на начало и указатель на конец списка. Оба эти указателя являются указателями на объекты класса **listob**. При создании списка оба указателя инициализируются значением **NULL**. Класс **dlist** поддерживает целый ряд операций над двухсвязными списками, в том числе:

- ввод нового элемента в список;
- удаление элемента из списка;
- просмотр списка в любом направлении (от начала к концу или от конца к началу);
- поиск элемента в списке;
- получение указателей на начало и на конец списка.

Разработаем подпрограммы, выполняющие эти операции и тестовую программу.

```
// dlist.cpp — parametrised class of the double connected list
#include <conio.h>
#include <iostream.h>
#include <stdlib.h>
template <class DataT> class listob;
template <class DataT>
ostream &operator << (operator &stream, listob<DataT> o)
{
    stream<<o.info<<endl; // вывод объекта
    return stream;
}
/* template <class DataT>
ostream &operator<<(ostream &stream, listob<DataT> *o)
{
    stream<<o->info<<endl; // вывод объекта по указателю
    return stream;
}*/
/* template <class DataT>
istream &operator>>(istream &stream, listob<DataT> &o)
{
    cout<<"Input data: ";
    stream>>o.info; // ввод объекта
    return stream;
}
*/
template <class DataT> class listob // класс узла
{
public:
    DataT info;          // информационная часть
    Listob<DataT> *next, // указатель на следующий элемент
                *prior;  // указатель на предшествующий элемент
    listob()
    { info=0; next = NULL; prior=NULL; } // конструктор
    listob(DataT c)
    { info=c; next=NULL; prior=NULL;} // конструктор
    listob<DataT> *getnext(){return next;}
    // чтение адреса следующего элемента
```

```

listob<DataT> *getprior(){return prior;}
// чтение адреса предшествующего элемента
void getinfo(DataT &c){c=info;} // чтение информации в аргумент
void change(DataT c){info=c;} // изменение информации
friend ostream &operator<<(ostream &stream, listob<DataT> o);
// дружественные функции
//friend ostream &operator<<(ostream &stream, listob<DataT> *o);
// ввода-вывода
//friend istream &operator>>(istream &stream, listob<DataT> &o);
};
template <class DataT> // параметризованный класс объекта списка
class dlist: public listob<DataT>
// класс списка — производный от класса узла
{
    listob<DataT> *start, *end;
// указатели на первый и последний элементы
public:
    dlist(){start=end=NULL;} // конструктор
    ~dlist(); // деструктор
    void store(DataT c); // запись в список
    void remove(listob<DataT> *ob); // удаление элемента
    void frwdlist(); // чтение в прямом направлении
    void bkwdlist(); // чтение в обратном направлении
    listob<DataT> *find(DataT c); // поиск
    listob<DataT> *getstart(){return start;} // начало поиска
    listob<DataT> *getend(){return end;} // указатель на конец списка
}
template <class DataT> dlist<DataT>::~~dlist
{
    listob<DataT> *p, *p1; // деструктор
    p=start;
    while(p)
    { p1=p->next; delete p; p=p1; } // освобождение памяти, занятой элементами списка
}
template <class DataT> void dlist<DataT>::store(DataT c)
{
    listob<DataT> *p;
    p= new listob<DataT>; // запись нового элемента
    if(!p){cout<<"Error of memory allocation\n"; exit(1);}
    p->info=c;
    if(start==NULL)
// если список пуст, то создается список, состоящий из одного элемента
    { end=start=p; }
    else // иначе изменяем значения указателей
    { p->prior=end; end->next=p; end=p; }
}

```

```

template <class DataT>
void dlist<DataT>::remove(listob<DataT> *ob)
// удаление элемента списка
{
    if(ob->prior)          // если не первый элемент
    {
        ob->prior->next=ob->next;
        if(ob->next)      // если не последний элемент
            ob->next->prior=ob->prior;
        else              // иначе удаляется последний
            end=ob->prior; // обновление указателя на конец списка
    }
else // удаляется первый элемент списка, если список не пуст
{ if(ob->next)
    {
        ob->next->prior = NULL;
        start=ob->next;
    }
else // иначе, т.е. если список пуст,
start=end=NULL; // установить начало и конец на 0
} }
template <class DataT>
void dlist<DataT>::frwdlist()
// вывод элементов списка в прямом направлении
{
    listob<DataT> *temp;
    temp=start;
    while(temp)
    {
        cout<<temp->info<< " ";
        temp = temp -> getnext();
    }
cout<<endl;
}
template <class DataT>
void dlist<DataT>::bkwdlist()
// вывод элементов списка в обратном направлении
{
    listob<DataT> *temp;
    temp=end;
    while(temp)
    {
        cout<<temp->info<< " ";
        temp = temp -> getprior();
    }
cout<<endl;
}

```

```

}
template <class DataT>
listob<DataT> *dlist<DataT>::find(DataT c)
// поиск объекта, содержащего информацию, совпадающую с указанной
{
    listob<DataT> *temp;
    temp=start;
    while(temp)
    {
        if(c==temp->info) return temp; // совпадение найдено
        temp = temp->getnext();
    }
    return NULL;          // совпадение не найдено
}
main()
{
    dlist<double> list;    // демонстрация списка элементов типа double
    double i;
    listob<double> *p;
    clrscr();
    list.store(1);        // запись элементов 1, 2, 3
    list.store(2);
    list.store(3);
    cout<<"\nDirect list";
    list.frwdlist();      // вывод в прямом направлении
    cout<<"\nreverse list";
    list.bkwdlist();      // вывод в обратном направлении
    cout<<endl;
    cout<<"Hand viewing of the list"; // ручной просмотр списка
    p=list.getstart();
    while(p)
    {
        p->getinfo(i); cout<<i<<" ";
        p=p->getnext(); // следующий элемент
    }
    cout<<endl<<endl;
    cout<<"find of 2\n";
    p=list.find(2);       // поиск элемента 2
    if(p)
    {
        p->getinfo(i);
        cout<<"we have find" <<i<<endl; // найден элемент i
    }
    cout<<endl;
    p->getinfo(i);
    cout<<"delete"<<i<<"\n";

```

```

list.remove(p);    // удаление элемента
cout<<"list after deleting";
list.fwddlist();   // список после удаления
cout<<endl;
cout<<"insert the new 4"; // запись элемента 4
list.store(4);
cout<<"\nlist after insert";
list.fwddlist();   // вывод в прямом направлении
cout<<endl;
p=list.find(1);    // поиск элемента 1
if(!p)
{ cout<<"Error. No such element\n"; return 1; } // если не найден, выйти
p->getinfo(i);     // чтение в i
cout<<"Change<<i<<"to 5\n"; // вывод значения i
p->change(5);      // изменение 1 на 5
cout<<"list after the change";
list.fwddlist();   // вывод в прямом направлении
cout<<"Reverse list";
list.bkwdlist();   // вывод в обратном направлении
cout<<endl;
getch();
return 0;
}

```

Результаты работы программы

Список в прямом направлении: 1 2 3
 Список в обратном направлении: 3 2 1
 Ручной просмотр списка: 1 2 3
 Поиск числа 2 в списке
 Число 2 было найдено
 Удаление числа 2 из списка
 Список после удаления: 1 3
 Запись нового элемента 4 в список
 Список после вставки нового элемента: 1 3 4
 Заменяем 1 на 5
 Список после замены: 5 3 4
 Просмотр полученного списка в обратном порядке: 4 3 5

4.5 Множественное наследование

Производный класс может иметь любое число базовых классов.



.....
*Использование двух или более классов называется **множественным наследованием**.*

При инициализации объекта производного класса сначала вызываются конструкторы базовых классов, в порядке их перечисления в объявлении производного класса, а потом — конструктор производного класса.



Пример

Пусть **INTEGER** — класс, объектами которого являются целые числа, **POSITIVE** — класс положительных целых чисел. Определим рациональную дробь как объект производного класса от этих двух классов. Пара, представляющая рациональную дробь, состоит из взаимно простых целых чисел.

```
#include <iostream.h>    // библиотека потокового ввода-вывода
#include <process.h>      // библиотека с прототипом функции exit
#include <conio.h>        // библиотека консольного ввода-вывода
class INTEGER            // класс целых чисел
{
public:
    long NUM;            // информационное поле
    INTEGER (long Val): NUM(Val) {} // конструктор
};
class POSITIVE          // класс положительных чисел
{
public:
    unsigned long Den;   // информационное поле
    POSITIVE(unsigned long d) : Den(d) // конструктор
    { if(d==0) {cout << "Ошибка"; exit(1);} } // ноль недопустим
};
class RATIONAL : public INTEGER, public POSITIVE
// класс дроби
{
    // дружественная функция вывода дроби в некоторый поток
    friend ostream &operator<<(ostream& stream, RATIONAL& o);
public:
    RATIONAL(long v, unsigned long u=1): INTEGER(v), POSITIVE(u)
    // конструктор
    {
        long w;
        if (v==0) {u=1; return;}
        if(v<0) {w = -v;}
        else
        { w=v; }
        // поскольку числитель и знаменатель должны быть взаимно простыми
        // числами, то следует найти наибольший общий делитель для числителя
        // и знаменателя
        while (w!=u)
```

```

    {
        if(w>u) w=w-u;
        if(u>w) u=u-w;
    }
    // и следует сократить дробь
    NUM = NUM/w;
    Den = Den/w;
}
};
ostream& operator<<(ostream& stream, RATIONAL& o)
{
    stream<<o.NUM<<"/"<<o.Den;
    return stream;
}
main()
{
    RATIONAL r1(10, 20), r2(-15, 10);
    clrscr();
    cout<<"Первая дробь (числитель равен 10, знаменатель равен 20): ";
    cout<<r1<<"\n";
    cout<<"Вторая дробь (числитель равен -15, знаменатель равен 10): ";
    cout<<r2<<"\n";
    getch();
}

```

Результаты работы программы

Первая дробь (числитель равен 10, знаменатель равен 20): 1/2
 Вторая дробь (числитель равен -15, знаменатель равен 10): -3/2

.....

В данном примере при инициализации объекта — рационального числа — сначала будет вызван конструктор **INTEGER**, затем — конструктор класса **POSITIVE**, затем — конструктор класса **RATIONAL**. Доступ к членам базовых классов, имеющих одинаковые имена, осуществляется через имена базовых классов, которым они принадлежат, при помощи операции разрешения доступа. *Например:*

```

Class A
{ public: void f(); };
class B
{ public: void f(); };
class C : public A, public B {};
main()

```

```

{
    C c;
    c.f();
    // ошибка — неизвестно, какая из функций вызывается A::f() или B::f()
    правильный вызов
}

```

Неоднозначность, возникающая при вызове функций, может быть преодолена с помощью переопределения функции в производном классе, *например*:

```

Class C : public A, public B
{ public: void f() {A::f();} }
int main()
{
    C c;
    c.f(); // правильный вызов функции A::f()
    c.B::f(); // правильный вызов функции B::f()
}

```

Базовые классы с одинаковым именем не могут присутствовать в определении производного класса. Например, если попытаться определить вектор, как пару точек

```

Class Point {int x,y;}
Class Vector : public Point, public Point {} // ошибка

```

то компилятор выведет сообщение об ошибке, поскольку для объекта

```
Vector v;
```

неясно, как начальную точку вектора отличить от конечной.

Для классов, порожденных от производных классов с общей базой, по умолчанию существует два экземпляра объекта общей базы. Это позволяет наследовать базовый класс любое количество раз, например определение:

```

Class Point {public: int x, y;}
Class Point2 : public Point {};
Class Vector : public Point, public Point2 {};

```

будет верным и будет задавать вектор как пару точек. Теперь обращение к начальной и конечной точкам вектора будет производиться с помощью оператора разрешения области видимости.

4.6 Виртуальные классы



.....
 Базовый класс называется **виртуальным**, если его поля не дублируются при неоднократном наследовании.

Виртуальный базовый класс объявляется при наследовании при определении производного класса следующим образом:

```
class имя_производного_класса:
virtual public имя_виртуального_базового_класса
{
тело производного класса;
}
```



Пример

На рис. 4.1 приведена иерархия производных класса четырёхугольника. Для того чтобы поля четырёхугольника не наследовались более одного раза, объявим его как виртуальный базовый класс. Ромб будем задавать с помощью центра, длин диагоналей и угла поворота вокруг первой диагонали.

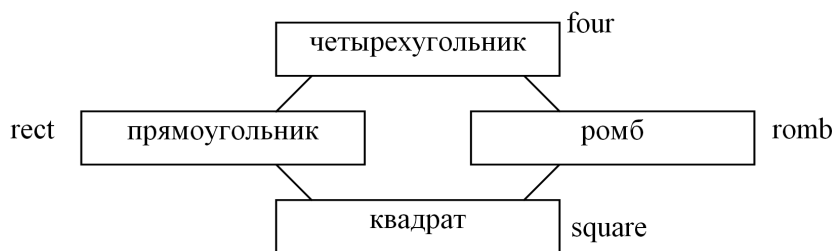


Рис. 4.1 – Иерархия подклассов четырёхугольника

Ниже для иллюстрации понятия виртуального класса приведём текст программы, в которой определена иерархия классов, отраженная на рис. 4.1. Класс квадрата дважды наследует координаты четырёх углов.

```
#include <graphics.h>
#include <math.h>
#include <conio.h>
class four          // четырёхугольник
{
protected:
    float x[4], y[4];    // координаты вершин
public:
    four(){}
    four(float *ix, float *iy) // конструктор
    {
        int i;
        for(i=0; i<4; i++)
        { x[i] = ix[i]; y[i] = iy[i]; }
    }
    ~four() {delete x; delete y;}    // деструктор
}
```

```

        void show();           // вывод четырёхугольника на экран
    };
    class rect: public virtual four           // прямоугольник
    {
    protected:
        int xleft, xright, ytop, ybottom;
    public:
        rect(int x1, int y1, int x2, int y2):
        xleft(x1), ytop(y1), xright(x2), ybottom(y2) // конструктор
        {
            x[0] = x1; y[0] = y1;
            x[1] = x1; y[1] = y2;
            x[2] = x2; y[2] = y2;
            x[3] = x2; y[3] = y1;
        }
    };
    // класс ромба
    class romb: public virtual four
    {
    protected:
        float xc, yc, alpha, a, b;
    public:
        romb(float x1, float y, float ugol, float d1, float d2)
        {
            xc = x1; yc = y; alpha = ugol; a = d1; b = d2;
            x[0] = xc + (a/2)*cos(alpha);
            x[0] = yc + (a/2)*sin(alpha);
            x[1] = xc - (b/2)*sin(alpha);
            x[1] = yc + (b/2)*cos(alpha);
            x[2] = xc - (a/2)*cos(alpha);
            x[2] = yc - (a/2)*sin(alpha);
            x[3] = xc + (b/2)*sin(alpha);
            x[3] = yc - (b/2)*cos(alpha);
        }
    };
    // стороны квадрата параллельны осям координат
    class square : public rect, public romb
    {
        int xcenter, ycenter; // центр квадрата
        int size;           // сторона квадрата
    public:
        square(int x0, int y0, int s): xcenter(x0+s/2), ycenter(y0+s/2),
        size(s), rect(x0 - s/2, y0 - s/2, x0 + s/2, y0 + s/2),
        romb(x0, y0, 3.14159/4, s, s) show();
    };
    void four :: show()      // вывод четырехугольника

```

```

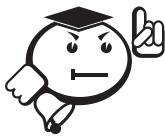
{
    int i;
    moveto (x[3], y[3]);
    for(i=0; i<4; i++)
        lineto(x[i], y[i]);
};
void main()
{
    int gd = DETECT, gm;
    initgraph (&gd, &gm, "c:\\prog\\bc31\\bgi");
    square q(320, 240, 100);
    getch(); // ожидание нажатия любой клавиши
}

```

В результате работы программы на экран будет выведен квадрат, сторона которого равна 100, а центр квадрата совпадает с центром экрана.

.....

Конструкторы и деструкторы виртуальных классов. Включение полей виртуального базового класса в производный класс осуществляется один раз, а их инициализация будет происходить в таком его производном классе, который не является его непосредственным наследником.



.....

Во время инициализации объекта некоторого класса **A** конструкторы классов, наследованных виртуально, активизируются перед конструкторами всех остальных базовых классов этого класса.

.....

Происходит это следующим образом:

- если в списке инициализации конструктора класса **A** используется инициализатор базового класса, наследованного виртуально, то активизируется конструктор этого базового класса;
- в противном случае конструктор виртуального базового класса инициализируется без параметров.

Деструкторы активизируются в обратном порядке.



Пример

Рассмотрим программу, работающую с иерархией классов, приведенных на рис. 4.2. Эта программа иллюстрирует порядок вызова конструкторов виртуальных классов.

Ниже приведён текст программы:

```

#include <stdio.h>      // стандартная библиотека ввода-вывода
// классы реализованы посредством конструкторов и информационных полей

```

```

class V
{
public:
    int a,b,c;
    V(): c(3);
    V(int p): a(p){};
};
class A: virtual public V
{
public:
    A():V(3) {a=1;}
};
class B: virtual public V
{
public:
    B() {b=2;}
};
class C: public A,B
{
public:
    // функция вывода
    void out(){printf("a=%d b=%d c=%d\n",a,b,c);}
};
int main()
{
    C ob1;
    ob1.out();
    return 0;
}

```

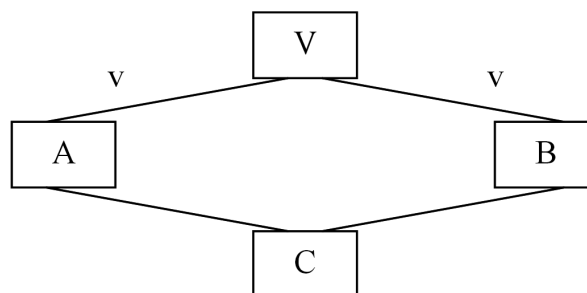


Рис. 4.2 – Иерархия классов

При создании объекта **ob1** класса **C** конструктор класса **C** вызовет конструктор **V()**, который установит **C=3**, затем конструкторы **A()** и **B()** объектов **A** и **B**, не имеющие параметров. Затем конструктор класса **A** вызовет конструктор виртуального базового класса **V (3)**, который установит **a = 3**, но в теле конструктора класса **A** будет произведено присваивание **a = 1**, в результате чего **a** станет равен **1**. Затем будет вызван конструктор класса **B**, который установит **b = 2**.

Результаты работы программы

a=1 b=2 c=3



Пример

Рассмотрим иерархию классов, приведённую на рис. 4.3. Этот пример, как и предыдущий, иллюстрирует порядок вызова конструкторов виртуальных классов, но для более сложного случая.

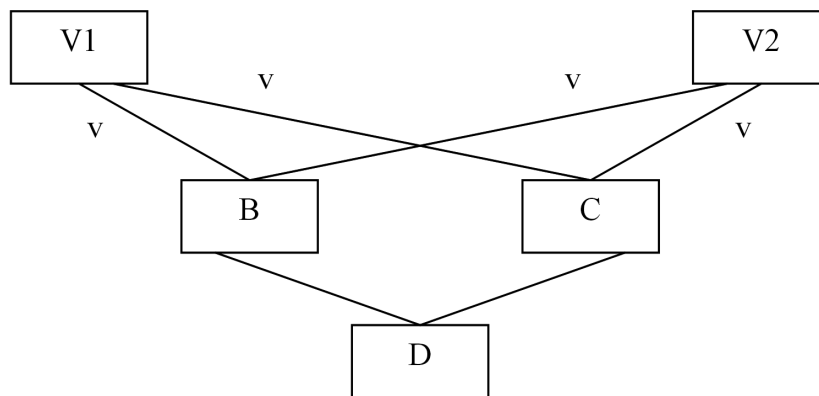


Рис. 4.3 – Многократное виртуальное наследование

Ниже приведён текст программы.

```

#include <stdio.h>          // стандартная библиотека ввода-вывода
class V1 // первый класс
{
    friend class D;    // дружественный класс
    friend class B;    // дружественный класс
    int fix1;
public:
    V1(int val): fix1(val){}; // конструктор
    V1(): fix1(10){};
};
class V2 // второй класс
{
    friend class D; // дружественный класс
    int fix2;
public:
    V2(): fix2(20){}; // конструктор
    V2(int p): fix2(p){};
};
// схема наследования
class B: virtual public V1, virtual public V2 {};
  
```

```

class C: virtual public V1, virtual public V2 {};
class D: public B,C {
public:
    D(): V1(30){};
    D(int p): V2(p){};
    // функция вывода
    void out(){printf("fix1=%d fix2=%d \n",fix1,fix2);}
};
int main()
{
    D ob1; D ob2(100);
    ob1.out();
    ob2.out();
    return 0;
}

```

При создании объекта **ob1** будут вызваны конструкторы **V1 (30)** и **V2()**, которые установят **fix1 = 30** и **fix2 = 20**. При создании объекта **ob2** будет вызван конструктор **V1()** без параметра, а затем — конструктор **V2 (100)**. Они установят **fix1 = 10** и **fix2 = 100**.

Результаты работы программы

```

fix1=30  fix2=20
fix1=10  fix2=100

```

.....

.....



Пример

.....

Рассмотрим иерархию классов, приведённую на рис. 4.4. Сначала вызываются конструкторы **V1, V2, V3**, а затем — **B, C, D**.

Ниже приведён текст программы, иллюстрирующей вышеприведённую иерархию.

```

#include <stdio.h>    // стандартная библиотека ввода-вывода
// построения иерархии
class V1 // класс V1
{
    friend class D;    // дружественный класс
    int fix1;
public:
    V1(int val): fix1(val){};
    V1(): fix1(10){};
};
class V2 // класс V2
{
    friend class D; // дружественный класс
    int fix2;

```

```

public:
    V2(): fix2(20){};
};
class V3 // класс V3
{
    int fix3;
    friend D;
public:
    V3(): fix3(40){};
    V3(int p): fix3(p){};
};
// схема наследования
class A: virtual public V1 {};
class B: virtual public V1 {};
class C: virtual public V2, virtual public V3 {};
class D: public B,C {
public:
    D(): V1(30){};
    D(int p): V3(p){};
    // функция вывода
void out(){printf("fix1=%d fix2=%d fix3=%d\n",fix1,fix2,fix3);}
};
int main()
{
    D ob1; D ob2(100);
    ob1.out();
    ob2.out();
    return 0;
}

```

Результаты работы программы

```

fix1=30  fix2=20  fix3=40
fix1=10  fix2=20  fix3=100

```

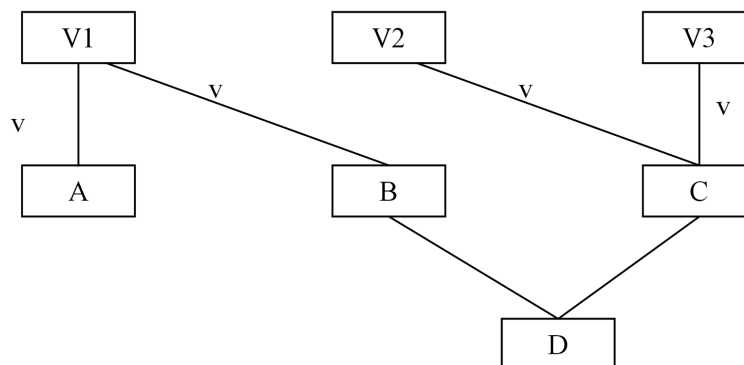


Рис. 4.4 – Иерархия классов из трех базовых классов



Контрольные вопросы по главе 4

- 1) Что такое производные классы?
- 2) Что такое переопределение?
- 3) Какой член базового класса называется переопределенным?
- 4) Что называется связным списком?
- 5) Как определяется множественное наследование?
- 6) Что такое виртуальный класс?
- 7) Как определяются конструкторы и деструкторы виртуальных классов?

Глава 5

ВИРТУАЛЬНЫЕ ФУНКЦИИ



.....
Полиморфизм в объектно-ориентированном программировании называется способность объекта отреагировать на некоторый запрос.
.....

Поскольку объект реагирует на запросы с помощью своих составных функций, то эта способность реализуется на основе механизма, позволяющего выбирать вызываемые функции не на шаге компиляции программы, а на шаге ее выполнения. Генерация вызова составной функции на шаге компиляции называется *ранним связыванием*, а на шаге выполнения — *поздним связыванием*. Функция, имя которой связывается с соответствующим ей кодом на стадии позднего связывания, называется *виртуальной*. В языке Си++ полиморфизм реализован на основе виртуальных функций [19, 20].

5.1 Переопределение составной функции



.....
*Если в базовом классе определена составная функция, которая должна различным образом выполняться для объектов различных производных классов, то она в этих производных классах должна быть определена заново. Такая функция называется **переопределенной**.*
.....

Предположим, что задан массив объектов базового класса. Если его элементы являются объектами производных классов, то функции базового класса не могут

быть переопределены. Например, рассмотрим класс «фрукты» и производные от него — «яблоки» и «апельсины».

```
#include <iostream.h>
#include <conio.h>
// Класс фрукты
class fruit {
public:
    void show()
    { cout << "фрукты"<< endl; }
};
// Класс яблоки
class apple {
public:
    void show()
    { cout << "яблоки" << endl; }
};
// Класс апельсины
class orange {
public:
    void show()
    { cout << "апельсины" << endl; }
};
void main()
{
    clrscr();    // Очистка экрана
    // Создаём объекты
    fruit *a = (fruit *)new apple, *b = (fruit *)new orange;
    a -> show(); b -> show(); // Выводим сообщения
    getch(); // Ожидание нажатия клавиши
}
```

В результате работы программы будет два раза выведено слово «фрукты», ибо оба оператора **a->show()** и **b->show()** вызовут функцию **show()** из базового класса. Для того чтобы решить проблему переопределения функций в производных классах, объекты которых заданы с помощью указателей на объекты базовых классов, применяются виртуальные функции. Они определяются в базовом классе следующим образом:

virtual тип_возвращаемого_значения имя(параметры).

Виртуальные составные функции позволяют выбирать члены класса с одним и тем же именем через указатель функции в зависимости от типа указателя.

В частности, если в нашем примере в базовом классе указать

virtual void show(),

а остальной текст оставить без изменения, то программа выведет слова «яблоки» и «апельсины».

Функция, определенная как виртуальная в базовом классе и переопределенная с таким же списком аргументов и типом возвращаемого значения в производном классе, становится *виртуальной* для объектов производного класса. Если она не переопределена в производном классе, то при ее вызове для объектов производного класса будет вызываться соответствующая функция из ближайшего по иерархии базового класса. Виртуальные функции не могут быть статическими.

5.2 Организация списка объектов различного типа

Определим класс, объектом которого является список объектов различного типа. Пусть, для определенности, список состоит из точек и отрезков. Точка задается двумя целыми координатами, а отрезок — четырьмя целыми координатами.

Класс точки зададим как класс элемента списка, имеющего, помимо координат, указатель на следующий элемент, конструктор и виртуальную функцию вывода на экран содержимого объекта класса точки:

```
// Класс точки
class Point
{
    // Собственные элементы
    friend MPoint;    // Класс MPoint будет дружественным
protected:
    // Защищённые элементы
    int x,y;          // Координаты
    int itype;        // Тип
    Point *next;      // Указатель на следующий элемент в списке
public:
    // Общедоступные элементы
    Point(int a,int b):x(a), y(b), itype(0) {}    // Конструктор
    virtual void get()
    {
        // Вывод координат точки
        cout << "Point(" << x << ', ' << y << ") \n";
    }
    virtual int type()    // Виртуальная функция возвращения типа
    { return itype; }
};
```

Класс отрезка определим как производный от класса точки, к которому добавлены координаты конца отрезка и переопределена функция вывода на экран:

```
// Класс линии (производный от класса точки)
class Line: public Point
{
    // Собственные элементы
    friend MPoint;    // Класс MPoint будет дружественным
    int x2,y2;        // Координаты второго конца линии
    Point *next;      // Указатель на следующий элемент в списке
public:
```

```

// Общедоступные элементы
// Конструктор
Line(int a,int b,int a2,int b2):Point(a,b), x2(a2), y2(b2) {}
virtual void get()
{
    // Вывод координат линии
    cout << "Line(" << x << ', ' << y << ")(";
    cout << x2 << ', ' << y2 << ")\\n";
}
};

```

Определим *класс списка*, состоящий из указателя на первый элемент и функций добавления точки, добавления отрезка и вывода элементов списка на экран:

```

// Класс списка
class MPoint
{ // Собственные элементы
    Point *p;    // Указатель на голову списка
public:
    // Общедоступные элементы
    MPoint() { p = NULL; }    // Конструктор
    void insert(Point z);    // Добавление точки в список
    void insert(Line t);    // Добавление линии в список
    void display();    // Вывод содержимого списка
};

```

Для того чтобы сделать доступными поля классов **Point** и **Line**, объявим класс **Mpoint** дружественным для этих классов. После добавления тестирующей главной программы получим окончательный текст программы.

```

#include <conio.h>
#include <iostream.h>
class MPoint;
// Класс точки
class Point
{
    // Собственные элементы
    friend MPoint;    // Класс MPoint будет дружественным
protected:
    // Защищённые элементы
    int x,y;    // Координаты
    int itype;    // Тип
    Point *next; // Указатель на следующий элемент в списке
public:
    // Общедоступные элементы
    Point(int a,int b):x(a), y(b), itype(0) {}    // Конструктор
    virtual void get()
    {

```

```

        // Вывод координат точки
        cout << "Point(" << x << ', ' << y << ")\\n";
    }
    virtual int type()    // Виртуальная функция возвращения типа
    { return itype; }
};

// Класс линии (производный от класса точки)
class Line: public Point
{
    // Собственные элементы
    friend MPoint;    // Класс MPoint будет дружественным
    int x2,y2;        // Координаты второго конца линии
    Point *next;      // Указатель на следующий элемент в списке
public:    // Общедоступные элементы
    // Конструктор
    Line(int a,int b,int a2,int b2):Point(a,b), x2(a2), y2(b2) {}
    virtual void get()
    {
        // Вывод координат линии
        cout << "Line(" << x << ', ' << y << ")(";
        cout << x2 << ', ' << y2 << ")\\n";
    }
};

// Класс списка
class MPoint
{
    // Собственные элементы
    Point *p;        // Указатель на голову списка
public:    // Общедоступные элементы
    MPoint() { p = NULL; }    // Конструктор
    void insert(Point z);    // Добавление точки в список
    void insert(Line t);    // Добавление линии в список
    void display();    // Вывод содержимого списка
};

// Вывод содержимого списка
void MPoint::display()
{
    Point *q = p; // Создаём указатель и устанавливаем его на голову списка
    while(q) // Пока не дойдём до конца списка
    {
        q->get(); // Вывод элемента списка
        q=q->next; // Переход к следующему элементу
    }
}

// Добавление точки в список
void MPoint::insert(Point z)
{
    if(!p) // Если список пустой

```

```

    {
        p = new Point(z.x, z.y);    // Создаём новый элемент
        p->next = NULL;             // Следующего элемента пока нет
        return;                     // Выход из функции
    }
// Если список не пустой
Point *q = p;    // Создаём указатель и устанавливаем его в начало списка
// Идём до конца списка
while(q->next)
    q = q->next;    // Переход к следующему элементу списка
q->next = new Point(z.x,z.y);    // Создаём новый элемент
q->next->next = NULL;            // Следующего элемента пока нет
}
// Добавление линии в список
void MPoint::insert(Line z)
{
    // Если список пустой
    if(!p)
    {
        p = new Line(z.x,z.y,z.x2,z.y2);    // Создаём новый элемент
        p->next = NULL;                     // Следующего элемента пока нет
        return;                             // Выход из функции
    }
// Если список не пустой
Point *q = p;    // Создаём указатель и устанавливаем его в начало списка
// Идём до конца списка
while(q->next)
    q = q->next;    // Переход к следующему элементу списка
q->next = new Line(z.x,z.y,z.x2,z.y2);    // Создаём новый элемент
q->next->next = NULL;                    // Следующего элемента пока нет
}
void main()
{
    clrscr();    // Очистка экрана
    MPoint a;    // Создаём список
    Line l1(10,100,-1,0);    // Создаём линию
    Point p1(1,2);    // Создаём точку
    a.insert(l1);    // Добавляем в список линию
    a.insert(l1);    // Добавляем в список линию
    a.insert(p1);    // Добавляем в список точку
    a.insert(p1);    // Добавляем в список точку
    a.insert(l1);    // Добавляем в список линию
    a.insert(p1);    // Добавляем в список точку
    a.display();    // Выводим содержимое списка на экран
    getch();        // Ожидание нажатия клавиши
}

```

В результате работы программы на экран будут выведены строки:

```
Line(10,100)(-1,0)
Line(10,100)(-1,0)
Point(1,2)
Point(1,2)
Line(10,100)(-1,0)
Point(1,2)
```

5.3 Техническая реализация виртуальных функций



.....
Совокупность классов и подклассов, в которых определяется и переопределяется виртуальная функция, называется **полиморфическим кластером** этой виртуальной функции.
.....

Обычную составную функцию также можно переопределить в наследуемых классах. Однако без атрибута **virtual** такая виртуальная функция будет связана с вызовом на этапе компиляции. Атрибут **virtual** гарантирует позднее связывание в пределах полиморфического кластера.

Чтобы добиться позднего связывания для объекта, его нужно объявить как указатель или ссылку на объект соответствующего класса. Для открытых производных классов (**public**) указатели и ссылки на объекты этих классов совместимы с указателями и ссылками на объекты базового класса. Объект Си++ представляет собой непрерывный участок памяти. Указатель на такой объект содержит адрес этого участка. Вызов составной функции транслируется в обычный вызов функции с дополнительным аргументом, содержащим указатель на объект. *Например:*

```
class Name *object;
Object -> message (10);
```

преобразуется в

```
className_message(object, 10);
```

При создании объектов производных классов их поля сцепляются с полями родительских классов. Эту функцию выполняет компилятор. Виртуальные функции реализованы с использованием таблиц функций. *Таблица виртуальных функций* **virtual_table** содержит составные функции каждого класса, принадлежащего полиморфическому кластеру. Указатель на эту таблицу имеют все объекты классов и производных классов полиморфического кластера. Компилятор преобразует вызов виртуальной функции через таблицу **virtual_table**.

Например, если класс **Third** имеет виртуальную функцию **out()**, то объявление объекта с помощью указателя и вызов

```
Third *c; c -> out();
```

компилируется в

```
(*(c -> virtual_table [2])) (c);
```

если переопределенная функция имеет в таблице виртуальных функций индекс 2.

5.4 Виртуальные деструкторы

Конструктор класса не может быть виртуальным, ибо при создании объекта тип этого объекта должен быть заранее известен компилятору. С деструктором ситуация иная: оператор **delete** при разрушении объекта производного класса, объявленного с помощью указателя на базовый класс, будет вызывать деструктор базового класса. В результате объект производного класса будет разрушаться некорректно. Деструкторы базовых классов вызываются после деструкторов производных классов. *Виртуальные деструкторы* базовых классов вызываются прямо или косвенно любыми деструкторами производных классов.



Рассмотрим следующую программу, в которой виртуальный деструктор базового класса будет переопределён деструктором производного класса.

```
#include <iostream.h>
#include <conio.h>
class integral
{
    int num;
public:
    integral(int n):num(n) {}
    virtual ~integral()
    { cout << "Inside integral\n"; }
};
class rational: public integral
{
    int den;
public:
    rational(int n,int d): integral(n), den(d) {}
    ~rational()
    { cout << "Inside rational\n"; }
};
rational obj(13,5);
int main()
{
    clrscr();
    integral *ref = &obj;
```

```

ref->integral::~~integral();
return 0;
}

```

В этом примере при явном вызове деструктора сначала будет выполнен деструктор производного класса, а потом — деструктор базового класса.

В результате работы программы на экран будет выведено:

```

Inside integral
Inside rational
Inside integral

```

Если же сделать деструктор базового класса неvirtуальным, то сработает только деструктор базового класса.

.....

5.5 Абстрактные классы



.....
*Функция, объявленная, но не определенная в базовом классе, называется **чисто виртуальной**.*

Чисто виртуальная функция должна быть переопределена в каком-нибудь из производных классов. В базовом классе она объявляется с помощью оператора

virtual min_возвр_значения имя(параметры) = 0;

Например,

```

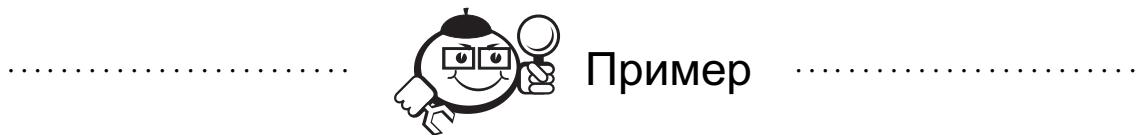
Class Figure
{
    int color;
public:
    virtual void show();
};

```

Класс, содержащий хотя бы одну чисто виртуальную функцию, называется *абстрактным классом*. Абстрактный класс может служить только в качестве базового для других классов, ибо объект такого класса создать невозможно. В производных от него классах чисто виртуальные функции должны быть либо переопределены, либо вновь указаны как абстрактные. Абстрактный класс нельзя указать в качестве типа аргумента или возвращаемого значения функции. Однако разрешено (и это часто используется) создавать указатель на абстрактный базовый класс, а также ссылку на такой класс, если для ее инициализации не требуется создания временного объекта.

Составные функции абстрактного класса могут вызывать чисто виртуальные составные функции этого же класса. Абстрактный класс может иметь конструкторо-

ры и деструкторы. Деструктор базового класса вызывается при разрушении объекта, после того, как все подобъекты этого объекта уже разрушены. Поэтому деструктор базового класса не должен вызывать чисто виртуальные функции своего класса, так как такой вызов приведет к ошибке.



Рассмотрим приложения производных классов в машинной графике. Основным базовым классом здесь будет область — подмножество точек плоскости, для которых задается цвет, и функция, устанавливающая, принадлежит ли этой области точка, которая является аргументом функции. Эта функция называется тестом на принадлежность.

Поскольку заранее неизвестно, каким образом задана область, то тест на принадлежность определим как чисто виртуальную функцию. Будем рассматривать следующую область:

- выпуклая область, заданная системой линейных неравенств (класс **Convex**);
- многоугольник, заданный списком своих вершин (класс **Polygon**);
- выпуклый многоугольник (класс **CPolygon**);
- звездчатый многоугольник (класс **SPolygon**).

Эти классы составляют иерархию, приведенную на рис. 5.1.

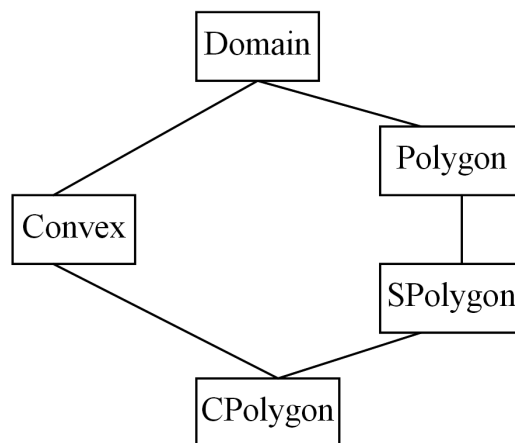


Рис. 5.1 – Иерархия классов областей

Определим область как абстрактный класс:

```

class Domain
{
protected: // Защищённые элементы класса
    int color; // цвет области
    int n; // количество сторон
public:
    // Общедоступные элементы класса
  
```

```

Domain(int c = WHITE): color(c) {} // Конструктор
// Определение принадлежности точки области
virtual int isin(Point p) = 0;
// Функция вывода области на экран
void show(double xmin, double ymin, double xmax, double ymax);
};

```

Класс **Point** определим позже.

Выпуклая область может быть неограниченной. Она в нашей программе будет пересечением конечного числа полуплоскостей, заданных линейными неравенствами $a_i x + b_i y + c_i \leq 0$, $i = 0, 1, \dots, n - 1$.

Функция **isin(Point p)** будет возвращать 1, если и только если координаты точки **p** удовлетворяют этим неравенствам.

```

class Convex: virtual public Domain
{
protected:
// Защищённые элементы класса
double *a, *b, *c; // Коэффициенты ограничивающих прямых ax+by+c=0
public:
// Общедоступные элементы класса
Convex() { n = 0; } // Конструктор по умолчанию
// Конструктор
Convex(double *av, double *bv, double *cv, int nv, Point p, int cl);
// Переопределённая функция определения принадлежности точки области
int isin(Point p);
};
// Конструктор
Convex::Convex(double *av, double *bv, double *cv, int nv,
                Point p, int cl):Domain(cl)
{
// Задается некоторая внутренняя точка p многоугольника
int i;
n = nv;
// Выделяем память под коэффициенты
a = new double[n];
b = new double[n];
c = new double[n];
for (i = 0; i < nv; i++)
{
if (av[i] * p.x + bv[i] * p.y + cv[i] <= 0) // Если вектор нормали направлен
// наружу
{ a[i] = av[i]; b[i] = bv[i]; c[i] = cv[i]; }
else // Иначе изменяем направление нормали на противоположное
{ a[i] = -av[i]; b[i] = -bv[i]; c[i] = -cv[i]; }
}
}
}

```

```
// Переопределённая функция определения принадлежности точки области
int Convex::isin(Point p)
{
    int i;
    // Перебор всех ограничивающих прямых
    for (i = 0; i < n; i++)
        if (a[i] * p.x + b[i] * p.y + c[i] > 0) return 0; // Точка расположена вне области
    return 1; // Если точка расположена с противоположной стороны от нормали
}
```

Второй из конструкторов класса **Convex** устанавливает коэффициенты прямых, ограничивающих область, при которых точка **p** становится внутренней.

Класс **Polygon** определим как последовательность вершин многоугольника:

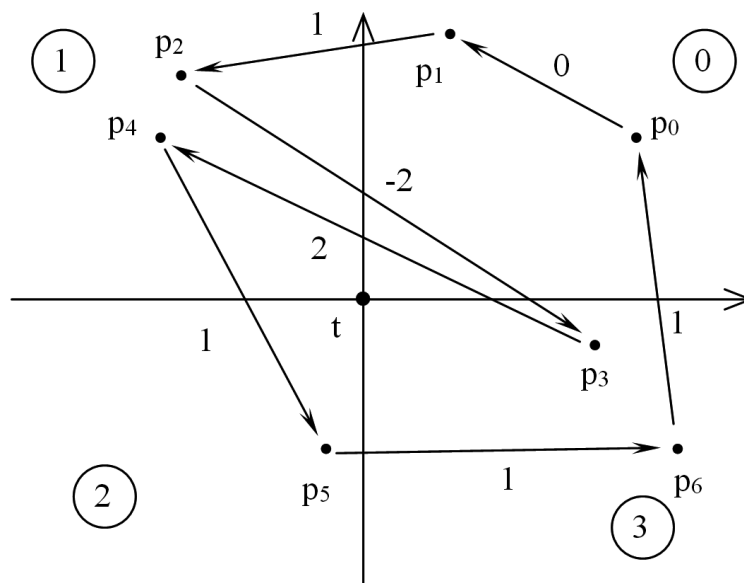
```
class Polygon: virtual public Domain
{
protected:
    // Защищённые элементы класса
    Point *p; // Список вершин многоугольника
public:
    // Общедоступные элементы класса
    Polygon() {n = 0;} // Конструктор по умолчанию
    // Конструктор
    Polygon(double *x, double *y, int num, int cl);
    // Переопределённая функция определения принадлежности точки области
    int isin(Point t);
};
```

Проверку на принадлежность точки многоугольнику выполним *методом углов*. В этом методе проверяемая точка **t** берется за начало новой системы координат. В результате вся плоскость будет разбита на четыре четверти. Пусть $c_i = \text{code}(p_i) \in \{0, 1, 2, 3\}$ обозначает четверть, в которой лежит точка p_i . Положим:

$$m_i = \begin{cases} 0, & \text{если } c_i = c_{i+1}, \\ 1, & \text{если } c_{i+1} = c_i + 1 \bmod 4, \\ 2, & \text{если } c_{i+1} = c_i - 1 \bmod 4, \\ \pm 2, & \text{если } c_{i+1} = c_i + 2 \bmod 4. \end{cases}$$

В последнем случае, если **t** находится слева от $p_i p_{i+1}$, то $m_i = +2$, а если справа, то $m_i = -2$. Индексом $\text{ind}_t M$ точки **t** относительно многоугольника M называется число $\sum_{i=0}^{n-1} m_i / 4$. Имеет место следующее утверждение: индекс $\text{ind}_t M$ равен 0 тогда и только тогда, когда точка **t** лежит вне многоугольника M .

Рассмотрим, *например*, точку **t** и многоугольник, изображенные ниже на рис. 5.2. Точки первой четверти имеют код $\text{code}(p) = 0$, второй — 1, третьей — 2, четвертой — 3. Получаем $m_0 = 0$, $m_1 = 1$, $m_2 = -2$, $m_3 = 2$, $m_4 = 1$, $m_5 = 1$, $m_6 = 1$. Индекс равен $\text{ind}_t M = (1 - 2 + 2 + 1 + 1) / 4 = 1$. Следовательно, точка **t** — внутренняя, по отношению к многоугольнику.

Рис. 5.2 – Вычисление индекса точки t

Предполагая, что класс **Point** имеет составную функцию **code(Point q)**, возвращающую номер четверти, которой принадлежит точка **q**, если взять данную точку в качестве начала системы координат, приведём подпрограмму теста на принадлежность точки многоугольнику:

```
int Polygon::isin(Point t)
{
    int i, ind = 0;
    Point q = p[n-1];
    // Перебор вершин полигона для вычисления индекса полигона
    for (i = 0; i < n; i++)
    {
        if (t.code(q) == t.code(p[i])) ;
        else if ((t.code(p[i]) - t.code(q) - 1)% 4 == 0) ind++;
        else if ((t.code(p[i]) - t.code(q) + 1)% 4 == 0) ind--;
        else if ((p[i] - q) * (t - q) > 0) ind += 2;
        else ind -= 2;
        q = p[i];
    }
    if (ind == 0) return 0;
    return 1;
}
```

Для реализации этой подпрограммы класс **Point** должен также содержать операции присваивания, разности и векторного произведения, равного ориентированной площади параллелограмма, построенного по векторам, соединяющим начало координат с точками.

Подпрограмму **show()** можно реализовать с помощью функции **fillpoly()**:

```
Void Polygon :: show()
```

```

{
    int coord = new int[2*n];
    setfillstyle (SOLID_FILL, color);
    fillpoly(n, coord);
}

```

Определим класс звездчатого многоугольника. Напомним, что многоугольник M называется *звездчатым*, если существует такая точка $\mathbf{r}$, что для каждой точки $\mathbf{p} \in M$ весь отрезок, соединяющий точки $\mathbf{p}$ и $\mathbf{r}$, содержится в M . Точки $\mathbf{r}$, обладающие этим свойством, называются *ядерными*.

Множество ядерных точек называется *ядром звездчатого многоугольника*. Заметим, что ядро звездчатого многоугольника всегда выпукло.

```

class SPolygon: public Polygon
{
protected: // Защищённые элементы класса
    Point pC; // Центр тяжести
public:
    // Общедоступные элементы класса
    SPolygon() {} // Конструктор по умолчанию
    // Конструктор
    SPolygon(double *x, double *y, int m, int cl);
};

```

Определим звездчатый полигон как производный от класса **Polygon**. Конструктор строит звездчатый полигон из набора точек $S_0, S_1, \dots, S_{n-1}$ с помощью присоединения по одной точке. Вначале звездчатый полигон состоит из единственной точки S_0 . Затем в цикле по $i = 1, 2, \dots, n - 1$ добавляются точки S_i . Точки упорядочиваются в порядке возрастания угла $S_i S_0$. Затем точка S_0 удаляется из списка вершин, и точки последовательно соединяются, как это показано на рис. 5.3. Лучи соединяют вершину S_0 с вершинами S_i .

Класс выпуклого многоугольника определим как производный от класса **SPolygon**. Конструктор будет строить выпуклый многоугольник как выпуклую оболочку с помощью алгоритма Дейкстры.

Пусть M — выпуклый многоугольник, $\mathbf{p}$ — произвольная внешняя точка. Сторона многоугольника называется *видимой* из точки $\mathbf{p}$, если существует такая точка $\mathbf{q}$, принадлежащая этой стороне, что отрезок, соединяющий ее с точкой $\mathbf{p}$, не имеет других общих точек со стороной, кроме точки $\mathbf{q}$. На рис. 5.4 видимыми будут стороны $S_0 S_1$, $S_1 S_2$ и $S_5 S_0$. В частности, для стороны $S_5 S_0$ такой точкой $\mathbf{q}$ будет S_0 .

Если заданы выпуклый многоугольник и точка $\mathbf{p}$, то, удаляя видимые стороны и соединяя точку $\mathbf{p}$ с двумя крайними точками оставшихся сторон, получаем выпуклый многоугольник, который будет выпуклой оболочкой точек S_i и $\mathbf{p}$. Конструктор строит выпуклую оболочку, последовательно добавляя по одной точке к предшествующим выпуклым многоугольникам. Ниже приведен текст программы, в которой реализована иерархия классов, приведенная на рис. 5.1.

```

#include <graphics.h>
#include <conio.h>

```

```

#include <math.h>
#include <stdlib.h>
#define PI 3.14159
struct Point // класс точки
{
    double x,y; // координаты точки
    int code(Point q); // код четверти, в которой лежит точка q
    double operator*(Point q); // векторное произведение
    Point operator-(Point q); // разность векторов
};
int Point::code(Point q) // код четверти, в которой лежит точка q
{
    // начало координат находится в точке (x,y)
    if (q.x-x>=0 && q.y-y>=0) return 0;
    if (q.x-x< 0 && q.y-y>=0) return 1;
    if (q.x-x>=0 && q.y-y< 0) return 3;
    if (q.x-x< 0 && q.y-y< 0) return 2;
}
double Point::operator*(Point q)
{ return x*q.y-y*q.x; // векторное произведение
}
Point Point::operator-(Point q) // разность векторов
{ Point t; t.x=x-q.x; t.y=y-q.y; return t; }
int operator <(Point p, Point q)
{
    Point t; // сравнение углов радиус-векторов p и q
    t.x=0; t.y=0; // коды четвертей вычисляются относительно (0,0)
    if(t.code(p)<t.code(q)) return 1;
    if(t.code(p)>t.code(q)) return 0;
    return (p*q>0); // вращение от p к q направлено против часовой стрелки
}
int intersect(Point p,Point p1, Point p2)
{
    // тест на пересечение луча и отрезка
    if (p1.y==p2.y) return 0;
    if ((p1.y<p2.y?p2.y:p1.y)<=p.y) return 0;
    if ((p1.y<p2.y?p1.y:p2.y)>p.y) return 0;
    if (p2.y-p1.y>0) return
        ((p.x-p1.x)*(p2.y-p1.y)-(p2.x-p1.x)*(p.y-p1.y) > 0);
    else return
        ((p.x-p1.x)*(p2.y-p1.y)-(p2.x-p1.x)*(p.y-p1.y) < 0);
}
class Domain // абстрактный класс области
{
protected:
    int color; // цвет области
    int n; // количество сторон
public:

```

```

    virtual int isin(Point p)=0; // функция принадлежности
    Domain(int c=15): color(c){} // конструктор
    void show(double xmin, double ymin, double xmax, double ymax);
    // функция вывода области
};

void Domain::show(double xmin, double ymin, double xmax, double ymax)
{
    int ix, iy; Point q;
    for (iy=0; iy<=getmaxy(); iy++)
    for (ix=0; ix<=getmaxx(); ix++)
    {
        q.x = xmin+ix*(xmax-xmin)/(getmaxx()+1);
        q.y = ymin+(getmaxy()+1-iy)*(ymax-ymin)/(getmaxy()+1);
        if (isin(q)) putpixel(ix,iy, color);
    }
}

class Convex: virtual public Domain
{
protected:
    double *a, *b, *c; // коэффициенты ограничивающих прямых ax+by+c=0
public:
Convex(){n=0;}
Convex(double *av, double *bv, double *cv, int nv, Point p,int cl);
    int isin(Point p); // функция принадлежности переопределена
};

Convex::Convex(double *av, double *bv, double *cv, int nv,
                Point p,int cl):Domain(cl)
{
    // задается некоторая внутренняя точка p многоугольника
    int i; n = nv;
    a = new double[n]; b = new double[n]; c = new double[n];
    for (i=0; i<nv; i++)
    {
        if (av[i]*p.x+bv[i]*p.y+cv[i]<=0) // если вектор нормали
        { // направлен наружу
            a[i]=av[i]; b[i]=bv[i]; c[i]=cv[i]; }
        else // иначе изменяем направление нормали на противоположное
        { a[i]=-av[i]; b[i]=-bv[i]; c[i]=-cv[i]; }
    }
}

int Convex::isin(Point p)
{
    int i;
    for (i=0; i<n;i++)
    if (a[i]*p.x+b[i]*p.y+ c[i] > 0) return 0; // точка лежит вне области
    return 1; // если точка лежит с противоположной стороны от нормали
}

class Polygon: virtual public Domain

```

```

{
    protected:
        Point *p;
    public:
        Polygon() {n=0;}
        Polygon(double *x, double *y, int num, int cl);
        int isin(Point t);
};

Polygon::Polygon(double *x, double *y, int num, int cl):Domain(cl)
{
    int i; n = num; p= new Point [num];
    for (i=0; i<n; i++)
        { p[i].x=x[i]; p[i].y=y[i]; }
}
/*
int Polygon::isin(Point t) // тест на принадлежность методом углов
{
    int i, ind=0;
    Point q = p[n-1];
    for (i=0; i<n; i++)
    {
        if (t.code(q)==t.code(p[i]));
            else if ((t.code(p[i])-t.code(q)-1)%4==0) ind++;
            else if ((t.code(p[i])-t.code(q)+1)%4==0) ind--;
            else if ((p[i]-q)*(t-q)>0) ind+=2;
            else ind-=2;
        q = p[i];
    }
    ind = ind/4;
    if (ind==0) return 0; else return 1;
} */
int Polygon::isin(Point t)
{
    int i, parity=0;
    for (i=0; i<n; i++)
        if (intersect(t, p[i],p[(i+1)%n]))
            parity = 1-parity; return parity;
}

class SPolygon: public Polygon
{
    protected:
        Point pC;
    public:
        SPolygon() {} // конструктор по умолчанию
        SPolygon(double *x, double *y, int m, int cl);
};

```

```

SPolygon::SPolygon(double *x, double *y, int m, int cl):Domain(cl)
{
    int i,j;
    Point t;
    p= new Point [m]; n=m;
    pC.x=0; pC.y=0;
    for(i=0;i<m;i++)
        { pC.x+=x[i]; pC.y+=y[i]; }
    pC.x= pC.x/m; pC.y= pC.y/m;
    for(i=0;i<m;i++)
        { p[i].x=x[i]; p[i].y=y[i]; }
    //Сортируем точки по возрастанию угла вокруг центра тяжести методом вставок
    for(i=1; i<m; i++)
        {
            t = p[i];
            for (j=i-1; (j>=0) && ((t-pC)<(p[j]-pC)); j--)
                p[j+1] = p[j];
            p[j+1]=t;
        }
}
class CPolygon: public SPolygon, public Convex
{
public:
    int isin(Point r){return Polygon::isin(r);};
    void Insert(Point t)
    {
        int i; int j0, j1, j;
        int *del = new int [n];
        Point *q= new Point [n+1];
        if (isin(t)) return;
        j0=j1=0;
        for (i=0; i<n; i++)
        {
            if ((t-p[i])*(p[(i+1)%n]-p[i])>=0) del[i]=1;
            else del[i]=0;
        }
        for (i=0;i<n;i++)
            if (del[i]==1&& del[(i+1)%n]==0) break;
        j=0; i=(i+1)%n;
        while(del[i]==0)
        {
            q[j++]=p[i];
            i=(i+1)%n;
        }
        q[j]=p[i];
        q[j+1] = t; delete [] p;
        p=new Point [j+2]; n=j+2;
        for (i=0; i<n; i++)
    
```

```

        p[i]=q[i];
        delete []q;
    }
    CPolygon(double *x, double *y, int m, int cl);
};
CPolygon::CPolygon(double *x, double *y, int m, int cl):Domain(cl)
{ // выпуклая оболочка методом Дейкстры
    int i; Point t; p= new Point [3]; n=3;
    for (i=0; i<3; i++)
    { p[i].x=x[i]; p[i].y=y[i]; }
    if ((p[1]-p[0])*(p[2]-p[1])<0)
    { t=p[1]; p[1]=p[2]; p[2]=t; }
    for(i=3; i<m; i++)
    {
        t.x=x[i]; t.y=y[i];
        Insert (t);
    } }
CPolygon::CPolygon(double *x, double *y, int m, int cl):Domain(cl)
{ // выпуклая оболочка методом заворачивания подарка
    int i, j0, j1, j=0, k=0; Point t, r, *q= new Point [m];
    for (i=0; i<m; i++)
    {
        q[i].x=x[i]; q[i].y=y[i];
        if (x[i]>x[j]) j=i; // находим точку с максимальной x-координатой
    }
    j0=j; // начало стороны многоугольника
    j1=(j+1)%m;
    while(j1%m!=j0)
    {
        p[k++]=q[j]; r=q[j]; t= q[(j+1)%m]; j1=(j+1)%m;
        for(i=(j+1)%m; i!=j; i=(i+1)%m)
            if (q[i]-r< t-r)
                { t= q[i]; j1=i; }
        j=j1;
    } p[k++]=q[j1%m];
}
int main()
{
    // определяем область, ограниченную прямыми x=200, y=200, x=-200, y=-200
    double a[4]={1,0,1,0}, b[4]={0,1,0,1}, c[4]={200,200,-200,-200};
    double rad=150, x[10], y[10]; // радиус окружности и вершины пятиугольника
    int i;
    Point r={0,0}; // внутренняя точка для области Convex
    Convex dom(f,b,c,4,r,LIGHTBLUE); // область Convex является квадратом
    for(i=0;i<5;i++)
    { // вершины большого пятиугольника

```

```

    x[i] = rad*cos(2*PI*i/5); y[i] = rad*sin(2*PI*i/5);
    // вершины маленького пятиугольника
    x[i+5] = rad*cos(2*PI*i/5+PI/5)/2;
    y[i+5] = rad*sin(2*PI*i/5+PI/5)/2;
}
randomize();
for(i=0;i<10;i++)
    x[i] += random(100)-200;
Polygon dom1(x,y,10,GREEN); // построение многоугольника
for(i=0;i<10;i++)
    x[i] += 300;
CPolygon dom2(x,y,10,BLACK); // построение выпуклой обложки
SPolygon dom3(x,y,10,LIGHTRED); // построение звездчатого полигона
int gm, gd=DETECT;
initgraph(&gd, &gm, "..\\bgi"); // инициализация графики
setfillstyle(SOLID_FILL, WHITE); // фон — белый
bar(0, 0, getmaxx(), getmaxy()); // закрашка экрана
dom.show(-300,-300,300,300); // вывод квадрата
dom1.show(-300,-300,300,300); // вывод многоугольника
dom2.show(-300,-300,300,300); // вывод выпуклой обложки
dom3.show(-300,-300,300,300); // вывод звездчатого полигона
getch(); closegraph();
    return 0;
}

```

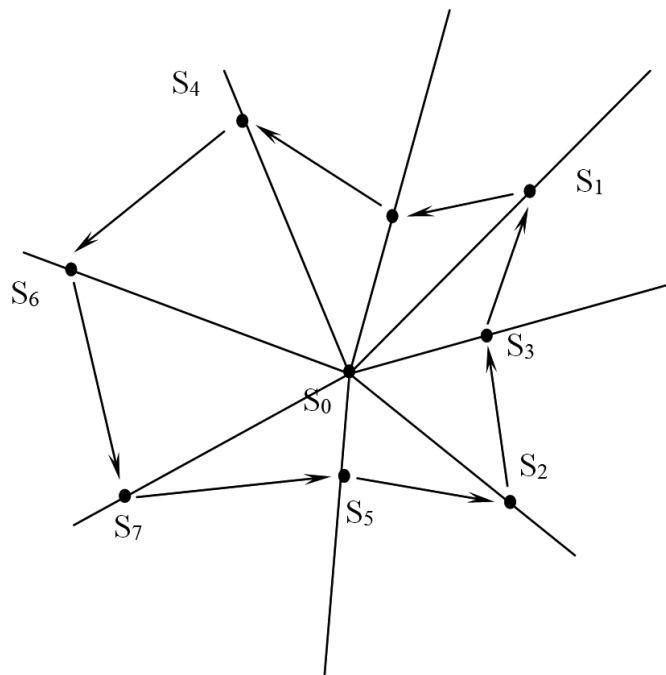


Рис. 5.3 – Построение звездчатого многоугольника

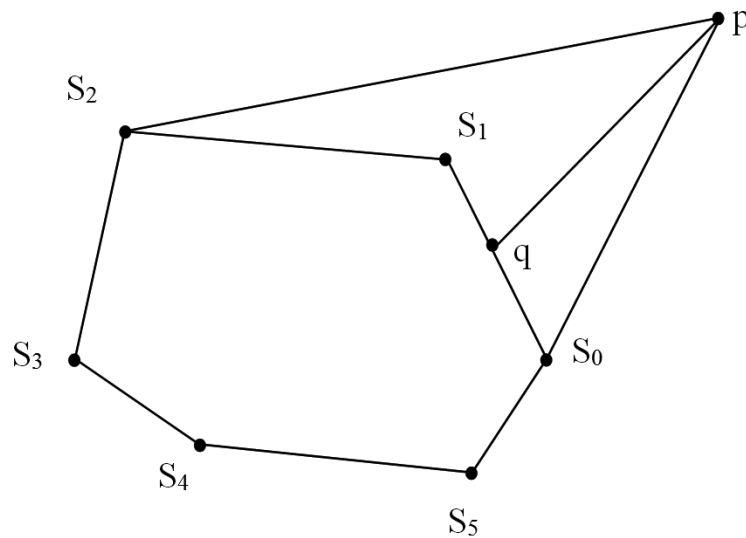


Рис. 5.4 – Добавление точки методом Дейкстры



Контрольные вопросы по главе 5

- 1) Что такое раннее связывание?
- 2) Что такое позднее связывание?
- 3) Что такое виртуальная функция?
- 4) Что такое переопределение составной функции?
- 5) Какая функция называется переопределенной?
- 6) Может ли конструктор класса быть виртуальным?
- 7) Может ли деструктор класса быть виртуальным?
- 8) Что такое абстрактные классы?
- 9) Какая функция называется чисто виртуальной?
- 10) Объект абстрактного класса можно создать?

Глава 6

ТЕОРЕТИЧЕСКИЕ ОСНОВЫ ОБЪЕКТНО-ОРИЕНТИРОВАННОГО ПРОГРАММИРОВАНИЯ

В первой главе был дан краткий экскурс в основы программирования на языке C++. Далее будет рассмотрено детальное описание объектно-ориентированных свойств языка.

ООП появилось в результате длительной эволюции технологии разработки программных продуктов. В основу эволюции легло стремление ускорить процесс создания надежных программных средств. Объектные модели различных версий универсальных языков программирования имеют свои особенности, но в их основе лежат единые концепции, что позволяет говорить о теоретических основах ООП [21–25].

6.1 Парадигмы программирования

В программировании существуют различные парадигмы, представляющие собой разные подходы к написанию программ. Большинство программистов знакомы лишь с немногими из них, это визуальное и процедурное программирование, однако существует программирование, основанное на логическом, параллельном программировании, программировании потоков данных, программирование реального времени и др. Объектно-ориентированный подход вбирает в себя различные преимущества многих парадигм программирования.

Такое количество парадигм связано с огромным количеством разнообразных научных и технических задач, а также новыми видами компьютерных систем. Общая архитектура компьютеров позволяет с большей или меньшей мерой эффективности моделировать одну архитектуру с помощью другой. Из архитектур наиболее удачны те, в которых за счет аппаратуры и программного обеспечения достигнута наивысшая скорость и простота использования.

Существующие языки программирования классифицируют по четырём основным группам (парадигмам): процедурные, объектно-ориентированные, функциональные и логические. Кратко рассмотрим особенности каждого подхода.

Процедурное программирование. Программа отделяется от данных. Программа состоит из последовательности команд, обрабатывающих данные. Данные, как правило, хранятся в виде переменных. Весь процесс вычисления сводится к изменению их содержимого.

Объектно-ориентированное программирование. Переменные и функции группируются в так называемые классы. Благодаря этому достигается более высокий уровень структуризации программы. Объекты, порождённые от классов, вызывают методы (функции или процедуры) друг друга и меняют таким образом состояние свойств (переменных). В принципе объектно-ориентированный способ написания программ не является чем то особенным и самостоятельным, поскольку базируется на процедурной модели программирования.

Функциональное программирование. Программа состоит из совокупности функций, которые вызывают друг друга. Переменные могут отсутствовать вообще. Алгоритмы, записанные в функциональном виде, как правило, короче и содержат меньше ошибок, чем аналогичные объектно-ориентированные или процедурные. Функциональное программирование считается программированием сверхвысокого уровня. Языки этой группы обладают относительно низким быстродействием из-за сложности реализации.

Логическое программирование. Основывается на формальной логике и Булевой алгебре (в некоторых языках применяются средства нечёткой логики, что позволяет создавать системы искусственного интеллекта). Программа, записанная на логическом языке программирования, не содержит в себе конкретных алгоритмов (действий и команд типа сделать то, затем это). Задаётся описание условий задачи и логических отношений, по которым система программирования сама рассчитывает возможные следствия и взаимосвязи введённых данных и формул.

Языки программирования часто оценивают по уровню. Уровень языка показывает, насколько язык близок к естественной для человека записи. Процедурные языки — самого низкого уровня. Функциональные — значительно выше. Логические языки в принципе могут быть самого высокого уровня, но из-за высокой сложности теории, лежащей в их основе, разрабатываются довольно медленно. Широко распространённых в мире языков логической группы довольно мало.

История программирования показывает стремление, с одной стороны, приблизиться к этому идеалу, а с другой стороны, как можно больше автоматизировать процесс решения задач, чтобы программист не описывал во всех подробностях алгоритмы и действия. Идеальным случаем такой автоматизации будет полное отсутствие каких-либо последовательностей действий — оператор будет вводить только условия задачи — всё остальное, включая выбор метода решения, построение структур данных, создание алгоритмов, будет выполнять сама система программирования.

6.2 От процедурного программирования к объектному

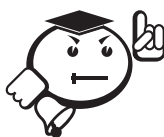
Развитие программирования активно ведется с начала 50-х годов XX века (т. е. уже более 50-ти лет!). На протяжении всех этих лет практика программирования требовала совершенствования технологических приемов и создания на их основе таких средств программирования, которые упростили бы процесс разработки программ, позволяя создавать все более сложные программные системы. Первые программы были организованы очень просто. Они состояли из собственно программы на машинном языке и обрабатываемых данных. Сложность программ ограничивалась способностью программиста одновременно, мысленно отслеживать последовательность выполняемых операций и местонахождение большого количества данных.

Создание сначала ассемблеров, а затем и языков высокого уровня сделало программу более обозримой за счет снижения уровня детализации и естественно позволило увеличить ее сложность. Появление в языках средств, позволяющих оперировать подпрограммами, существенно снизило трудоемкость разработки программ. Подпрограммы можно было сохранять и использовать в других программах. В результате были накоплены огромные библиотеки расчетных и служебных подпрограмм, которые по мере надобности вызывались из разрабатываемой программы.

Типичная программа того времени состояла из основной программы, области глобальных данных и набора подпрограмм (в основном библиотечных), выполняющих обработку всех данных или их части (рис. 6.1, а, б).

Слабым местом такой архитектуры было то, что при увеличении количества подпрограмм возрастала вероятность искажения части глобальных данных какой-либо подпрограммой. Например, обычно подпрограмма поиска корней уравнения на заданном интервале по методу деления отрезка пополам меняет величину интервала. Если при выходе из подпрограммы не предусмотреть восстановления первоначального интервала, то в глобальной области окажется неверное значение интервала, полученное при последнем делении отрезка в ходе работы подпрограммы. Необходимость исключения таких ошибок привела к идее использования в подпрограммах локальных данных (рис. 6.2).

И вновь сложность разрабатываемого программного обеспечения стала ограничиваться возможностью программиста отслеживать процессы обработки данных уже на новом уровне. К этому добавились проблемы согласования интерфейса при ведении разработки несколькими программистами. В результате встал вопрос создания технологии разработки сложных программных продуктов, снижающей вероятность появления ошибок.



.....
Усилиями многих авторов такая технология была создана и получила название «структурное программирование». Структурное программирование представляет собой совокупность рекомендуемых технологических приемов, охватывающих выполнение всех этапов разработки программного обеспечения.
.....

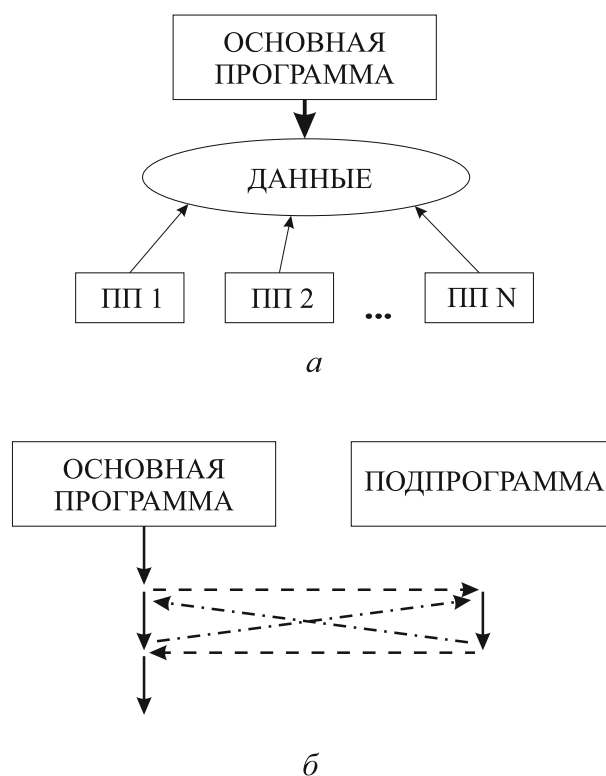


Рис. 6.1 – Архитектура программы, использующей глобальную область данных:
a – структура программы и *б* – виды взаимодействия основной программы с подпрограммами

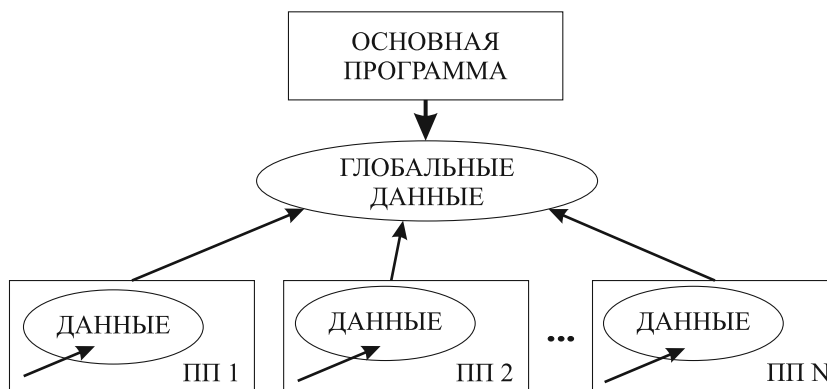


Рис. 6.2 – Архитектура программы, использующей подпрограммы с локальными данными

Были сформулированы основные принципы выполнения разработки:

- принцип нисходящей разработки, рекомендуемый на всех этапах вначале определять наиболее общие моменты, а затем поэтапно выполнять детализацию (что позволяет последовательно концентрировать внимание на небольших фрагментах разработки);
- структурное программирование, рекомендуемое определенные структуры алгоритмов и стиль программирования (чем нагляднее текст программы, тем меньше вероятность ошибки);

- принцип сквозного структурного контроля, предполагающий проведение содержательного контроля всех этапов разработки (чем раньше обнаружена ошибка, тем проще ее исправить).

В основе структурного программирования лежит декомпозиция (разбиение на части) сложных систем с целью последующей реализации в виде отдельных небольших (до 30...100 операторов) подпрограмм. В отличие от используемого ранее интуитивного подхода к декомпозиции, структурный подход требовал представления задачи в виде иерархии подзадач простейшей структуры, для получения которой рекомендовалось применять метод пошаговой детализации. С появлением других принципов декомпозиции (объектного, логического и т. д.) данный способ получил название процедурной декомпозиции.

Метод пошаговой детализации заключается в следующем. Определяется общая структура программы в виде одного из трех вариантов (рис. 6.3):

- последовательности подзадач (например, ввод данных, преобразование данных, вывод данных),
- альтернативы подзадач (например, добавление записей к файлу или их поиск),
- повторения подзадачи (например, циклически повторяемая обработка данных).

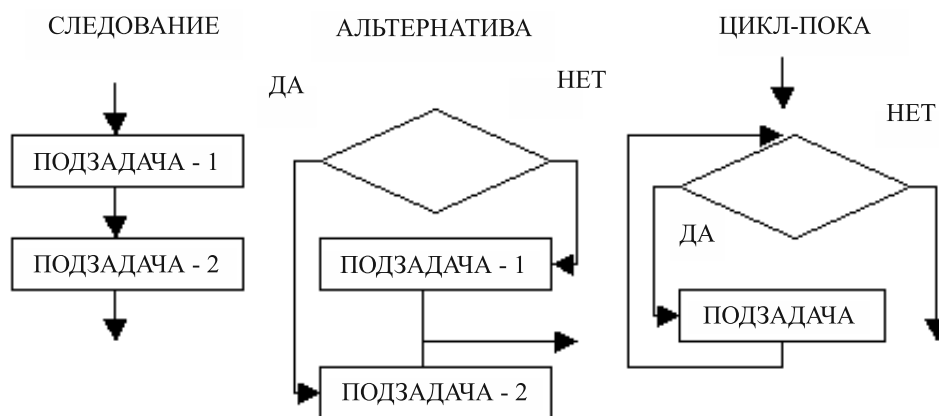


Рис. 6.3 – Основные структуры процедурной декомпозиции

Каждая подзадача, в свою очередь, разбивается на подзадачи с использованием тех же структур; процесс продолжается, пока на очередном уровне не получится подзадача, которая достаточно просто реализуется средствами используемого языка (1–5 управляющих операторов языка). В качестве примера можно привести процедурную декомпозицию программы «записная книжка». Пусть требуется разработать программу, которая в удобной для пользователя форме позволит записывать и затем находить телефоны различных людей и организаций. «Удобная» форма на современном уровне программирования предполагает общение программы с пользователем через «меню». Анализ задачи показывает, что программу можно строить как последовательность подпрограмм. Следовательно, на первом шаге декомпозиции с использованием пошаговой детализации получаем алгоритм:

Основная программа:

- 1 Начать работу.
 - 2 Вывести меню на экран.
 - 3 Ввести команду.
 - 4 Выполнить цикл обработки вводимых команд.
- Завершить работу.

Первые три подзадачи (1–3), выявленные на данном шаге, представляются несложными, поэтому на следующем шаге детализируем действие «Выполнить цикл обработки вводимых команд».



Пример

Выполнить цикл обработки вводимых команд:

цикл-пока команда ≠ «завершить работу»
 Выполнить команду.
 Ввести команду
 все-цикл.

После этого детализируем операцию «Выполнить команду». Выполняем композицию, используя сразу несколько конструкций ветвления.



Пример

Выполнить команду:

если команда = «открыть книжку»
 то Открыть книжку
 иначе если команда = «добавить»
 то Добавить запись
 иначе если команда = «найти»
 то Найти запись
 все-если
 все-если
 все-если.

На этом шаге можно пока остановиться, так как оставшиеся действия достаточно просты. «Вложив» результаты пошаговой детализации, получим структурное представление алгоритма основной программы объемом не более 20...30 операторов.



Пример

Основная программа:

```

Начать работу.
Вывести меню на экран.
Ввести команду.
цикл-пока команда ≠ «завершить работу»
    если команда= «открыть книжку»
    то Открыть книжку
    иначе если команда= «добавить»
    то Добавить запись
        иначе если команда= «найти»
        то Найти запись
        все-если
    все-если
Ввести команду
все-цикл
Завершить работу.
  
```

Окончательно, на первом уровне выделены подзадачи: «Вывести меню», «Ввести команду», «Открыть книжку», «Добавить запись» и «Найти запись». На следующем уровне определяются подзадачи задач второго уровня, например:



Пример

Открыть книжку:

```

Ввести имя файла
если существует файл Имя_книжки
то Открыть файл
иначе Вывести сообщение об ошибке
все-если
  
```

На этом этапе получаем подзадачи: «Ввести имя файла» и «Открыть файл». Поступив аналогично с наиболее сложными подзадачами первого уровня, получаем схему двухуровневой алгоритмической декомпозиции задачи. На рис. 6.4 показано, из каких подпрограмм будет состоять разрабатываемая система и взаимодействие последних по вызовам. Созданная по результатам декомпозиции программа имеет правильную с точки зрения структурной технологии организацию, т. е. включает 12 относительно небольших подпрограмм, вызываемых из основной программы или подпрограмм более высокого уровня. Сформулированная таким

образом методика декомпозиции закрепила сложившийся в то время процедурный или алгоритмический подход к программированию, при котором основное внимание концентрируется на определении последовательности действий.

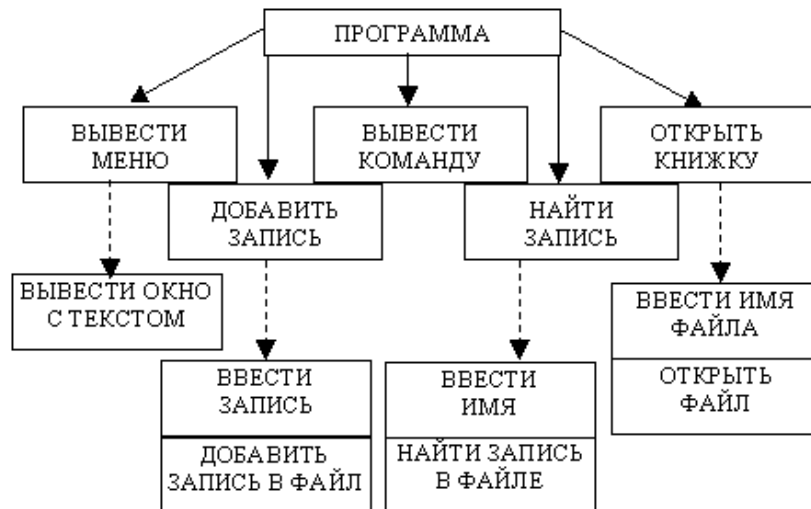


Рис. 6.4 – Алгоритмическая декомпозиция системы «Записная книжка»

Поддержка принципов структурного программирования была заложена в основу так называемых процедурных языков программирования. Как правило, они включали основные «структурные» операторы управления, поддерживали вложение подпрограмм, локализацию и ограничение области «видимости» данных. Среди наиболее известных языков этой группы стоит назвать PL/1, ALGOL-68, Pascal, C.

Дальнейший рост сложности и размеров разрабатываемого программного обеспечения потребовал развития структурирования данных, и соответственно в языках появляется возможность определения пользовательских типов данных. Одновременно усиливается стремление разграничить доступ к глобальным данным программы для уменьшения количества ошибок. Результатом было появление и развитие технологии модульного программирования. Модульное программирование (рис. 6.5) предполагает выделение групп подпрограмм, использующих одни и те же глобальные данные, в отдельно компилируемые модули (библиотеки подпрограмм), например модуль графических ресурсов, модуль подпрограмм вывода на принтер. Связи между модулями осуществляются через специальный интерфейс, в то время как доступ к реализации модуля (телам подпрограмм и некоторым «внутренним» переменным) запрещен. Эту технологию поддерживают современные версии языков Pascal и C (C++), языки Ада и Modula. Использование модульного программирования существенно упрощает ведение разработки программ несколькими программистами, каждый из которых разрабатывает свои модули. Внутренняя организация модулей скрыта от остальных и потому может изменяться независимо. Взаимодействие модулей осуществляется через специально оговоренные интерфейсы модулей. Кроме того, модули в дальнейшем могут использоваться в других разработках, что увеличивает производительность труда программистов.

Практика программирования показывает, что структурный подход в сочетании с модульным программированием позволяет получать достаточно надежные

программы, размер которых не превышает 100 000 операторов. Узким местом модульного программирования является то, что ошибка в интерфейсе при вызове подпрограммы выявляется только при выполнении программы (из-за раздельной компиляции модулей обнаружить эти ошибки раньше невозможно). При увеличении размера программы свыше 100 000 операторов обычно возрастает сложность межмодульных интерфейсов, и предусмотреть взаимовлияние отдельных частей программы становится практически невозможно.

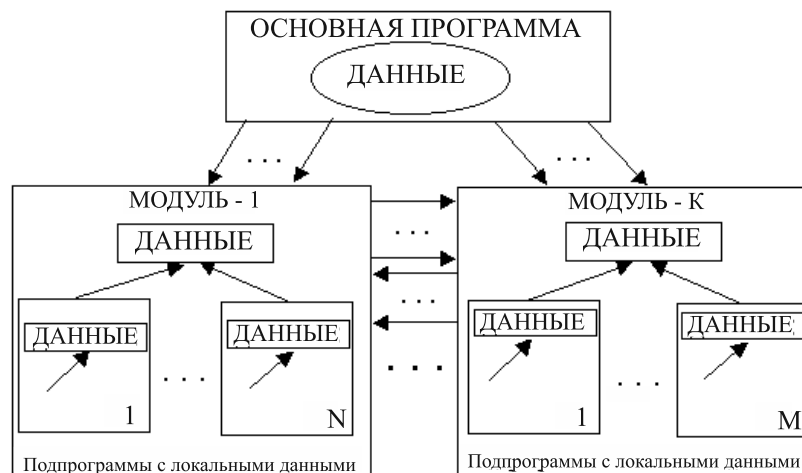


Рис. 6.5 – Архитектура программы, состоящей из модулей

Стремление уменьшить количество связей между отдельными частями программы привело к появлению объектно-ориентированного программирования (ООП).

6.3 Основные принципы и этапы объектно-ориентированного программирования

В теории программирования ООП определяется технология создания сложного программного обеспечения, которая основана на представлении программы в виде совокупности объектов, каждый из которых является экземпляром определенного типа (класса), а классы образуют иерархию с наследованием свойств. Взаимодействие программных объектов в такой системе осуществляется путем передачи сообщений (рис. 6.6). Такое представление программы впервые было использовано в языке имитационного моделирования сложных систем Simula, появившемся еще в 60-х годах. Естественный для языков моделирования способ представления программы получил развитие в другом специализированном языке моделирования — языке Smalltalk (70-е годы), а затем был использован в новых версиях универсальных языков программирования, таких как Pascal, C++, Ада, Modula.

Основное достоинство ООП — сокращение количества межмодульных вызовов и уменьшение объемов информации, передаваемой между модулями, по сравнению с модульным программированием. Это достигается за счет более полной локализации данных и интегрирования их с подпрограммами обработки, что поз-

воляет вести практически независимую разработку отдельных частей (объектов) программы.

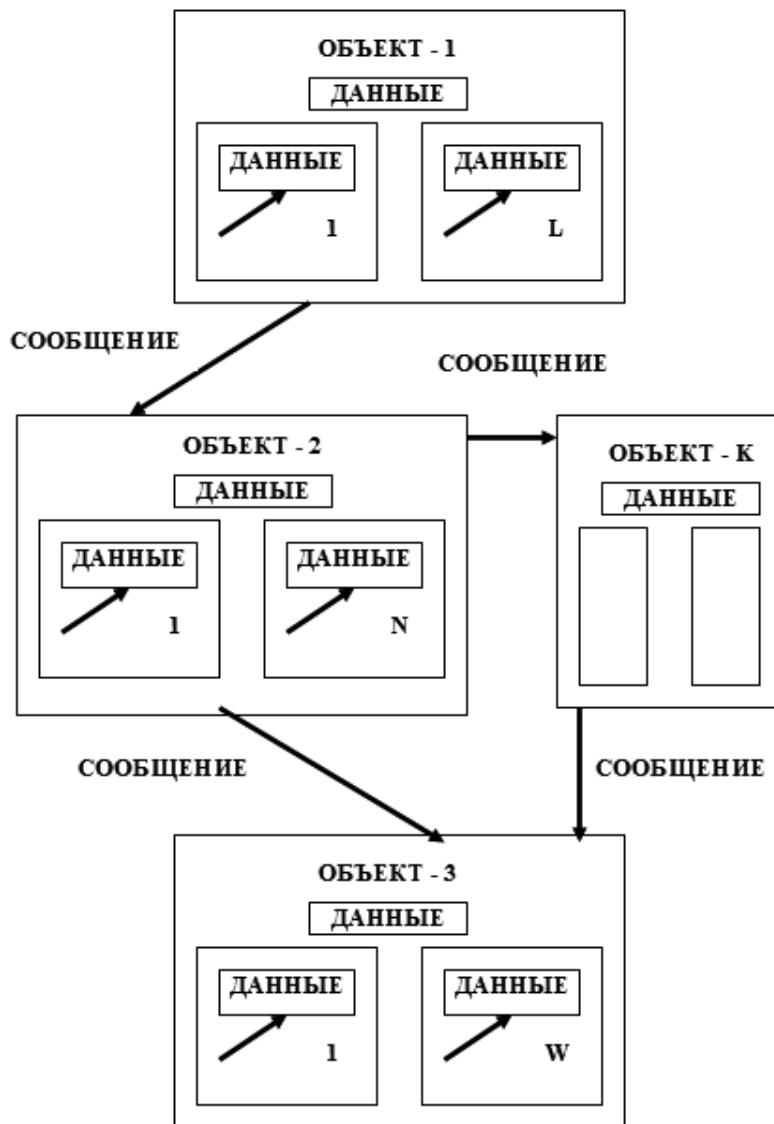
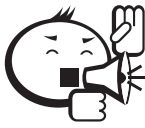


Рис. 6.6 – Архитектура программы при ООП

Кроме этого, объектный подход предлагает новые технологические средства разработки, такие, как наследование, полиморфизм, композиция, наполнение, позволяющие конструировать сложные объекты из более простых. В результате существенно увеличивается показатель повторного использования кодов, появляется возможность создания библиотек объектов для различных применений, и разработчикам предоставляются дополнительные возможности создания систем повышенной сложности.

Основной недостаток ООП — некоторое снижение быстродействия за счет более сложной организации программной системы.



В основу ООП положены следующие принципы:

- абстрагирование,
- ограничение доступа,
- модульность,
- иерархичность,
- типизация,
- параллелизм,
- устойчивость.

Рассмотрим, что представляет собой каждый принцип.



Абстрагирование — процесс выделения абстракций в предметной области задачи. Абстракция — совокупность существенных характеристик некоторого объекта, которые отличают его от всех других видов объектов и, таким образом, четко определяют особенности данного объекта с точки зрения дальнейшего рассмотрения и анализа.

В соответствии с определением применяемая абстракция реального предмета существенно зависит от решаемой задачи: в одном случае нас будет интересовать форма предмета, в другом вес, в третьем — материалы, из которых он сделан, в четвертом — закон движения предмета и т. д. Современный уровень абстракции предполагает объединение всех свойств абстракции (как касающихся состояния анализируемого объекта, так и определяющих его поведение) в единую программную единицу, некий абстрактный тип (класс).



Ограничение доступа — сокрытие отдельных элементов реализации абстракции, не затрагивающих существенных характеристик ее как целого.

Необходимость ограничения доступа предполагает разграничение двух частей в описании абстракции:

- 1) интерфейс — совокупность доступных извне элементов реализации абстракции (основные характеристики состояния и поведения);
- 2) реализация — совокупность недоступных извне элементов реализации абстракции (внутренняя организация абстракции и механизмы реализации ее поведения).

Ограничение доступа в ООП позволяет разработчику:

- выполнять конструирование системы поэтапно, не отвлекаясь на особенности реализаций используемых абстракций;
- легко модифицировать реализацию отдельных объектов, что в правильно организованной системе не потребует изменения других объектов.

Объединение всех свойств предмета (составляющих его состояния и поведения) в единую абстракцию и ограничения доступа к реализации этих свойств получило название инкапсуляции.



.....
Модульность — принцип разработки программной системы, предполагающий реализацию ее в виде отдельных частей (модулей).

При выполнении декомпозиции системы на модули желательно объединять логически связанные части, по возможности обеспечивая сокращение количества внешних связей между модулями. Принцип унаследован от модульного программирования, следование ему упрощает проектирование и отладку программы.



.....
Иерархия — ранжированная или упорядоченная система абстракций.

Принцип иерархичности предполагает использование иерархий при разработке программных систем.

В ООП используются два вида иерархии.

- *Иерархия «целое/часть»* — показывает, что некоторые абстракции включены в рассматриваемую абстракцию как ее части, например лампа состоит из цоколя, нити накаливания и колбы. Этот вариант иерархии используется в процессе разбиения системы на разных этапах проектирования (на логическом уровне — при декомпозиции предметной области на объекты, на физическом уровне — при декомпозиции системы на модули и при выделении отдельных процессов в мультипроцессной системе).
- *Иерархия «общее/частное»* — показывает, что некоторая абстракция является частным случаем другой абстракции, например «обеденный стол — конкретный вид стола», а «столы — конкретный вид мебели». Используется при разработке структуры классов, когда сложные классы строятся на базе более простых путем добавления к ним новых характеристик и, возможно, уточнения имеющихся.

Один из важнейших механизмов ООП — наследование свойств в иерархии общее/частное. Наследование — такое соотношение между абстракциями, когда одна из них использует структурную или функциональную часть другой или нескольких других абстракций (соответственно простое и множественное наследование).



.....

Типизация — ограничение, накладываемое на свойства объектов и препятствующее взаимозаменяемости абстракций различных типов (или сильно сужающее возможность такой замены).

.....

В языках с жесткой типизацией для каждого программного объекта (переменной, подпрограммы, параметра и т. д.) объявляется тип, который определяет множество операций над соответствующим программным объектом. Рассматриваемые далее языки программирования на основе Паскаля используют строгую, а на основе языка С — среднюю степень типизации.

Использование принципа типизации обеспечивает:

- раннее обнаружение ошибок, связанных с недопустимыми операциями над программными объектами (ошибки обнаруживаются на этапе компиляции программы при проверке допустимости выполнения данной операции над программным объектом);
- упрощение документирования;
- возможность генерации более эффективного кода.

Тип может связываться с программным объектом статически (тип объекта определен на этапе компиляции — раннее связывание) и динамически (тип объекта определяется только во время выполнения программы — позднее связывание). Реализация позднего связывания в языке программирования позволяет создавать переменные — указатели на объекты, принадлежащие различным классам (полиморфные объекты), что существенно расширяет возможности языка.

Термины «раннее связывание» и «позднее» относятся к этапу, на котором обращение к процедуре связывается с ее адресом. В случае раннего связывания адреса всех функций и процедур известны в тот момент, когда происходят компиляция и компоновка программы. Это позволяет приписать каждому обращению к процедуре соответствующий адрес. В большинстве традиционных языков, включая Си и Паскаль, используется только раннее связывание. В противоположность этому, в случае позднего связывания адрес процедуры не связывается с обращением к ней до этого момента, пока обращение не произойдет фактически, т. е. во время выполнения программы.

Рассмотрим раннее и позднее связывание на примере с окнами. В такой динамической системе нельзя заранее предсказать, сколько окон будет на экране и каких типов они будут (например, меню, редактор, и т. д.), и в какой последовательности пользователь будет с ними работать. В программе, где используется только раннее связывание, вся информация о количестве и типах окон хранится в основной программе. Все возможные действия пользователя над окнами тоже должны быть предусмотрены в этой программе. Каждый раз, когда пользователь производит какие-то действия с окном (будь то Редактор, Поле приглашения или что угодно), программа должна разобраться, что именно и с каким именно окном произошло, и вызвать соответствующие процедуры для выполнения надлежащих действий с этим окном. Таким образом, программе приходится отслеживать очень многое, она усложняется и теряет гибкость. Стоит добавить один новый тип

окна или изменить поведение уже существующего, как придется корректировать программу во всех тех местах, где определяется, какие подпрограммы подлежат вызову.

Каким же образом можно улучшить положение с помощью позднего связывания? Рассмотрим случай, когда одно окно частично перекрывает другое. Если «верхнее» окно будет передвинуто или закрыто, то «нижнее» следует перерисовать для восстановления ранее перекрытой части.

Так как меню перерисовывается иначе, чем поле или редактор, то каждый объект в оконной иерархии должен знать, как перерисовать себя. Таким образом, в каждом классе будет своя функция-член, которая перерисовывает данный объект на экране. Следовательно, если требуется перерисовать объект, то программе не нужно анализировать, к какому типу окна он относится (как это требовалось при раннем связывании). Она просто вызывает функцию-член данного объекта «ПЕРЕРИСОВКА». Объект исполняет эту свою функцию и корректно перерисовывает себя на экране.



.....
Параллелизм — свойство нескольких абстракций одновременно находиться в активном состоянии, т. е. выполнять некоторые операции.

Существует целый ряд задач, решение которых требует одновременного выполнения некоторых последовательностей действий. К таким задачам, например, относятся задачи автоматического управления несколькими процессами.

Реальный параллелизм достигается только при реализации задач такого типа на многопроцессорных системах, когда имеется возможность выполнения каждого процесса отдельным процессором. Системы с одним процессором имитируют параллелизм за счет деления времени процессора между задачами управления различными процессами. В зависимости от типа используемой операционной системы (одно- или мультипрограммной) деление времени может выполняться либо разрабатываемой системой (как в MS DOS), либо используемой ОС (как в системах Windows).



.....
Устойчивость — свойство абстракции существовать во времени независимо от процесса, породившего данный программный объект, и/или в пространстве, перемещаясь из адресного пространства, в котором он был создан.

Различают:

- временные объекты, хранящие промежуточные результаты некоторых действий, например вычислений;
- локальные объекты, существующие внутри подпрограмм, время жизни которых исчисляется от вызова подпрограммы до ее завершения;
- глобальные объекты, существующие пока программа загружена в память;

- сохраняемые объекты, данные которых хранятся в файлах внешней памяти между сеансами работы программы.

Все указанные выше принципы в той или иной степени реализованы в различных версиях объектно-ориентированных языков.

6.4 Объектно-ориентированные языки программирования

Язык считается объектно-ориентированным, если в нем реализованы первые четыре из рассмотренных выше семи принципов. В теории программирования принято различать объектно-ориентированные и объектные языки программирования. Последние отличаются тем, что они не поддерживают наследования свойств в иерархии абстракций, например Ада — объектный язык, а C++ и объектные версии Паскаля — объектно-ориентированные. Несмотря на то, что принципиально ООП возможно на многих языках программирования, желательно для создания объектно-ориентированных программ использовать объектно-ориентированные языки, включающие специальные средства, например Borland Pascal (начиная с версии 5.5), C++, Delphi и т. д.

Самая простая объектная модель использована при разработке Borland Pascal 7.0. Она специально создавалась для облегчения перехода программистов на использование технологии ООП и не поддерживает абстракции метаклассов, почти не содержит специальных средств сокрытия реализации объектов. Но даже в таком варианте она позволяет создавать достаточно сложные системы. Объектные модели остальных языков являются практически полными. Особое место занимают объектные модели Delphi и C++Builder. Эти модели обобщают опыт ООП для MS DOS и включают некоторые новые средства, обеспечивающие эффективное создание более сложных систем. На базе этих моделей созданы визуальные среды для разработки приложений Windows. Сложность программирования под Windows удалось существенно.

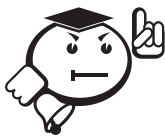
Первый объектно-ориентированный язык программирования Simula 67 был разработан в конце 60-х годов в Норвегии. Авторы этого языка очень точно угадали перспективы развития программирования: их язык намного опередил свое время. Однако современники (программисты 60-х годов) оказались не готовы воспринять ценности языка Simula 67, и он не выдержал конкуренции с другими языками программирования (прежде всего, с языком Fortran). Прохладному отношению к языку Simula 67 способствовало и то обстоятельство, что он был реализован как интерпретируемый (а не компилируемый) язык, что было совершенно неприемлемым в 60-е годы, так как интерпретация связана со снижением эффективности (скорости выполнения) программ.

Но достоинства языка Simula 67 были замечены некоторыми программистами, и в 70-е годы было разработано большое число экспериментальных объектно-ориентированных языков программирования: например, языки CLU, Alphard, Concurrent Pascal и др. Эти языки так и остались экспериментальными, но в результате их исследования были разработаны современные объектно-ориентированные языки программирования: C++, Smalltalk, Eiffel и др. Наиболее распространенным

объектно-ориентированным языком программирования безусловно является C++. Свободно распространяемые коммерческие системы программирования C++ существуют практически на любой платформе.

Разработка новых объектно-ориентированных языков программирования продолжается. С 1995 года стал широко распространяться новый объектно-ориентированный язык программирования Java, ориентированный на сети компьютеров и, прежде всего, на Internet. Синтаксис этого языка напоминает синтаксис языка C++, однако эти языки имеют мало общего. Java-интерпретируемый язык: для него определены внутреннее представление (bytecode) и интерпретатор этого представления, которые уже сейчас реализованы на большинстве платформ. Интерпретатор упрощает отладку программ, написанных на языке Java, обеспечивает их переносимость на новые платформы и адаптируемость к новым программным окружениям. Он позволяет исключить влияние программ, написанных на языке Java, на другие программы и файлы, имеющиеся на новой платформе, и тем самым обеспечить безопасность при выполнении этих программ. Эти свойства языка Java позволяют использовать его как основной язык программирования для программ, распространяемых по сетям (в частности, по сети Internet).

6.5 Этапы разработки программных систем с использованием ООП



.....
Процесс разработки программного обеспечения с использованием ООП включает четыре этапа:

- 1) анализ,
 - 2) проектирование,
 - 3) эволюция,
 - 4) модификация.
-

Анализ



.....
Цель анализа — максимально полное описание задачи.
.....

На этом этапе выполняется анализ предметной области задачи, объектная декомпозиция разрабатываемой системы и определяются важнейшие особенности поведения объектов (описание абстракций). По результатам анализа разрабатывается структурная схема программного продукта, на которой показываются основные объекты и сообщения, передаваемые между ними, а также выполняется описание абстракций.

Проектирование



.....
Различают:

- *логическое проектирование*, при котором принимаемые решения практически не зависят от условий эксплуатации (операционной системы и используемого оборудования);
 - *физическое проектирование*, при котором приходится принимать во внимание указанные факторы.
-

Логическое проектирование заключается в разработке структуры классов: определяются поля для хранения составляющих состояния объектов и алгоритмы методов, реализующих аспекты поведения объектов. При этом используются рассмотренные выше приемы разработки классов (наследование, композиция, наполнение, полиморфизм и т. д.). Результатом является иерархия или диаграмма классов, отражающие взаимосвязь классов, и описание классов.

Физическое проектирование включает объединение описаний классов в модули, выбор схемы их подключения (статическая или динамическая компоновка), определение способов взаимодействия с оборудованием, с операционной системой и/или другим программным обеспечением (например, базами данных, сетевыми программами), обеспечение синхронизации процессов для систем параллельной обработки и т. д.

Эволюция системы



.....
Эволюция системы — это процесс поэтапной реализации и подключения классов к проекту.
.....

Процесс начинается с создания основной программы или проекта будущего программного продукта. Затем реализуются и подключаются классы, так чтобы создать грубый, но, по возможности, работающий прототип будущей системы. Он тестируется и отлаживается. Например, таким прототипом может служить система, включающая реализацию основного интерфейса программного продукта (передача сообщений в отсутствующую пока часть системы не выполняется). В результате мы получаем работоспособный прототип продукта, который может быть, например, показан заказчику для уточнения требований. Затем к системе подключается следующая группа классов, например связанная с реализацией некоторого пункта меню. Полученный вариант также тестируется и отлаживается, и так далее, до реализации всех возможностей системы.

Использование поэтапной реализации существенно упрощает тестирование и отладку программного продукта.

Модификация



.....
***Модификация** — это процесс добавления новых функциональных возможностей или изменение существующих свойств системы.*
.....

Как правило, изменения затрагивают реализацию класса, оставляя без изменения его интерфейс, что при использовании ООП обычно обходится без особых неприятностей, так как процесс изменений затрагивает локальную область. Изменение интерфейса — также не очень сложная задача, но ее решение может повлечь за собой необходимость согласования процессов взаимодействия объектов, что потребует изменений в других классах программы. Однако сокращение количества параметров в интерфейсной части по сравнению с модульным программированием существенно облегчает и этот процесс.

Простота модификации позволяет сравнительно легко адаптировать программные системы к изменяющимся условиям эксплуатации, что увеличивает время жизни систем, на разработку которых затрачиваются огромные временные и материальные ресурсы. Особенностью ООП является то, что объект или группа объектов могут разрабатываться отдельно и, следовательно, их проектирование может находиться на различных этапах. Например, интерфейсные классы уже реализованы, а структура классов предметной области еще только уточняется. Обычно проектирование начинается, когда какой-либо фрагмент предметной области достаточно полно описан в процессе анализа. Рассмотрение основных приемов объектного подхода начнем с объектной декомпозиции.

6.6 Объектная декомпозиция

Как уже упоминалось выше, при использовании технологии ООП решение представляется в виде результата взаимодействия отдельных функциональных элементов некоторой системы, имитирующей процессы, происходящие в предметной области поставленной задачи. В такой системе каждый функциональный элемент, получив некоторое входное воздействие (называемое сообщением) в процессе решения задачи, выполняет заранее определенные действия (например, может изменить собственное состояние, выполнить некоторые вычисления, нарисовать окно или график и, в свою очередь, воздействовать на другие элементы). Процессом решения задачи управляет последовательность сообщений. Передавая эти сообщения от элемента к элементу, система выполняет необходимые действия. Функциональные элементы системы, параметры и поведение которой определяются условием задачи, обладающие самостоятельным поведением (т. е. «умеющие» выполнять некоторые действия, зависящие от полученных сообщений и состояния элемента), получили название объектов. *Процесс представления предметной области задачи в виде совокупности объектов, обменивающихся сообщениями, называется объектной декомпозицией.*

Для того чтобы понять, о каких объектах и сообщениях идет речь при выполнении объектной декомпозиции в каждом конкретном случае, следует вспомнить, что первоначально объектный подход был предложен для разработки имитационных моделей поведения сложных систем. Набор объектов таких систем обычно определяется при анализе моделируемых процессов.

Полученную модель можно реализовать в виде объектно-ориентированной программы. При программировании модели блока колонок, очереди и генератора потока автомашин будут представлены в виде объектов некоторых специально разработанных классов, а управляющая модель — в виде основной программы, инициирующей процесс моделирования. Передача сообщений в системе будет имитироваться как вызовы соответствующих методов объектов. Объектная декомпозиция, так же как и процедурная, может применяться многократно или быть многоуровневой. Это значит, что каждый объект может рассматриваться как система, которая состоит из элементов, взаимодействующих друг с другом через передачу сообщений. При многоуровневой декомпозиции на каждом уровне мы получаем объекты с более простым поведением, что позволяет разрабатывать системы повышенной сложности по частям.

Покажем, как происходит многоуровневая декомпозиция на примере простейшего графического редактора. Выполним объектную декомпозицию программы, которая по запросу пользователя рисует одну из двух фигур: квадрат или круг. При желании пользователь должен иметь возможность изменить цвет контура, размер фигуры и координаты ее центра.

По правилам выполнения объектной декомпозиции разрабатывается имитационная модель программы. Для этого придется проанализировать все происходящие в имитируемой системе процессы и выделить элементы, обладающие собственным поведением, воздействующие на другие элементы и/или являющиеся объектами такого воздействия.



Рис. 6.7 – Диаграмма объектов графического редактора

Основной процесс системы — процесс управления рисованием фигур, указанных пользователем. Все команды пользователя должны интерпретироваться, и в результате интерпретации должны формироваться команды на рисование или изменение параметров фигур. Эти процессы можно моделировать, используя три объ-

екта: Монитор (блок управления, который получает и интерпретирует команды пользователя) и два объекта — фигуры (см. рис. 6.7), каждый со своими параметрами.

Фигуры получают следующие сообщения: «Нарисовать», «Изменить цвет контура», «Изменить размер», «Изменить координаты». Все эти сообщения инициируются Монитором в соответствии с командой пользователя. Получив от пользователя команду «Завершить», Монитор прекращает выполнение программы. Рассмотрим подход объектной декомпозиции к программе «Записная книжка». Попробуем выполнить объектную декомпозицию для программы «Записная книжка», на которой ранее демонстрировалась специфика процедурной декомпозиции (см. рис. 6.4).

Как уже констатировалось ранее, работу с программами такого типа пользователю удобнее осуществлять через меню. Значит, при запуске программы на экране должно появиться меню, содержащее набор возможных операций. В зависимости от выбора пользователя далее будут активизироваться части программы, ответственные за открытие «книжки», добавление записей или за поиск записей (рис. 6.8).

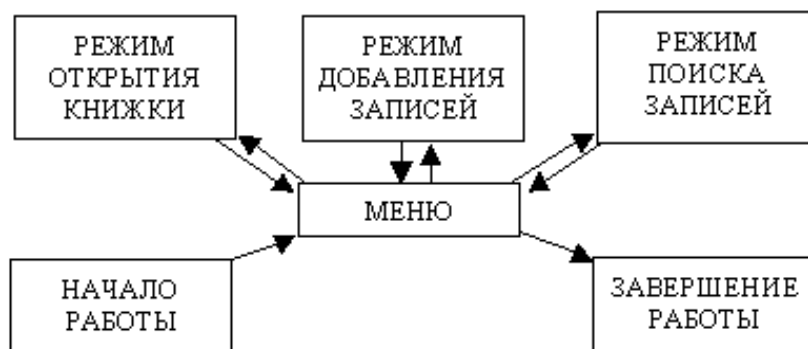


Рис. 6.8 – Диаграмма состояний интерфейса пользователя (первый вариант)

На этом этапе анализа мы уже можем выделить четыре самостоятельных части программы, которые взаимодействуют в процессе ее выполнения: Меню, Открытие книжки, Ввод записей, Поиск записей. Часть Меню является управляющей и активизирует в процессе работы с программой остальные части для выполнения требуемых операций. Добавим к системе еще несколько видимых объектов — сообщений пользователю, которые будут появляться на экране при обнаружении несоответствий в процессе работы, например «Записная книжка не найдена» — активизируется Открытием книжки или «Нет информации об абоненте» — активизируется Поиском записей (рис. 6.9).

Объекты Меню, Открытие книжки, Ввод записей, Поиск записей в процессе работы должны получать информацию от пользователя и сообщать ему результаты работы и соответственно должны иметь некоторое экранное представление. Совокупность таких экранных представлений (форм) образует интерфейс с пользователем.

Помимо объектов интерфейса система содержит еще, по крайней мере, один объект — Файл записей, используемый для хранения введенной информации. Этот объект должен получать от Открытия, Ввода и Поиска сообщения-команды соответственно открытия файла, добавления информации и поиска. Команда открытия

должна сопровождаться именем файла, команда добавления информации — идентификационным заголовком и телефоном, а команда поиска — идентификационным заголовком.

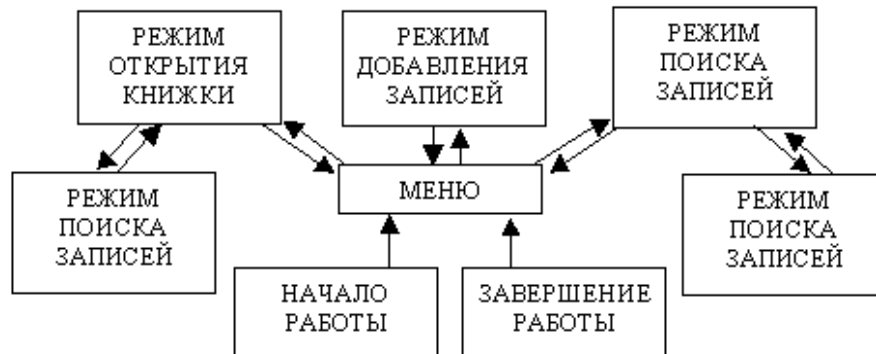


Рис. 6.9 – Полная диаграмма состояний интерфейса пользователя

Объект Меню будет отвечать за выбор пунктов меню и вызов формы Открытие, формы Ввода и формы Поиска. Форма Открытие должна осуществлять открытие или создание файла. Она будет вводить имя файла и вызывать объект Файл для выполнения операции открытия. Форма Ввод должна вводить данные (фамилию, имя, отчество и телефон) и передавать эту информацию объекту Файл для сохранения. Форма Поиск должна вводить данные для поиска (фамилию или имя, или и то и другое сразу) и запрашивать поиск телефона по введенным данным.

Все операции с файлом будет выполнять объект Файл. Окончательный вариант объектной декомпозиции проектируемой системы представлен на рис. 6.10 (сравните с результатами процедурной декомпозиции для той же программы, приведенными на рис. 6.4).

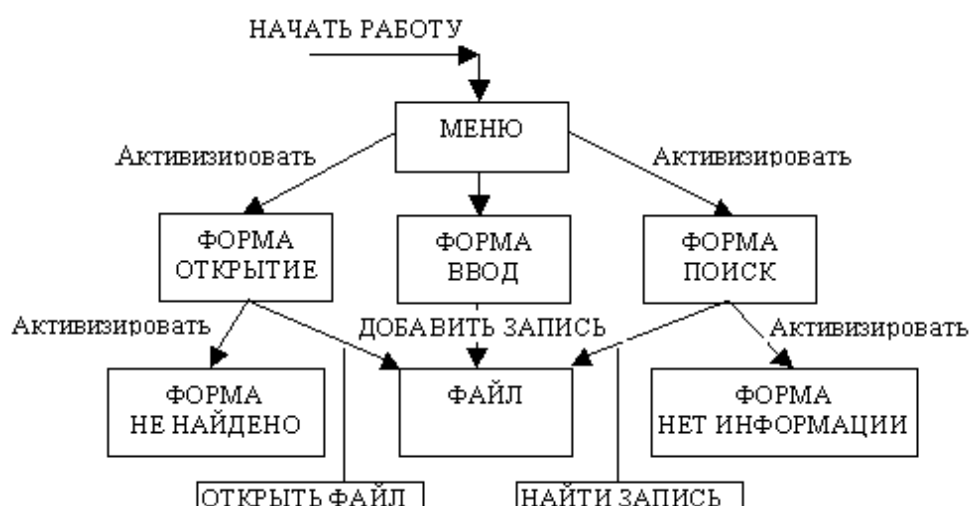


Рис. 6.10 – Диаграмма объектов системы «Записная книжка»

Таким образом, можно сформулировать следующие рекомендации по выполнению объектной декомпозиции:

- 1) Для сложных систем объектная декомпозиция должна выполняться поэтапно: на первом этапе — объектная декомпозиция всей системы, на последующих — декомпозиция объектов как подсистем.
- 2) При декомпозиции системы в целом в качестве объектов могут выделяться элементы двух типов:
 - элементы интерфейса пользователя (окна меню, окна сообщений, окна форм ввода-вывода и т. д.);
 - средства хранения, организации и преобразования данных (базы данных, файлы, протоколы, структуры данных и т. д.).

При этом для каждого объекта должно определяться множество получаемых и передаваемых сообщений и основные характеристики. Процесс декомпозиции прекращается при получении объектов, которые могут быть достаточно просто реализованы, т. е. имеют четко определенную структуру и поведение.

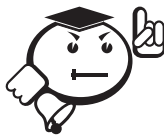
6.7 Объекты и сообщения

В предыдущем разделе было показано, что под объектом применительно к ООП понимается отдельно реализуемая часть предметной области задачи. Разрабатываемая программа, таким образом, состоит из объектов, которые взаимодействуют через передачу сообщений. Каждый объект, получая сообщения, должен определенным образом «реагировать» на них, выполняя заранее определенные для каждого типа сообщения действия. Например, если объект Форма Открытие из примера «Записная книжка» будет активизирован, то он должен запросить у пользователя имя файла, проверить наличие такого файла и затем либо открыть его, послав соответствующее сообщение объекту Файл, либо активизировать объект Сообщение «Записная книжка не найдена». Реакция объекта на сообщение может зависеть от его состояния: так объект Файл, получив сообщение «Добавить запись», прежде чем добавлять запись, должен проверить, открыт ли соответствующий файл, и при закрытом файле должен выдать пользователю отказ на выполнение операции добавления.

Состояние объекта характеризуется набором конкретных значений некоторого перечня всех возможных свойств данного объекта, например (пример графического редактора, см. рис. 6.7) характеризуется: скоростью передачи данных, типом графического режима (количество цветов и тип монитора и др.). Как правило, набор свойств в процессе функционирования не изменяется, могут изменяться лишь конкретные значения этих свойств. Поведение объектов, как уже говорилось выше, характеризуется определенным набором реакций на получаемые сообщения и часто зависит от состояния объекта. Объекты, обладающие сходными состояниями и поведением с точки зрения решаемой задачи, образуют группу (например, объекты записной книжки). Свойства объекта, которые отличают его от всех других объектов группы, составляют его индивидуальность. Так каждая функция Записной книжки может иметь собственную структуру действий. Если объект может обладать некоторым состоянием, то, соответственно, может возникнуть необходимость в получении информации об этом состоянии. Для получения такой инфор-

мации объекту посылается сообщение-запрос. В ответ на запрос объект должен переслать отправителю требуемую информацию. В таких случаях говорят, что над объектом выполнена операция селекции.

Обращение к объекту для изменения его состояния по всем или отдельным составляющим инициирует выполнение операции модификации. Отправитель сообщения-команды, реакцией на которую должна быть модификация объекта, может ожидать завершения операции, а может продолжить выполнение своей программы. Второй вариант обработки называется асинхронным, и его реализация требует использования (или моделирования) параллельной обработки. Если объект содержит несколько однотипных компонент, например массив чисел, то операция, требующая последовательной обработки этих компонент, называется итерацией. Поэлементно могут выполняться как операции селекции, так и операции модификации.



.....
Полный список возможных операций над объектами выглядит следующим образом (рис. 6.11):

- создание объекта;
- уничтожение объекта;
- модификация — изменение состояния объекта; селекция — доступ для определения состояния;
- итерация — доступ к содержимому объекта по частям в определенной последовательности (используется при наличии в объекте некоторых однотипных компонент). Соответственно, каждое сообщение, принимаемое объектом, может инициировать выполнение одной или нескольких операций указанных типов.

.....
Например, «Активизировать» для интерфейсного элемента, в зависимости от типа реализации (статической или динамической) может означать либо создание объекта — при динамической реализации, либо его модификацию — «видимый» вместо «невидимый» при статической реализации.

При выполнении объектной декомпозиции устанавливаются отношения между объектами, которые бывают двух типов. В случае, когда один объект передает сообщение другому, говорят, что эти объекты находятся в отношении использования (рис. 6.12), причем объект, инициирующий сообщение, называется активным, а объект, получающий сообщение, — пассивным. Соответственно отношение использования может принимать форму воздействия (активный объект воздействует на пассивный объект, передавая ему сообщение), исполнения (пассивный объект исполняет указание активного объекта) и посредничества (некоторый объект-посредник, получив сообщение от активного объекта, передает его пассивному объекту).

Если объект является результатом декомпозиции более сложного объекта, то говорят, что между этими объектами существует отношение включения — первый объект включает второй (иерархия целое/часть). Виды операций над объектами и типы отношения между ними определяют особенности реализации объектов.

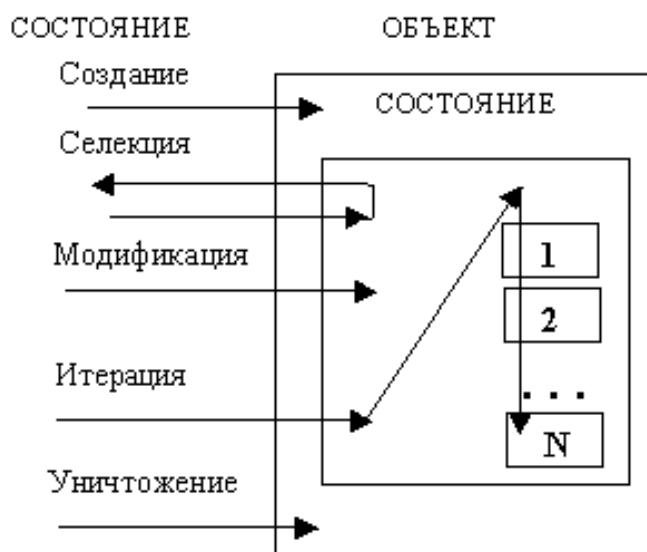


Рис. 6.11 – Типы операций над объектом

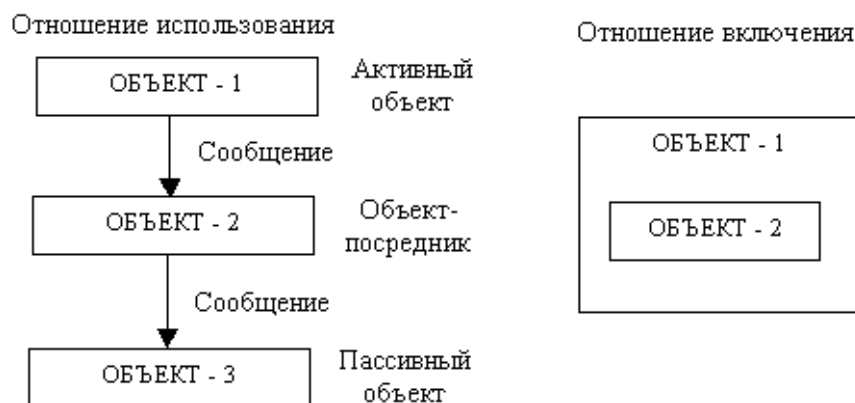


Рис. 6.12 – Типы отношений между объектами

6.8 Классы

Реализация объектов, полученных в результате декомпозиции, принципиально возможна на любом языке, даже на ассемблере. Однако наличие специальных средств позволяет существенно упростить программирование, дополнительно обеспечивая программиста заготовками классов из библиотек и встроенными механизмами обеспечения требуемых свойств. Для представления абстракций объектов используется специальный определяемый программистом тип данных — класс. Класс — это структурный тип данных, который включает описание полей данных, а также процедур и функций, работающих с этими полями данных. Применительно к классам такие процедуры и функции получили название методов.

Реализация объединения данных с определенными видами их обработки делает классы пригодными для описания состояния и поведения моделей реальных объектов. Совокупность полей определяется множеством аспектов состояния объ-

екта с точки зрения решаемой задачи, а совокупность методов — множеством аспектов поведения объекта (см. рис. 6.13). В программах используются переменные типа класса. Такие переменные принято называть объектами. Рассмотрим пример — описание класса Окно. Пусть необходимо разработать класс, переменные которого используются для изображения на экране цветного прямоугольника заданного цвета и размера (см. рис. 6.14). Проектируемый класс должен содержать поля для сохранения параметров окна: $X1$, $Y1$, $X2$, $Y2$ — координаты верхнего левого и нижнего правого углов и $Color$ — цвет.

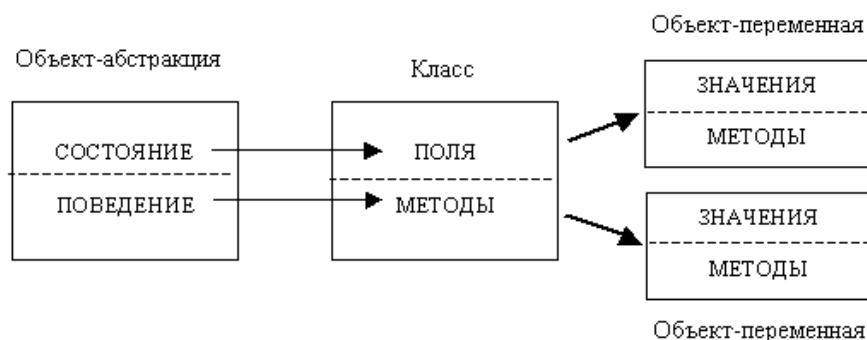


Рис. 6.13 – Соответствие объекта-абстракции классу и объектам-переменным

Пусть единственное сообщение, обрабатываемое объектом, будет — «Нарисовать окно». Соответственно класс должен содержать метод, реализующий процесс рисования объекта. Поля объекта можно инициализировать при создании переменной-объекта, передав ему сообщение инициализации, включающее значения полей. Следовательно, класс должен содержать метод инициализации.

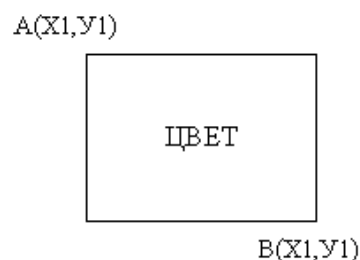


Рис. 6.14 – Вид и параметры окна на экране

Окончательно получаем класс, имеющий следующую структуру:

Класс Окно:

поля $X1, Y1, X2, Y2, Color$

метод Инициализировать($aX1, aY1, aX2, aY2, aColor$)

метод Изобразить

Конец описания.

Создавая объекты типа Окно, инициализируя их в соответствии с условием и посылая им сообщение «Нарисовать окно», получим разные окна на экране,

причем параметры этих окон будут храниться в объектах. Каждая переменная типа класса включает набор полей, объявленных в классе. Совокупность значений, содержащихся в этих полях, моделирует конкретное состояние объекта предметной области. Изменение этих значений в процессе работы отражает изменение состояния моделируемого объекта.

Воздействие на объект выполняется посредством изменения его полей или вызова его методов. Доступ к полям и методам объекта осуществляется, за исключением специальных случаев, с указанием имени объекта (при этом используются составные имена):

<имя объекта>.<имя поля>

или

<имя объекта>.<имя метода>.

Все методы объекта обязательно имеют доступ ко всем полям своего объекта. В рассматриваемых далее языках программирования это достигается через неявную передачу в метод специального параметра — адреса области данных конкретного объекта (Self — в Паскале и this — в C++). Таким образом, уменьшается количество параметров, явно передаваемых в метод.

Ограничение доступа. Большинство версий объектно-ориентированных языков позволяет ограничить доступ к некоторым полям и методам объекта, обеспечивающим функционирование «внутренностей» объекта. При наличии таких возможностей специальными средствами выделяют интерфейс и реализацию класса. Описание класса без учета синтаксиса конкретного языка выглядит следующим образом:



Пример

Класс <имя_класса>

интерфейс

<объявление полей и методов класса,
к которым возможно обращение извне>

реализация

<объявление полей и методов класса,
к которым невозможно обращение извне>

Конец описания.

Как уже говорилось выше, объединение полей данных и процедур и функций, работающих с этими данными, в единый пакет при наличии специальных правил доступа к элементам пакета называется инкапсуляцией.

Наличие интерфейса обеспечивает уменьшение возможности «разрушения» (несанкционированного изменения значений полей) объекта извне. При этом сокрытие особенностей реализации упрощает внесение изменений в реализацию

класса как в процессе отладки, так и при модификации программы. Таким образом, класс определяет существование глобальной области данных внутри объекта, доступной методам объекта.

С другой стороны, доступ к объекту регламентируется и должен выполняться через специальный интерфейс (см. рис. 6.15). Как и любая переменная программы, объект должен быть размещен в памяти (создан) и удален из памяти (уничтожен). Причем создание и уничтожение объектов выполняется статически или динамически. *Статическое создание объектов* выполняется в процессе компиляции программы, а статическое уничтожение — при завершении программы: объект удаляется из памяти вместе с программой. *Динамическое создание и уничтожение объектов* выполняется в процессе работы программы специальными командами.



Рис. 6.15 – Интерфейс и реализация класса

Операция создания и инициализация полей объекта получила название — конструирование объекта, а операция уничтожения объекта — деструкции объекта. Соответствующие методы, если они определены в классе, получили название конструкторов и деструкторов. Конкретные особенности этих методов в различных языках программирования будут обсуждаться в соответствующих разделах. Рассмотрим сокрытие реализации класса (класс Файл) на примере «Записной книжки». В соответствии с результатами объектной декомпозиции (см. рис. 6.10) Файл должен реагировать на следующие сообщения: «Открыть», «Добавить запись», «Найти запись».

Каждое сообщение должно дополняться необходимой информацией. Так, сообщение «Открыть» должно сопровождаться передачей имени файла. Сообщение «Добавить запись» должно сопровождаться передачей текста записи, состоящей из двух частей: фамилии абонента или названия организации и телефона. Сообщение «Найти запись» должно сопровождаться передачей фамилии абонента или названия организации, по которым должен быть найден телефон. Каждому сообщению должен соответствовать метод, объявленный в интерфейсной части класса и реализующий требуемые действия.

Метод Открыть должен выполнять открытие указанного файла для выполнения операций ввода-вывода. Операция открытия файла требует указания файловой переменной. Поскольку при выполнении операций с этим файлом в дальнейшем должна будет использоваться та же файловая переменная, целесообразно описать ее в секции реализации класса, где она будет доступна методам данного объекта, но не доступна из других объектов программы.

Метод Добавить должен проверять наличие открытого файла и сообщать пользователю о невозможности выполнения операции с закрытым файлом. Если же файл открыт, метод должен выполнить добавление записей в файл. Для выполнения проверки необходимо хранить информацию об открытии файла, для этого можно использовать специальное поле Состояние_файла, которое также можно объявить в секции реализации. Метод Найти также вначале должен проверять, открыт ли файл. Если файл открыт, метод выполняет поиск информации в файле. Операции проверки открытия файла выполняются в методах Добавить и Найти идентично, их можно выделить в отдельный метод Проверка_открытия, который можно описать в секции реализации.

Для инициализации поля Состояние_файла (в исходном состоянии файл закрыт) можно использовать специальный метод, традиционно с этой целью используется конструктор. При завершении программы файл необходимо закрыть. Операция закрытия может быть описана в деструкторе.

Окончательно для реализации объекта Файл можно предложить класс следующей структуры:



Пример

Класс Файл:

интерфейс

конструктор Инициализировать;

метод Открыть (имя файла);

метод Добавить (фамилия, телефон);

метод Найти (фамилия);

деструктор Закрыть_файл;

реализация

поле Файловая_переменная;

поле Состояние_файла;

метод Проверка_открытия;

Конец описания.

После определения структуры класса должны быть разработаны алгоритмы методов и назначены типы полей и передаваемых параметров.

В программе должна быть использована переменная типа Файл. Так как файл в программе используется постоянно, соответствующую переменную лучше определять статически. Передача сообщений объекту будет реализована как вызов соответствующего метода интерфейсной части.

6.9 Основные средства разработки классов

Языки, поддерживающие ООП, существенно облегчают разработчику создание новых классов за счет реализации механизмов наследования, композиции, наполнения и полиморфизма.

Наследование

В ООП существует возможность конструирования новых более сложных классов из уже имеющихся посредством добавления полей и определения новых методов (принцип иерархичности). При этом исходный класс, на базе которого выполняется конструирование, часто называется родителем, а производный — потомком. Специальный механизм наследования обеспечивает классу-потомку возможность использования полей и методов одного или нескольких родительских классов. Если непосредственный родитель единственный, то наследование называется простым, а если таких классов несколько — то множественным. При этом класс-родитель (или классы-родители) и класс-потомок образуют иерархию (см. рис. 6.16).

Наличие механизма наследования в языке программирования позволяет повторно не определять в классе уже описанные параметры и свойства объектов, производный класс их просто «наследует».



Рис. 6.16 – Иерархия классов при различных видах наследования

Рассмотрим пример — наследование, класс `Окно_меняющее_цвет`. Построим на базе класса `Окно` класс-потомок, который может изменять цвет окна на экране. Для этого к родительскому классу достаточно добавить метод `Изменить_цвет`:

.....  **Пример**

Класс `Окно_меняющее_цвет` — родитель: класс `Окно`:
 метод `Изменить_цвет(aColor);`
Конец описания.

.....

Класс `Окно_меняющее_цвет` содержит все поля родительского класса и все его методы. Дополнительно объекты типа `Окно_меняющее_цвет` могут менять цвет окна на указанный в сообщении «Изменить_цвет» (см. рис. 6.17).

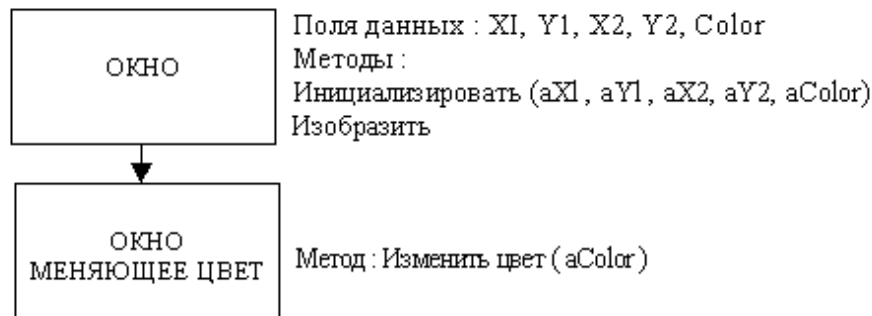


Рис. 6.17 – Иерархия классов `Окно` и `Окно_меняющее_цвет`

При множественном наследовании, реализованном, например, в C++, наследуются поля и методы всех родителей. В том случае, если среди родителей есть классы, принадлежащие одной иерархии, происходит дублирование полей и методов, наследуемых от общих родителей. Для того чтобы избежать неоднозначности, в C++ введено понятие виртуального наследования, при использовании которого виртуально наследуемые поля и методы не дублируются.

Таким образом, в иерархическом дереве классов по мере удаления от корня мы встречаем все более сложные классы, экземплярами которых будут объекты с более сложной структурой и поведением. Наследование свойств в иерархии существенно упрощает работу программиста. В настоящее время созданы библиотеки наиболее часто встречающихся классов, которые можно использовать вновь и вновь, строя на их основе классы для решения различных задач.

Полиморфизм

Простой полиморфизм. При создании иерархии классов может обнаружиться, что некоторые свойства объектов, сохраняя название, изменяются по сути. Для реализации таких иерархий в языке программирования должен быть предусмотрен **полиморфизм**, обеспечивающий возможность задания различных реализаций некоторого единого по названию метода для классов различных уровней иерархии. В ООП такой полиморфизм называется простым, а методы, имеющие одинаковое название, — статическими полиморфными. Совокупность полиморфных методов с одним именем для иерархии классов образует единый полиморфный метод иерархии, в котором реализация полиморфного метода для конкретного класса представляет отдельный аспект. Заметим, что термин «полиморфизм» в программировании, в соответствии со своим изначальным смыслом («многообразие»), используется для обозначения встроенного механизма определения соответствия кода функции типу параметров. Такой механизм реализуется не только в средствах ООП. Различают несколько терминов, связанных с конкретными механизмами реализации полиморфизма для различных случаев.

Чистый полиморфизм — используется для обозначения того, что один код функции может по-разному интерпретироваться в зависимости от типа аргументов; используется в языках высокого уровня абстракции, например в языке LISP или SMALLTALK.

Перегрузка (полиморфные имена функций) — используется, когда определяется несколько функций с одним именем — одно и то же имя функции может многократно использоваться в разных местах программы; выбор нужной функции может определяться типами аргументов, областью видимости (внутри модуля, файла, класса и т. д.); если выбор определяется типом аргументов, то перегрузка называется параметрической; например, язык C++ позволяет разработчику выполнять параметрическую перегрузку функций вне классов.

Переопределение (простой полиморфизм) — используется в ООП при наличии различных определений методов в иерархии классов, конкретный метод определяется типом объекта при компиляции программы (раннее связывание), методы называются статическими полиморфными.

Полиморфные объекты (сложный полиморфизм) — используются в ООП при наличии различных определений методов в иерархии классов — конкретный метод также определяется типом объекта, но в процессе выполнения программы (позднее связывание), методы называются виртуальными полиморфными (рассмотрены далее).

Обобщенные функции или шаблоны — используются в ООП при реализации в языке параметризованных классов (например, в C++), параметрами такого класса являются типы аргументов методов класса (рассмотрены далее).

Рассмотрим простой полиморфизм на примере — класс `Окно_с_текстом`. Пусть необходимо разработать на базе класса `Окно` класс `Окно_с_текстом`. Для этого к полям класса `Окно` необходимо добавить специальные поля для определения координат первой буквы текста — `Xt`, `Yt` и поле, содержащее сам текст — `Text`. Кроме этого, понадобится специальный метод, который будет обрабатывать сообщение «Нарисовать». Однако у нас уже существует родительский метод `Изобразить`, который обрабатывает это сообщение. Следовательно, необходимо заменить родительский метод методом потомка. Механизм полиморфизма позволяет для класса-потомка `Окно_с_текстом` предусмотреть собственный метод `Изобразить` (см. рис. 6.18). Метод `Инициализировать` также должен быть переопределен, так как он должен инициализировать дополнительные поля класса.

Класс `Окно_с_текстом` — родитель: класс `Окно`:

поля `Xt`, `Yt`, `Text`

метод `Инициализировать` (`aX1`, `aY1`, `aX2`, `aY2`, `aColor`,
`aXt`, `aYt`, `aText`)

метод `Изобразить`

Конец описания.

Примечание. При реализации методов `Изобразить` и `Инициализировать` потомка можно вызвать соответствующие родительские методы, а затем добавить операторы, определяющие собственные действия метода для разрабатываемого класса.

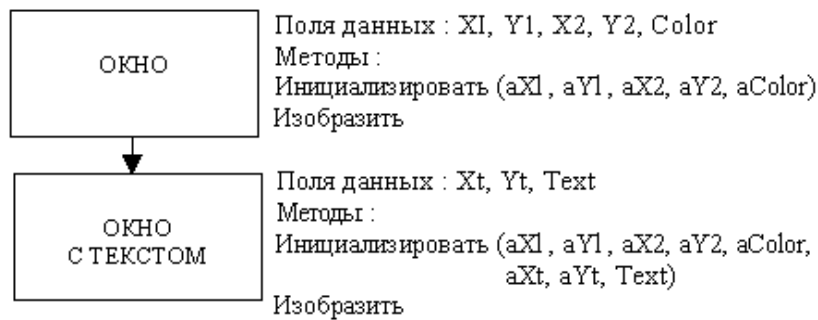


Рис. 6.18 – Иерархия классов Окно и Окно_с_текстом

Сложный полиморфизм или создание полиморфных объектов

Полиморфными объектами или полиморфными переменными называются переменные, которым в процессе выполнения программы может быть присвоено значение, тип которого отличается от типа переменной. Такая ситуация может возникнуть:

- при передаче объекта типа класса-потомка в качестве фактического параметра подпрограмме, в которой этот параметр описан как параметр типа класса-родителя (явно — в списке параметров или неявно — в качестве внутреннего параметра, используемого при вызове методов — `self` или `this`);
- при работе с указателями, когда указателю на объект класса-родителя присваивается адрес объекта класса-потомка.

Тип полиморфного объекта становится известным только на этапе выполнения программы. Соответственно при вызове полиморфного метода для такого объекта нужный аспект также должен определяться на этапе выполнения. Для этого в языке должен быть реализован механизм позднего связывания, позволяющий определять тип объекта и соответственно аспект полиморфного метода, к которому идет обращение в программе на этапе ее выполнения.

С помощью механизма позднего связывания реализуется оперативная перестройка программы в соответствии с типами используемых объектов. Поясним сказанное на примере. Пример — сложный полиморфизм. Пусть родительский класс содержит два метода `Out` и `Print`, один из которых вызывает другой. Класс-потомок наследует метод `Out`, но имеет собственный метод `Print`. Метод `Out` наследуется и может быть вызван как для объекта класса-родителя, так и для объекта класса-потомка (см. рис. 6.19).

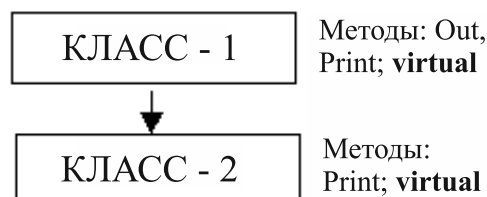


Рис. 6.19 – Иерархия классов для примера сложного полиморфизма.

При вызове метода Out для объекта класса-потомка необходимо обеспечить, чтобы этот метод вызывал метод Print потомка, а не родителя (см. рис. 6.20).



Рис. 6.20 – Пример необходимости позднего связывания

Определить, для объекта какого класса: родителя или потомка вызывается метод Out, можно только на этапе выполнения. Следовательно, для метода Print необходимо обеспечить позднее связывание. Если определение адреса метода Print происходило бы на этапе компиляции программы, то и для объекта родительского класса, и для объекта класса-потомка из метода Out вызывался бы метод Print класса-родителя. Описание метода Print как метода, для которого запрещается определение адреса на этапе компиляции, приведет к тому, что адрес метода Print будет определяться в процессе выполнения программы. В этот момент уже будет известно, объект какого класса вызывает метод Out, и будет вызываться метод Print именно этого класса.

Методы, для которых должно реализовываться позднее связывание, получили название *виртуальных*. Для их описания в рассматриваемых далее языках программирования используется служебное слово *virtual*. Следует отметить, что методы, работающие с полиморфными объектами, — это всегда методы классов-родителей, описывающие общие моменты поведения объектов. В сложной иерархии, таким образом, можно выделить семейство классов со схожим поведением объектов. Они образуют поддерево, в корне которого находится класс, определяющий общие моменты поведения.

Реализация механизма позднего связывания осуществляется с использованием специальной таблицы, получившей название таблицы виртуальных методов (ТВМ). ТВМ создается для каждого класса, имеющего собственные или наследующего виртуальные методы. Она содержит АВМ (адреса виртуальных методов) (см. рис. 6.21). Объекты таких классов содержат адрес ТВМ своего класса. При вызове виртуального метода для объекта происходит обращение к ТВМ класса, по которой и определяется требуемый метод.

При использовании полиморфных объектов возникают проблемы с доступом к полям объекта, описанным в классе-потомке: указатель на объект класса-родителя связан с описанием полей класса-родителя, и поля, описанные в классе-потомке, для него «невидимы». В таких случаях приходится средствами используемого языка явно переопределять тип объекта.

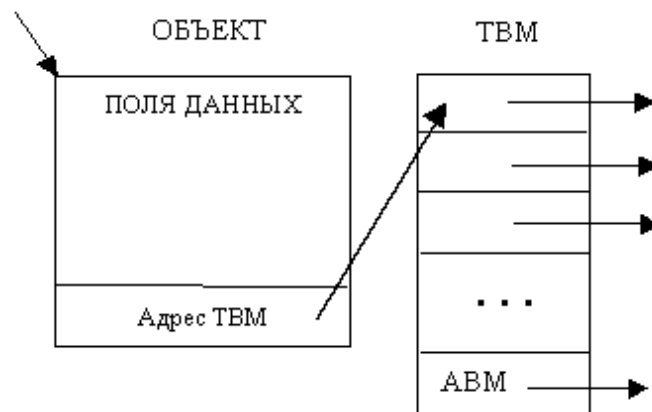


Рис. 6.21 – Реализация механизма позднего связывания

Композиция

Ранее было указано, что в результате объектной декомпозиции второго и более уровней могут быть получены объекты, находящиеся между собой в отношении включения (см. рис. 6.12). Классы для реализации таких объектов могут строиться двумя способами: 1) с использованием наследования или 2) композиции. Применение наследования эффективно в том случае, если разрабатываемый класс имеет с исходным сходную структуру и элементы поведения, например `Окно` и `Окно_меняющее_цвет`. В тех случаях, когда сходное поведение не просматривается или наследование по каким-либо причинам нецелесообразно, можно использовать композицию классов.

Композицией называется такое отношение между классами, когда один является частью второго. Конкретно, композиция реализуется включением в класс поля, являющегося объектом другого класса. Такие поля обычно называют объектными полями. Рассмотрим применение композиции на примере класса `Сообщение` (пример `Окно_с_текстом`). Визуально сообщение обычно выглядит как окно с текстом и кнопкой подтверждения (см. рис. 6.22).



Рис. 6.22 – Вид окна сообщения

При разработке класса `Сообщение` попытаемся использовать уже написанный ранее класс `Окно_с_текстом`. `Окно Сообщение` без кнопки представляет собой объект класса `Окно_с_текстом`. Кнопка также представляет собой `Окно_с_текстом`. Попытка использования при разработке множественного наследования приведет к дублированию полей данных. Чтобы этого избежать, используем для хранения параметров изображения кнопки поле типа `Окно_с_текстом`.



Пример

Класс Сообщение может быть записан так:

Класс Сообщение — родитель: класс Окно_с_текстом:

Поле Кнопка: Окно_с_текстом

Метод Инициализировать (aX1,aY1,aX2,aY2,aColor,
aXt, aYt, aText,
bX1,bY1,bX2,bY2,bColor,
bXt, bYt, bText)

метод Изобразить

Конец описания.

Метод «Инициализировать» класса потомок при этом должен получить двойной список параметров по сравнению с методом «Инициализировать» класса родитель. Первым набором инициализируются родительские поля, а вторым — аналогичные поля включенного объекта. Метод Изобразить должен выводить на экран оба окна: одно — используя родительский метод и второе — используя метод поля-объекта. Доступ к компонентам объектного поля осуществляется с указанием имени объекта, имени поля и имени компонента:

<имя объекта>.<имя поля>.<имя компонента>

или

<имя объекта>.<имя поля>.<имя метода>

Наполнение

Включение объектов в некоторый класс можно реализовать и с использованием указателей на эти объекты. В отличие от объектного поля, которое включает в класс точно указанное количество объектов (1 или более — при использовании массива объектов) конкретного класса, использование указателей позволяет включить 0 и более объектов, если они собраны в массив или списковую структуру (линейную или нелинейную). Механизм наполнения в основном используется для подключения объекта или структуры объектов к некоторому классу, реализующему управление сразу всей структурой.



Контрольные вопросы по главе 6

- 1) Что такое парадигмы программирования?
- 2) В чем заключается парадигма объектно-ориентированного программирования?
- 3) В чем заключается парадигма функционального программирования?
- 4) В чем заключаются основные принципы и этапы объектно-ориентированного программирования?
- 5) Какие принципы положены в основу ООП?
- 6) Что такое принцип параллелизма?
- 7) Что такое принцип устойчивости?
- 8) Что такое объектная декомпозиция?
- 9) Как связаны объекты и сообщения?
- 10) Какие типы операций над объектом вы знаете?

Глава 7

ТЕХНОЛОГИЯ ООП ПРОГРАММИРОВАНИЯ

В соответствии с обычным значением слова «технология» под *технологией программирования* будем понимать совокупность производственных процессов, приводящую к созданию требуемого программного средства (ПС), а также описание этой совокупности процессов. Другими словами, технологию программирования мы будем понимать здесь в широком смысле как технологию разработки *программных средств*, включая в нее все процессы, начиная с момента зарождения идеи этого средства, и в частности, связанные с созданием необходимой программной документации. Каждый процесс этой совокупности базируется на использовании каких-либо методов и средств, например компьютер (в этом случае будем говорить о *компьютерной* технологии программирования).

В литературе имеются и другие, несколько отличающиеся, определения технологии программирования. Используется в литературе и близкое к технологии программирования понятие *программной инженерии*, определяемой как систематический подход к разработке, эксплуатации, сопровождению и изъятию из обращения программных средств. Главное различие между технологией программирования и программной инженерией как дисциплинами для изучения заключается в способе рассмотрения и систематизации материала.

В технологии программирования акцент делается на изучении процессов разработки ПС (*технологических процессов*) и порядке их прохождения — методы и инструментальные средства разработки ПС *используются* в этих процессах (их применение и образуют технологические процессы). Тогда как в программной инженерии изучаются различные методы и инструментальные средства разработки ПС с точки зрения достижения определенных целей — эти методы и средства могут использоваться в разных технологических процессах (и в разных технологиях программирования).

Не следует также путать технологию программирования с методологией программирования. В технологии программирования методы рассматриваются «свер-

ху» — с точки зрения организации технологических процессов, а в методологии программирования методы рассматриваются «снизу» — с точки зрения основ их построения методология программирования определяется как совокупность механизмов, применяемых в процессе разработки программного обеспечения и объединенных одним общим подходом).

Имея в виду, что надежность является неотъемлемым атрибутом ПС, мы будем рассматривать технологию программирования как технологию разработки *надежных* ПС. Это означает:

- рассматриваются все процессы разработки ПС, начиная с момента возникновения замысла ПС;
- интересны не только вопросы построения программных конструкций, но и вопросы описания функций и принимаемых решений с точки зрения их человеческого (неформального) восприятия;
- в качестве продукта технологии принимается надежная ПС.

Такой взгляд на технологию программирования будет существенно влиять на организацию технологических процессов, на выбор в них методов и инструментальных средств [26–31].

7.1 Технология программирования и информатизация общества

Технология программирования играла разную роль на разных этапах развития программирования. По мере повышения мощности компьютеров и развития средств и методологии программирования росла и сложность решаемых на компьютерах задач, что привело к повышенному вниманию к технологии программирования. Резкое удешевление стоимости компьютеров и, в особенности, стоимости хранения информации на компьютерных носителях привело к широкому внедрению компьютеров практически во все сферы человеческой деятельности, что существенно изменило направленность технологии программирования. Человеческий фактор стал играть в ней решающую роль. Сформировалось достаточно глубокое понятие качества ПС, причем предпочтение стало отдаваться не столько эффективности ПС, сколько удобству работы с ним для пользователей (не говоря уже о его надежности). Широкое использование компьютерных сетей привело к интенсивному развитию распределенных вычислений, дистанционного доступа к информации и электронного способа обмена сообщениями между людьми. Компьютерная техника из средства решения отдельных задач все более превращается в средство информационного моделирования реального и мыслимого мира, способное просто отвечать людям на интересующие их вопросы. Начинается этап глубокой и полной информатизации (компьютеризации) человеческого общества. Все это ставит перед технологией программирования новые и достаточно трудные проблемы.

Сделаем краткую характеристику развития программирования по десятилетиям. В 50-е годы мощность компьютеров (первого поколения) была невелика, а программирование для них велось, в основном, в машинном коде. Решались,

главным образом, научно-технические задачи (счет по формулам), задание на программирование содержало, как правило, достаточно точную постановку задачи. Использовалась интуитивная технология программирования: почти сразу приступали к составлению программы по заданию, при этом часто задание несколько раз изменялось (что сильно увеличивало время и без того итерационного процесса составления программы), минимальная документация оформлялась уже после того, как программа начинала работать. Тем не менее именно в этот период родилась фундаментальная для технологии программирования концепция модульного программирования, ориентированная на преодоление трудностей программирования в машинном коде. Появились первые языки программирования высокого уровня, из которых только ФОРТРАН пробился для использования в следующие десятилетия.

В 60-е годы можно было наблюдать бурное развитие и широкое использование языков программирования высокого уровня (АЛГОЛ 60, ФОРТРАН, КОВОЛ и др.), значение которых в технологии программирования явно преувеличивалось. Надежда на то, что эти языки решат все проблемы, возникающие в процессе разработки больших программ, не оправдалась. В результате повышения мощности компьютеров и накопления опыта программирования на языках высокого уровня быстро росла сложность решаемых на компьютерах задач, в результате чего обнаружилась ограниченность языков, проигнорировавших модульную организацию программ. И только ФОРТРАН, бережно сохранивший возможность модульного программирования, гордо процветал в следующие десятилетия (все его ругали, но его пользователи отказаться от его услуг не могли из-за грандиозного накопления фонда программных модулей, которые с успехом использовались в новых программах). Кроме того, было понято, что важно не только то, на каком языке мы программируем, но и то, как мы программируем. Это было уже началом серьезных размышлений над методологией и технологией программирования. Появление в компьютерах 2-го поколения прерываний привело к развитию мультипрограммирования и созданию больших программных систем. Широко стала использоваться коллективная разработка, которая поставила ряд серьезных технологических проблем.

В 70-е годы получили широкое распространение информационные системы и базы данных. К середине 70-х годов стоимость хранения одного бита информации на компьютерных носителях стала меньше, чем на традиционных носителях. Это резко повысило интерес к компьютерным системам хранения данных. Началось интенсивное развитие технологии программирования, прежде всего, в следующих направлениях:

- обоснование и широкое внедрение нисходящей разработки и структурного программирования,
- развитие абстрактных типов данных и модульного программирования (в частности, возникновение идеи разделения спецификации и реализации модулей и использование модулей, скрывающих структуры данных),
- исследование проблем обеспечения надежности и мобильности ПС,
- создание методики управления коллективной разработкой ПС,
- появление инструментальных программных средств (программных инструментов) поддержки технологии программирования.

80-е годы характеризуются широким внедрением персональных компьютеров во все сферы человеческой деятельности и тем самым созданием обширного и разнообразного контингента пользователей ПС. Это привело к бурному развитию пользовательских интерфейсов и созданию четкой концепции качества ПС. Появляются языки программирования (например, Ада), учитывающие требования технологии программирования. Развиваются методы и языки спецификации ПС. Начинается бурный процесс стандартизации технологических процессов и, прежде всего, документации, создаваемой в этих процессах. Выходит на передовые позиции объектный подход к разработке ПС. Создаются различные инструментальные среды разработки и сопровождения ПС. Развивается концепция компьютерных сетей.

90-е годы знаменательны широким охватом всего человеческого общества международной компьютерной сетью, персональные компьютеры стали подключаться к ней как терминалы. Это поставило ряд проблем (как технологического, так и юридического и этического характера) регулирования доступа к информации компьютерных сетей. Остро встала проблема защиты компьютерной информации и передаваемых по сети сообщений. Стали бурно развиваться компьютерная технология (CASE-технология) разработки ПС и связанные с ней формальные методы спецификации программ. Начался решающий этап полной информатизации и компьютеризации общества.

7.2 Общие принципы разработки программных средств

Разработка программных средств имеет ряд специфических особенностей. Прежде всего, следует отметить некоторое противостояние: *неформальный* характер требований к ПС (постановки задачи) и понятия ошибки в нем, но *формализованный* основной объект разработки — программы ПС. Тем самым разработка ПС содержит определенные этапы формализации, а переход от неформального к формальному существенно неформален.

Разработка ПС носит *творческий характер* (на каждом шаге приходится делать какой-либо выбор, принимать какое-либо решение), а не сводится к выполнению какой-либо последовательности регламентированных действий. Тем самым эта разработка ближе к процессу проектирования каких-либо сложных устройств, но никак не к их массовому производству. Этот творческий характер разработки ПС сохраняется до самого ее конца.

Следует отметить также особенность продукта разработки. Он представляет собой некоторую совокупность текстов (т.е. *статических* объектов), смысл же (семантика) этих текстов выражается процессами обработки данных и действиями пользователей, запускающих эти процессы (т.е. является *динамическим*). Это предопределяет выбор разработчиком ряда специфичных приемов, методов и средств.

Продукт разработки имеет и другую специфическую особенность: ПС при своем использовании (эксплуатации) не расходуется и не расходует используемых ресурсов.

7.3 Жизненный цикл программного средств



.....

*Под **жизненным циклом** ПС понимают весь период его разработки и эксплуатации (использования), начиная от момента возникновения замысла ПС и кончая прекращением всех видов его использования.*

.....

Жизненный цикл охватывает довольно сложный процесс создания и использования ПС. Этот процесс может быть организован по-разному для разных классов ПС и в зависимости от особенностей коллектива разработчиков. В настоящее время можно выделить 5 основных подходов к организации процесса создания и использования ПС.

Водопадный подход. При таком подходе разработка ПС состоит из цепочки этапов. На каждом этапе создаются документы, используемые на последующем этапе. В исходном документе фиксируются требования к ПС. В конце этой цепочки создаются программы, включаемые в ПС.

Исследовательское программирование. Этот подход предполагает быструю (насколько это возможно) реализацию рабочих версий программ ПС, выполняющих лишь в первом приближении требуемые функции. После экспериментального применения реализованных программ производится их модификация с целью сделать их более полезными для пользователей. Этот процесс повторяется до тех пор, пока ПС не будет достаточно приемлемо для пользователей. Такой подход применялся на ранних этапах развития программирования, когда технологии программирования не придавали большого значения (использовалась интуитивная технология). В настоящее время этот подход применяется для разработки таких ПС, для которых пользователи не могут точно сформулировать требования (например, для разработки систем искусственного интеллекта).

Прототипирование. Этот подход моделирует начальную фазу исследовательского программирования вплоть до создания рабочих версий программ, предназначенных для проведения экспериментов с целью установить требования к ПС. В дальнейшем должна последовать разработка ПС по установленным требованиям в рамках какого-либо другого подхода (например, водопадного).

Формальные преобразования. Этот подход включает разработку формальных спецификаций ПС и превращение их в программы путем корректных преобразований. На этом подходе базируется компьютерная технология (CASE-технология) разработки ПС.

Сборочное программирование. Этот подход предполагает, что ПС конструируется, главным образом, из компонент, которые уже существуют. Должно быть некоторое хранилище (библиотека) таких компонент, каждая из которых может многократно использоваться в разных ПС. Такие компоненты называются *повторно используемыми*. Процесс разработки ПС при данном подходе состоит скорее из сборки программ из компонент, чем из их программирования.

Обычно, на практике, рассматривается водопадный подход с некоторыми модификациями. Во-первых, потому, что в этом подходе приходится иметь дело с боль-

шинством процессов программной инженерии, а во-вторых, потому, что в рамках этого подхода создается большинство больших программных систем. Именно этот подход рассматривается в качестве индустриального подхода разработки программного обеспечения.

Исследовательское программирование исходит из взгляда на программирование как на искусство. Оно применяется тогда, когда водопадный подход не применим из-за того, что не удастся точно сформулировать требования к ПС. Прототипирование рассматривается как вспомогательный подход, используемый в рамках других подходов, в основном для прояснения требований к ПС. Компьютерной технологии (включая обсуждение жизненного цикла ПС, созданного по этой технологии) будет посвящена отдельная лекция. Сборочное программирование мы в нашем курсе рассматривать не будем, хотя о повторно используемых программных модулях мы говорить будем, обсуждая свойства программных модулей.

В рамках водопадного подхода различают следующие стадии жизненного цикла ПС (см. рис. 7.1): разработку ПС, производство программных изделий (ПИ) и эксплуатацию ПС.

Стадия *разработки* ПС состоит из этапа его внешнего описания, этапа конструирования ПС, этапа кодирования (программирование в узком смысле) ПС и этапа аттестации ПС. Всем этим этапам сопутствуют процессы документирования и управления ПС. Этапы конструирования и кодирования часто перекрываются, иногда довольно сильно. Это означает, что кодирование некоторых частей программного средства может быть начато до завершения этапа конструирования.

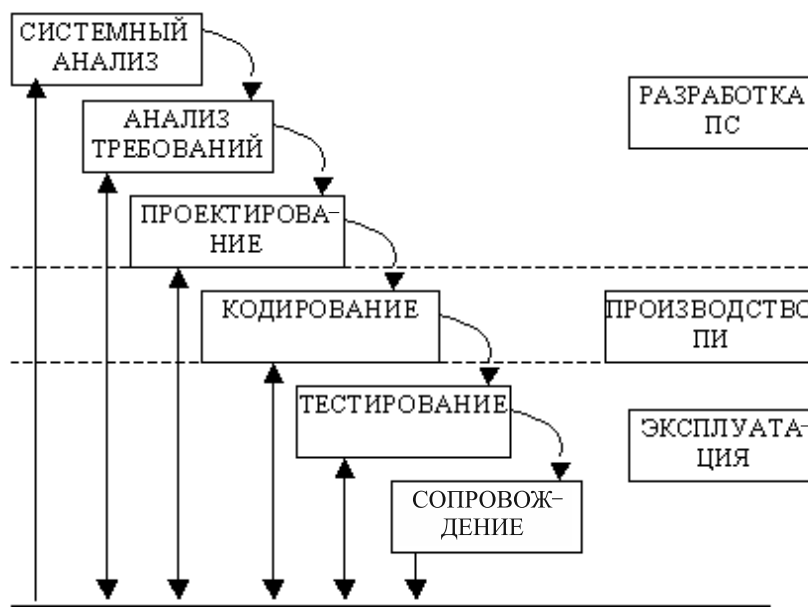


Рис. 7.1 – Стадии и фазы жизненного цикла ПС

Этап *внешнего описания* ПС включает процессы, приводящие к созданию некоторого документа, который мы будем называть *внешним описанием ПС*. Этот документ является описанием поведения ПС с точки зрения внешнего по отношению к нему наблюдателя с фиксацией требований относительно его качества. Внешнее описание ПС начинается с анализа и определения требований к ПС со стороны

пользователей (заказчика), а также включает процессы спецификации этих требований. *Конструирование* ПС охватывает процессы: разработку архитектуры ПС, разработку структур программ ПС и их детальную спецификацию.

Кодирование ПС включает процессы создания текстов программ на языках программирования, их отладку с частичным тестированием ПС.

На этапе *аттестации* ПС производится оценка качества ПС. Если эта оценка оказывается приемлемой для практического использования ПС, то разработка ПС считается законченной. Это обычно оформляется в виде некоторого документа, фиксирующего решение комиссии, проводящей аттестацию ПС.

Программное изделие (ПИ) — экземпляр или копия разработанного ПС. *Изготовление* ПИ — это процесс генерации и/или воспроизведения (снятия копии) программ и программных документов ПС с целью их поставки пользователю для применения по назначению.

Производство ПИ — это совокупность работ по обеспечению изготовления требуемого количества ПИ в установленные сроки. Стадия производства ПИ в жизненном цикле ПС является, по существу, вырожденной (не существенной), так как представляет рутинную работу, которая может быть выполнена автоматически и без ошибок. Этим она принципиально отличается от стадии производства различной техники. В связи с этим в литературе эту стадию, как правило, не включают в жизненный цикл ПС.

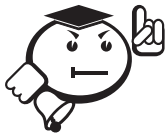
Стадия *эксплуатации* ПС охватывает процессы хранения, внедрения и сопровождения ПС, а также транспортировки и применения ПИ по своему назначению. Она состоит из двух параллельно проходящих фаз: фазы применения ПС и фазы сопровождения ПС.

Применение ПС — это использование ПС для решения практических задач на компьютере путем выполнения ее программ.

Сопровождение ПС — это процесс сбора информации о качестве ПС в эксплуатации, устранения обнаруженных в нем ошибок, его доработки и модификации, а также извещения пользователей о внесенных в него изменениях.

7.4 Понятие качества программного средства

Каждое ПС должно выполнять определенные функции, т. е. делать то, что задумано. Хорошее ПС должно обладать еще целым рядом свойств, позволяющим успешно его использовать в течение длительного периода, т. е. обладать определенным качеством. *Качество* ПС — это совокупность его черт и характеристик, которые влияют на его способность удовлетворять заданные потребности пользователей. Это не означает, что разные ПС должны обладать одной и той же совокупностью таких свойств в их наивысшей степени. Этому препятствует тот факт, что повышение качества ПС по одному из таких свойств часто может быть достигнуто лишь ценой изменения стоимости, сроков завершения разработки и снижения качества этого ПС по другим его свойствам. Качество ПС является удовлетворительным, когда оно обладает указанными свойствами в такой степени, чтобы гарантировать успешное его использование.



.....
 Совокупность свойств ПС, которая образует удовлетворительное для пользователя качество ПС, зависит от условий и характера эксплуатации этого ПС, т.е. от позиции, с которой должно рассматриваться качество этого ПС. Поэтому при описании качества ПС, прежде всего, должны быть фиксированы *критерии* отбора требуемых свойств ПС.

В настоящее время *критериями качества ПС* принято считать:

- функциональность,
- надежность,
- легкость применения,
- эффективность,
- сопровождаемость,
- мобильность.

Функциональность — это способность ПС выполнять набор функций, удовлетворяющих заданным или подразумеваемым потребностям пользователей. Набор указанных функций определяется во внешнем описании ПС.

Надежность подробно обсуждалась в первой лекции.

Легкость применения — это характеристики ПС, которые позволяют минимизировать усилия пользователя по подготовке исходных данных, применению ПС и оценке полученных результатов, а также вызывать положительные эмоции определенного или подразумеваемого пользователя.

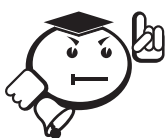
Эффективность — это отношение уровня услуг, предоставляемых ПС пользователю при заданных условиях, к объему используемых ресурсов.

Сопровождаемость — это характеристики ПС, которые позволяют минимизировать усилия по внесению изменений для устранения в нем ошибок и по его модификации в соответствии с изменяющимися потребностями пользователей.

Мобильность — это способность ПС быть перенесенным из одной среды (окружения) в другую, в частности с одного компьютера на другой.

Функциональность и надежность являются обязательными критериями качества ПС, причем обеспечение надежности будет красной нитью проходить по всем этапам и процессам разработки ПС. Остальные критерии используются в зависимости от потребностей пользователей в соответствии с требованиями к ПС.

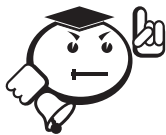
7.5 Надежность программных средств



.....
 Рассмотрим теперь общие принципы обеспечения надежности ПС, что, как мы уже подчеркивали, является основным мотивом разработки ПС, задающим специфическую окраску всем технологическим процессам разработки ПС.

В технике известны четыре подхода к обеспечению надежности:

- предупреждение ошибок;
- самообнаружение ошибок;
- самоисправление ошибок;
- обеспечение устойчивости к ошибкам.



.....
 Целью подхода предупреждения ошибок — не допустить ошибок
 в готовых продуктах, в нашем случае — в ПС.

Проведенное рассмотрение природы ошибок при разработке ПС позволяет для достижения этой цели сконцентрировать внимание на следующих вопросах:

- борьба со сложностью,
- обеспечение точности перевода,
- преодоление барьера между пользователем и разработчиком,
- обеспечение контроля принимаемых решений.

Этот подход связан с организацией процессов разработки ПС, т. е. с технологией программирования. И хотя, как мы уже отмечали, гарантировать отсутствие ошибок в ПС невозможно, но в рамках этого подхода можно достигнуть приемлемого уровня надежности ПС.

Остальные три подхода связаны с организацией самих продуктов технологии, в нашем случае — программ. Они учитывают возможность ошибки в программах. Самообнаружение ошибки в программе означает, что программа содержит средства обнаружения отказа в процессе ее выполнения. Самоисправление ошибки в программе означает не только обнаружение отказа в процессе ее выполнения, но и исправление последствий этого отказа, для чего в программе должны иметься соответствующие средства. Обеспечение устойчивости программы к ошибкам означает, что в программе содержатся средства, позволяющие локализовать область влияния отказа программы либо уменьшить его неприятные последствия, а иногда предотвратить катастрофические последствия отказа. Однако эти подходы используются весьма редко (может быть, относительно чаще используется обеспечение устойчивости к ошибкам). Связано это, во-первых, с тем, что многие простые методы, используемые в технике в рамках этих подходов, неприменимы в программировании, например дублирование отдельных блоков и устройств (выполнение двух копий одной и той же программы всегда будет приводить к одинаковому эффекту — правильному или неправильному). Во-вторых, добавление в программу дополнительных фрагментов приводит к ее усложнению (иногда — значительному), что в какой-то мере мешает методам предупреждения ошибок.

7.6 Методы борьбы со сложностью программных средств

При разработке ПС известны два общих метода борьбы со сложностью систем:

- обеспечение независимости компонент системы;
- использование в системах иерархических структур.

Обеспечение независимости компонент означает разбиение системы на такие части, между которыми должны остаться по возможности меньше связей. Одним из воплощений этого метода является модульное программирование. Использование в системах иерархических структур позволяет локализовать связи между компонентами, допуская их лишь между компонентами, принадлежащими смежным уровням иерархии. Этот метод, по существу, означает разбиение большой системы на подсистемы, образующие малую систему. Здесь существенно используется способность человека к абстрагированию.

7.7 Контроль принимаемых решений

Обязательным шагом в каждом процессе (этапе) разработки ПС должна быть проверка правильности принятых решений. Это позволит обнаруживать и исправлять ошибки на самой ранней стадии после ее возникновения, что, во-первых, существенно снижает стоимость ее исправления и, во-вторых, повышает вероятность правильного ее устранения. С учетом специфики разработки ПС необходимо применять везде, где это возможно:

- смежный контроль,
- сочетание как статических, так и динамических методов контроля.

Смежный контроль означает, проверку полученного документа лицами, не участвующими в его разработке, с двух сторон: во-первых, со стороны автора исходного для контролируемого процесса документа и, во-вторых, лицами, которые будут использовать полученный документ в качестве исходного в последующих технологических процессах. Такой контроль позволяет обеспечивать однозначность интерпретации полученного документа.

Сочетание статических и динамических методов контроля означает, что нужно не только контролировать документ как таковой, но и проверять, какой процесс обработки данных он описывает. Это отражает одну из специфических особенностей ПС (статическая форма, динамическое содержание).



Контрольные вопросы по главе 7

- 1) Что понимается под технологией программирования?
- 2) Что такое жизненный цикл программного средства?
- 3) Сколько выделяется основных подходов к организации процесса создания и использования ПС?
- 4) Что такое сборочное программирование?
- 5) Назовите стадии и фазы жизненного цикла ПС.
- 6) Что такое этап аттестации ПС?
- 7) Что такое программное изделие?
- 8) Какие критерии качества ПС существуют?
- 9) Какие вопросы рассматриваются при предупреждении ошибок?
- 10) Какие при разработке ПС известны методы борьбы со сложностью систем?

Глава 8

ОБЪЕКТНЫЙ ПОДХОД К РАЗРАБОТКЕ ПРОГРАММНЫХ СРЕДСТВ

Окружающий нас мир состоит из объектов и отношений между ними. Согласно данным словаря В. Даля объект (предмет) — это все, что представляется чувствам (объект вещественный) или уму (объект умственный). Таким образом, *объект* воплощает некоторую сущность и имеет некоторое состояние, которое может изменяться со временем как следствие влияния других объектов, находящихся с первым в каких-либо отношениях. Он может иметь внутреннюю структуру: состоять из других объектов, также находящихся между собой в некоторых отношениях. Исходя из этого можно построить иерархическое строение мира из объектов. Однако при каждом конкретном рассмотрении окружающего нас мира некоторые объекты считаются неделимыми, причем, в зависимости от целей рассмотрения, такими (неделимыми) могут приниматься объекты разного уровня иерархии.



.....
Отношение связывает некоторые объекты: можно считать, что объединение этих объектов обладает некоторым свойством.
.....

Если отношение связывает n объектов, то такое отношение называется n -местным (парным). На каждом месте объединения объектов, которые могут быть связаны каким-либо конкретным отношением, могут находиться разные объекты, но вполне определенные (в этом случае говорят: объекты определенного класса). Одноместное отношение называется *простым свойством объекта* (соответствующего класса). Многоместное отношение объектов будем называть *ассоциативным свойством объекта*, если этот объект участвует в этом отношении.



.....
Состояние объекта может быть изучено по значению простых или ассоциативных свойств этого объекта. Множество всех объектов, которые обладают каким-то общим набором свойств, называется *классом объектов*.

В процессе познания или изменения окружающего нас мира мы всегда принимаем в рассмотрение ту или иную упрощенную модель мира (*модельный мир*), в которую включаем объекты и отношения некоторых интересующих нас классов из окружающего нас мира. Каждый объект, имеющий внутреннюю структуру, может представлять свой модельный мир, включающий объекты этой структуры и отношения, которые их связывают. Таким образом, окружающий нас мир можно рассматривать (в некотором приближении) как иерархическую структуру модельных миров. В настоящее время в процессе познания или изменения окружающего нас мира широко используется компьютерная техника для обработки различного рода информации. В связи с этим применяется компьютерное (информационное) представление объектов и отношений. Каждый объект информационно может быть представлен некоторой структурой данных, отображающей его состояние.



.....
Простые свойства этого объекта могут задаваться непосредственно в виде отдельных компонент этой структуры либо специальными функциями над этой структурой данных.



.....
Ассоциативные свойства (n -местные отношения для $n > 1$) можно представить либо в активной форме, либо в пассивной форме.

В активной форме n -местное отношение представляется некоторым программным фрагментом, реализующим либо n -местную функцию (определяющую значение свойства соответствующего объединения объектов), либо процедуру, осуществляющую по состоянию представлений объектов, связываемых представляемым отношением, изменение состояний некоторых из них. В пассивной форме такое отношение может быть представлено некоторой структурой данных (в которую могут входить и представления объектов, связываемых этим отношением), интерпретируемой на основании принятых соглашений по общим процедурам, независимым от конкретных отношений (например, реляционная база данных). В любом случае представление отношения определяет некоторые действия по обработке данных.

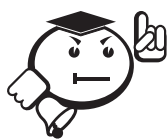
При исследовании модельного мира пользователи могут по-разному получать информацию от компьютера. В одних случаях пользователей может интересовать получение информации об отдельных свойствах определенных объектов или результаты какого-либо взаимодействия между некоторыми объектами модельного мира. Для удовлетворения таких запросов разрабатываются соответствующие

ПС, которые выполняют интересующие пользователей функции, или подходящие информационные системы, способные выдавать информацию об интересующих пользователей отношениях.

В начальный период развития компьютерной техники (при недостаточно высокой мощности компьютеров) такой подход к исследованию модельного мира был вполне естественным. Именно он и провоцировал *реляционный (процедурный, функциональный* и т. п.) подход к разработке ПС, который был подробно рассмотрен в предшествующих разделах пособия. Сущность этого подхода состоит в систематическом использовании *декомпозиции функций (отношений)* для описания и построения структуры ПС (включая тексты программ). При этом сами объекты модельного мира, с которыми связаны заказываемые и реализуемые функции, представлялись фрагментарно (в том объеме, который необходим для выполнения этих функций) и в форме, удобной для реализации этих функций.

Тем самым обеспечивалась эффективная реализация требуемых функций, но не создавалось цельного и адекватного компьютерного представления модельного мира, интересующего пользователя. Попытки даже незначительного расширения объема и характера информации об этом модельном мире, которую можно получить от ПС, могли потребовать серьезной модернизации этого ПС.

В других случаях пользователя может интересовать наблюдение за изменением состояний объектов модельного мира в результате их взаимодействий. Это требует использования подходящих информационных моделей таких объектов, создания программных средств, моделирующих процессы взаимодействия объектов модельного мира, и предоставления пользователю доступа к этим информационным моделям (к *пользовательским* объектам). С помощью традиционных методов разработки это оказалось довольно трудоемкой задачей. Наиболее полно отвечает решению этой задачи *объектный* подход к разработке ПС. Сущность его состоит в систематическом использовании *декомпозиции объектов* при описании и построении ПС. При этом функции (отношения), выполняемые таким ПС, будут выражаться через отношения объектов других уровней, т. е. их декомпозиция будет существенно зависеть от декомпозиции объектов.



С точки зрения разработчиков ПС следует различать следующие категории объектов (и соответственно их классов):

- объекты модельного (вещественного или умственного) мира,
- информационные модели объектов реального мира (будем называть их *пользовательскими объектами*),
- объекты процесса выполнения программ,
- объекты процесса разработки ПС (*технологические объекты программирования*).

Кроме того, в зависимости от способа представления в компьютере модельного мира и характера взаимодействия с ним со стороны пользователя, следует различать пассивные и активные объекты. *Пассивный объект* представляет собой

некоторый фрагмент информационной среды, который способен хранить разные данные определенного типа (представляющие разные *состояния* этого объекта) и с которым связан некоторый набор операций (*применимых* к этому объекту). Операции над таким объектом применяются под воздействием некоторой *внешней* по отношению к этому объекту *активной силы*, исходящей либо от пользователя, либо от какого-либо программного фрагмента в процессе его выполнения. *Активный объект* представляет собой такое расширение пассивного объекта, в котором фрагмент информационной среды способен также хранить и программные фрагменты, способные находиться в процессе выполнения (в *активном состоянии*). Активный объект, у которого какие-либо программные фрагменты находятся в активном состоянии, способен воспринимать сообщения или сигналы из операционной среды, в которую он погружен, и самостоятельно выполнять некоторые операции как реакцию на эти сообщения или сигналы. Таким образом, можно считать, что активный объект обладает *внутренней активной силой*.

Когда говорят об объектно-ориентированном подходе к разработке ПС, имеют в виду объектный подход с ориентацией на описание объектов модельного мира и построением их информационных моделей, причем используются, в основном, активные объекты. При этом многие процессы разработки ПС приобретают специфические («объектные») черты:

- использование системы понятий, позволяющих описывать объекты и их классы,
- декомпозиция объектов, являющаяся основным средством упрощения ПС,
- использование внепрограммных абстракций для упрощения процессов разработки,
- предпочтение (приоритет) разработки структуры данных перед реализацией функций.

Основные из этих специфических особенностей разработки ПС покажем в рамках водопадной модели технологии [32–56].

8.1 Особенности объектного подхода к разработке программного средства

При объектном подходе этап внешнего описания ПС оказывается существенно более емким и содержательным по сравнению с реляционным подходом. Определение требований заключается в неформальном описании модельного мира, который пользователь собирается изучать или просто использовать с помощью требуемого ПС. При этом повышается роль прототипирования, которое при этом подходе часто окупается уменьшением объема работы на последующих этапах разработки ПС.

Часто определение требования завершается разработкой *диаграммы вариантов использования*, в которой фиксируются, в каких случаях будет использоваться ПС. Это позволяет при разработке ПС ограничиться только такими ее функциональными возможностями, которые поддерживают эти случаи (варианты) использования. По существу, диаграмма вариантов использования представляет со-

бой описание некоторого контекста использования. На рис. 8.1 приведен пример диаграммы вариантов использования.

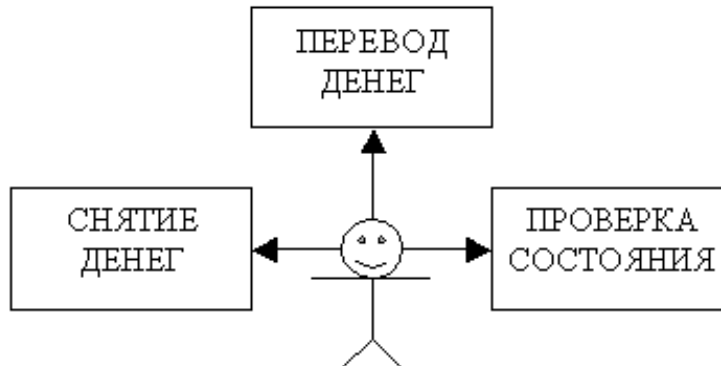
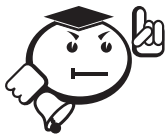


Рис. 8.1 – Диаграмма вариантов использования для банкомата



.....
 Существенно изменяется содержание процесса спецификации требований: вместо разработки функциональной спецификации ПС создается формальное описание модельного мира, состоящее из трех частей:

- 1) объектной модели,
 - 2) динамической модели,
 - 3) функциональной модели.
-

Назначение этих частей можно образно определить следующим образом: объектная модель определяет то, с чем что-то случается; динамическая модель определяет, когда это случается; функциональная модель определяет то, что случается.

Объектная модель показывает статическую объектную структуру модельного мира, который должно представлять разрабатываемое ПС (программная система). Она включает определения используемых классов объектов и отношений между этими классами, а также определение используемых объектов этих классов и отношения между этими объектами. Обычно *класс объектов* в объектной модели представляется в виде тройки

(Имя класса,

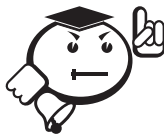
Список атрибутов,

Список операций).

Каждый атрибут представляется некоторым именем и может принимать значения определенного типа. По существу, атрибут класса выражает некоторое простое свойство объектов этого класса. Представление некоторых простых свойств объектов атрибутами весьма удобно, особенно когда по значениям этих атрибутов осуществляется классификация объектов. Операции, указываемые в представлении класса, отражают другие свойства объектов этого класса (как простые, так

и ассоциативные). Они показывают, что можно делать с объектами этого класса (или что могут делать сами эти объекты).

В объектной модели *отношения между объектами* некоторых классов обобщаются в *отношения между этими классами*. При этом используются, как правило, только одноместные и двуместные отношения между объектами. Более сложные отношения приводят к неоправданному усложнению объектных моделей, а с другой стороны, такие отношения всегда могут быть сведены к двуместным за счет определения дополнительных классов. Одноместные отношения называют *атрибутами*, причем некоторые атрибуты объекта получаются из атрибутов класса присвоением им конкретных значений. Отношение между двумя (и более) объектами называют *связями*, а их обобщение (отношение между классами) обычно называют *ассоциациями*.



.....
Ассоциации определяют допустимые связи между объектами. Различают следующие виды ассоциаций:

- взаимодействия состояний объектов,
 - агрегирования (структурирования) объектов,
 - абстрагирования (порождения) классов.
-

Ассоциация «взаимодействие», по существу, означает, что объекты классов, находящихся в таком отношении, могут быть параметрами некоторых операций. Ассоциация «агрегирование» означает, что объект одного из классов, находящихся в таком отношении, включает (или может включать) в себя (как часть) объекты другого из этих классов. Ассоциация «абстрагирование» означает, что один из классов, находящихся в таком отношении, наследует свойства другого из этих классов и может обладать также и другими (дополнительными) свойствами.

Для представления объектной модели часто используются графические языки спецификации объектов (например, язык UML). На таких языках классы и объекты задаются прямоугольниками, в которых указывается специфицирующая их информация. Для задания отношений между двумя классами соответствующие им прямоугольники связываются линией, снабженной различными графическими значками и некоторыми надписями. Графические значки специфицируют характер (вид) отношения между этими классами, а надписи обеспечивают полную идентификацию этого отношения (делают его конкретным).

Например, на рис. 8.2 заданное отношение между классами Страна и Город имеет характер «один к одному». Более конкретно это отношение означает, что каждый объект класса Страна обязательно связан отношением «имеет столицей» с одним и только одним объектом класса Город и этот объект класса Город не связан таким отношением ни с каким другим объектом класса Страна.

Для описания декомпозиции объектов используется отношение вида агрегирования. Например, отношение «программа состоит из одного или нескольких модулей» представлено на рис. 8.3.

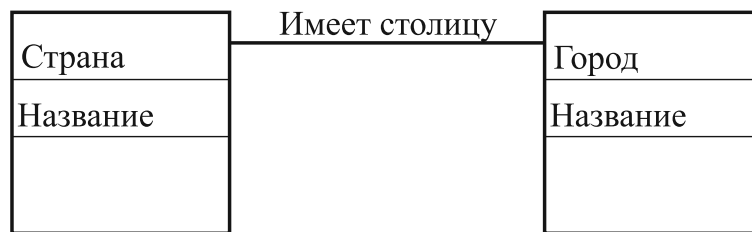


Рис. 8.2 – Пример отношения между классами объектов

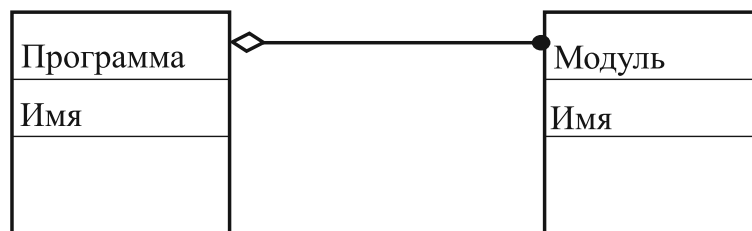


Рис. 8.3 – Пример отношения агрегирования между классами объектов

Следует заметить, что объектная модель полностью включает описание внешней информационной среды при реляционном подходе (из предмета Базы данных).



.....
Динамическая модель показывает допустимые последовательности изменений состояний объектов из объектной модели модельного мира, который должно представлять разрабатываемое ПС (программная система).

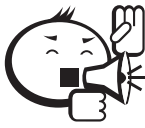
Она описывает последовательности операций в ответ на внешние сигналы (взаимодействия) без рассмотрения того, что эти операции делают. Динамическая модель необходима, если в соответствующей объектной модели имеются активные объекты.



.....
 Основные понятия динамической модели: события и состояния объектов.

Под *событием* здесь понимается элементарное воздействие одного объекта на другой, происходящее в определенный момент времени. Одно событие может логически предшествовать другому или быть не связанным с другим. Другими словами, события в динамической модели *частично упорядочены*.

Под *состоянием объекта* здесь понимается совокупность значений атрибутов объекта и представления текущих связей этого объекта с другими объектами. Состояние объекта связывается с интервалом времени между некоторыми двумя событиями, на которые реагирует этот объект. Объект *переходит* из одного состояния в другое в результате реакции на некоторое событие (в конце интервала, связанного с этим состоянием).



.....
 В связи с этим в динамической модели для каждого класса активных объектов строится своя *диаграмма состояний*.

Она представляет собой граф, вершинами которого являются состояния, а дугами — переходы между этими состояниями (переходы часто обозначаются именами событий). Некоторые переходы могут быть связаны с условиями, разрешающими эти переходы. *Условие* — это предикат, зависящий от значений некоторых атрибутов объекта. Каждое условие указывается на дуге, переходом по которой управляет это условие.

Существенно, что в диаграмме состояний с некоторыми состояниями или событиями связываются определенные операции. Операция, связываемая с событием, обозначает реакцию объекта на это событие, и считается, что она выполняется мгновенно (в точке некоторого временного интервала). Такая операция называется *действием*. Операция, связываемая с состоянием, выполняется в рамках временного интервала, с которым связано это состояние (т. е. имеет продолжительность, ограниченную этим интервалом). Такая операция называется *деятельностью*. Диаграмма состояний определяет управление активизацией указанных операций. Таким образом, диаграмма состояний описывает поведение одного класса объектов.

Динамическая модель в целом объединяет все диаграммы состояний с помощью событий между классами. *Функциональная модель* показывает, как вычисляются выходные значения из входных без указания порядка, в котором эти значения вычисляются. Она определяет все операции, условия и ограничения, используемые в объектной и динамической моделях (*внешние операции*). Функциональная модель соответствует определению *внешних функций* при реляционном подходе к разработке ПС.

Для определения крупных операций в функциональной модели используются *поточковые диаграммы* (*диаграммы потоков данных*), позволяющие выразить эти операции через более простые операции. Основными понятиями поточковых диаграмм являются *процессы*, *объекты* и *потоки данных*. Поточковая диаграмма — это граф, вершинами которого являются объекты или процессы, а дугами — потоки данных. Процессы преобразуют данные, поступающие от одних объектов и направляемые для хранения в другие объекты. Эти процессы представляют *внутренние* операции, через которые выражается операция, представляемая данной поточковой диаграммой. Объекты могут быть пассивными (*хранилищами данных*) и активными (*агентами*). Пассивные объекты используются только для хранения данных, а активные объекты используются как для хранения, так и для преобразования данных. Потоки данных определяют допустимые направления перемещения данных и типы перемещаемых данных.

Таким образом, основным содержанием этапа разработки при объектном подходе является *объектное моделирование*. При этом широко используются формальные языки спецификаций, в том числе и графические. Одним из наиболее употребительных в настоящее время таких языков является язык UML.

8.2 Особенности объектного подхода на этапе конструирования программного средства

На этапе конструирования при объектном подходе продолжается процесс объектного моделирования: уточняются модели, построенные на этапе внешнего описания, в терминах описания программных систем и производится дальнейшая декомпозиция объектов. В процессе разработки объектной архитектуры ПС выделяются все объекты, с информационными моделями которых собирается непосредственно работать пользователь, и завершается их программная спецификация, а также определяется их пользовательский интерфейс. Такие объекты мы будем называть *пользовательскими*. Классы таких объектов или отдельные активные объекты образуют архитектурные подсистемы. Определяется метод взаимодействия между этими подсистемами.

В случае использования активных объектов основным широким классом архитектур при объектном подходе является *коллектив параллельно действующих программ*, причем здесь роль программ выполняют как раз эти активные объекты, а способ управления передачей сообщений зависит от выбранного подкласса таких архитектур. Типичной архитектурой такого класса является архитектура «клиент-сервер» (см. рис. 8.4). В такой системе один из активных объектов, называемый *сервером*, выполняет определенные программные услуги по запросам других активных объектов, называемых *клиентами*. Такой запрос передается серверу с помощью сообщения от клиента, результат выполнения сервером запроса передается соответствующему клиенту с помощью другого сообщения.

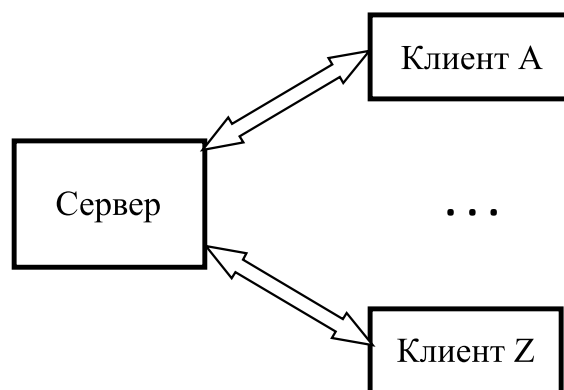


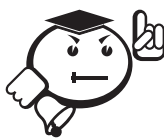
Рис. 8.4 – Архитектура «Клиент-Сервер»

Дальнейшая разработка структуры программных подсистем и их кодирование на языках программирования может осуществляться уже в рамках реляционного подхода на ориентированных на него языках программирования — пользователь внутреннюю организацию этих подсистем уже «не видит». Однако во многих случаях существуют сильные аргументы за то, чтобы продолжить объектную декомпозицию этих подсистем. Объектная структура этих подсистем может быть существенно более понятной *разработчику*, чем их структура при реляционном подходе. Кроме того, продолжение объектной декомпозиции и использование основных понятий и методов объектного подхода при дальнейшей разработке ПС представ-

ляется «естественным», так как весь процесс разработки становится единообразным (концептуально целостным).

При разработке модульной структуры при объектном подходе используются преимущественно информационно прочные модули. Информационно прочный модуль, реализующий какой-либо класс объектов или логически связанную совокупность таких классов, обычно называют *компонентом* ПС.

8.3 Особенности объектного подхода на этапе кодирования программного средства



.....
На этапе кодирования при объектном подходе используются языки программирования уже другого типа — *объектно-ориентированные*.
.....

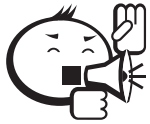
Считается, что язык программирования поддерживает объектно-ориентированное программирование, если он включает конструкции:

- 1) для инкапсуляции и абстракции данных,
- 2) наследования,
- 3) динамического полиморфизма.

Однако самое существенное различие между реляционными и объектно-ориентированными языками программирования заключается в следующем: в качестве основных средств декомпозиции программ в реляционных языках используются описания функций и процедур (декомпозиция отношений), а в объектно-ориентированных языках — описание классов объектов (декомпозиция объектов). Объекты, возникающие в программах при такой декомпозиции архитектурных подсистем, мы будем называть *объектами процесса выполнения программ*. Изменяется и технология разработки программного модуля. При реляционном подходе для разработки модуля (в основном, реализующих функцию или процедуру) рекомендовались структурное программирование и пошаговая детализация. При объектно-ориентированном подходе для разработки компонента (информационно прочного модуля) более подходит предложенная Дейкстрой нисходящая технология разработки *слоистой* структуры модуля.

В этой технологии каждый модуль рассматривается состоящим из некоторой упорядоченной совокупности программных слоев, причем каждый слой является реализацией некоторой абстрактной машины (в нашем случае это можно рассматривать как реализацию класса или логически связанной совокупности классов). Таким образом, модуль образует определенная цепочка иерархически связанных абстрактных машин (так называемые «бусы»). Реализация каждой абстрактной машины выражается в терминах ниже стоящей абстрактной машины (ниже стоящего программного слоя) или в терминах выбранного языка программирования (при реализации абстрактной машины самого нижнего уровня). Разработка модуля начинается с разработки абстрактной машины самого верхнего уровня.

8.4 Качество ПО и методы его обеспечения



.....
Требования к ПО определяют, какие свойства и характеристики оно должно иметь для удовлетворения потребностей пользователей и других заинтересованных лиц.
.....

Для того чтобы ПО действительно было полезным, важно, чтобы отражались их реальные потребности, которые часто отличаются от непосредственно выражаемых пользователями и заказчиками желаний. Для выявления этих потребностей зачастую приходится проводить достаточно значительную работу. При разработке ПО для непосредственной продажи это практически очевидно — чтобы оно действительно продавалось, необходимо сначала выяснить, что же хотят его потенциальные покупатели. Но и при разработке на заказ, когда заказчик приходит с вроде бы готовым заданием — например, ему нужно сделать систему поддержки контроля качества строительства и эксплуатации магистральных газопроводов — полезно выяснить, в чем же именно он нуждается, какие основные задачи должна решать эта система, какие из задач стоят очень остро и обязательно должны быть отражены уже в первой версии, а какие могут и потерпеть.

Определение потребностей базируется на анализе предметной области, выявляющем основные цели и задачи деятельности организаций — будущих пользователей системы. Потребности выявляются на основе наиболее актуальных проблем и задач, которые эти организации видят перед собой. При этом часто требуется аккуратное рассмотрение текущей ситуации, выявление значимых проблем и расстановка приоритетов их решения, поскольку чаще всего решить сразу все проблемы невозможно.

Формулировка потребностей может быть разбита на следующие этапы:

- 1) выделить одну-две-три основных проблемы,
- 2) определить причины возникновения проблем, оценить их степень влияния и выделить наиболее существенные из проблем, влекущие появление остальных,
- 3) выявить всех заинтересованных лиц, которым придется иметь дело с системой,
- 4) определить границы системы — линию раздела между системой, которую надо сделать, и ее окружением, влияние которого надо учитывать,
- 5) определить ограничения на возможные решения.

Формулировка потребностей не должна накладывать лишних ограничений на возможные решения, удовлетворяющие им. Нужно попытаться сформулировать, что именно является проблемой, а не придумывать сразу возможные решения. При выявлении потребностей пользователей используются такие приемы, как анкетирование, демонстрация раскадровок работы будущей системы, интерактивные сеансы, где пользователям предоставляется возможность самим предложить ва-

рианты внешнего вида системы и ее работы или поменять предложенные кем-то другим прототипы.

На основе потребностей пользователей формулируются возможные *функции* будущей системы, представляющие собой услуги, удовлетворяющие потребности одной или нескольких групп пользователей (или других заинтересованных лиц). Идеи для определения функций можно брать из имеющегося опыта (наиболее часто используемый источник) или из результатов мозговых штурмов и других форм выработки.

Формулировка функций должна быть достаточно короткой, ясной для пользователей, без лишних деталей. Например:

- 1) все данные о сделках и клиентах будут сохраняться в базе данных,
- 2) статус выполнения заказа можно будет узнать через Интернет,
- 3) система будет поддерживать до 1000 одновременно работающих пользователей,
- 4) расписание проведения ремонтных работ будет строиться автоматически.

Только имея набор функций, достаточно полно решающий проблемы, выделенные на предыдущем этапе как наиболее существенные, можно составлять требования — детализацию работы этих функций. При этом надо учитывать, что часто ПО является частью программно-аппаратной системы, требования к которой надо преобразовать на требования к программной и аппаратной ее составляющим (в последнее время, в связи со значительным падением цен на мощное аппаратное обеспечение общего назначения, фокус внимания переместился, в основном, на программное обеспечение). Соотношение между проблемами, потребностями, функциями и требованиями изображено на рис. 8.5.

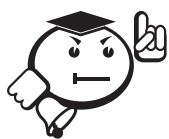


Рис. 8.5 – Соотношение между проблемами, потребностями, функциями и требованиями

Каждое требование раскрывает детали поведения системы при выполнении ею некоторой функции в некоторых обстоятельствах. При этом часть требований происходит из пожеланий заинтересованных лиц и решений, удовлетворяющих эти пожелания, а часть — из внешних ограничений, накладываемых на систему,

например, законами той области знания, в рамках которой системе придется работать, государственным законодательством, корпоративной политикой и пр. Еще до перехода от функций к требованиям полезно расставить приоритеты и оценить трудоемкость их реализации и рискованность. Это позволит отказаться от реализации наименее важных функций еще до их детальной проработки, а также выявить наиболее проблемные места проекта.

Список стандартов, определяющих правила работы с требованиями к ПО (и более общими, системными требованиями), выглядит так.



.....
IEEE 830-1998 Recommended Practice for Software Requirements Specifications описывает структуру документов для фиксации требований к ПО.
.....

Кроме того, он определяет характеристики, которыми должен обладать правильно составленный набор требований:

- корректность (соответствие реальным потребностям),
- недвусмысленность (однозначность понимания),
- полнота (отражение всех основных потребностей),
- непротиворечивость (согласованность между различными элементами),
- упорядоченность по важности и стабильности,
- возможность проверки,
- модифицируемость.

Прослеживаемость в ходе разработки (возможность увязать требование с подсистемами, модулями и операциям, ответственными за его выполнение, и с тестами, проверяющими его выполнение). IEEE 1233-1998, 2002 Guide for Developing System Requirements Specifications. Описывает правила построения требований для (программно-аппаратных) систем в целом. Сейчас широко распространены техники фиксации требований в виде *вариантов использования*.

Вариантом использования (use case) называют некоторый сценарий действий системы, который обеспечивает ощутимый и значимый для ее пользователей результат. На практике в виде одного варианта использования оформляется сценарий действий системы, который будет, скорее всего, неоднократно возникать во время ее использования и имеет достаточно четко определенные условия начала выполнения и завершения. Пример сценария использования. Покупатель в Интернет-магазине выбирает товар. Для этого он может выбрать категорию товара, фирму-изготовителя или группу таких фирм и отфильтровать оставшиеся товары по цене, габаритам и цвету. Определившись, он выбирает товар, кликая на соответствующем значке мышкой (см. рис. 8.6).

В языке UML варианты использования изображают в виде овалов на соответствующих диаграммах. Варианты использования могут быть связаны с участвующими в них действующими лицами (actors), представляющими различные роли пользователей системы или внешние системы, взаимодействующие с ней.

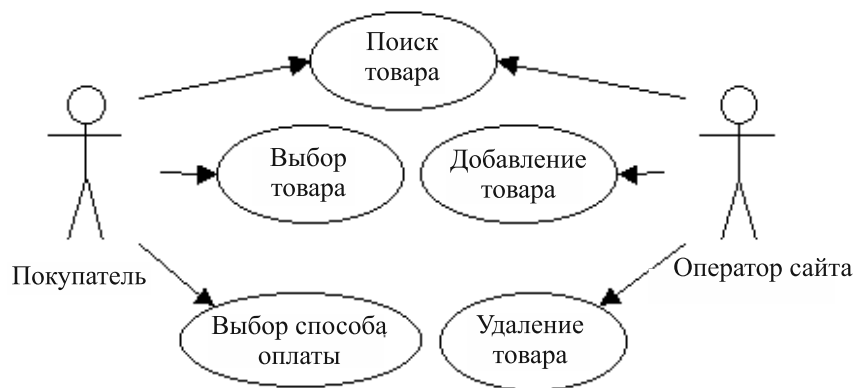


Рис. 8.6 – Диаграмма «сырых» вариантов использования для простого Интернет-магазина

Кроме того, варианты использования могут быть связаны друг с другом двумя видами связей: связями расширения и использования.

Вариант использования В *расширяет* вариант использования А, если В определяет дополнительные действия, которые вставляются в некоторое место в сценарии работы А. Рассмотрим систему регистрации участия студентов в курсах. В ее рамках есть вариант использования «Регистрации участия студента в некотором курсе». Если в институте есть иностранные студенты и их регистрация требует каких-то дополнительных действий, вариант использования «Регистрация участия иностранного студента в курсе» будет расширять указанный ранее.

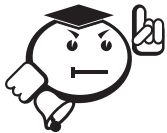
Вариант использования В *использует* вариант использования А, если В включает полностью сценарий А где-то внутри своего сценария работы. В рамках той же системы может существовать вариант использования «Исправление ошибочно введенных данных о студенте». Если в рамках регистрации студента такое исправление становится возможным, то вариант использования «Регистрация участия студента в курсе» использует «Исправление ошибочно введенных данных». Основная разница между расширением и использованием состоит в целях двух находящихся в таком отношении вариантов использования. Если их цели по большому счету одинаковы, пользователь ожидает от них примерно одинаковых результатов, то один из них расширяет другой, если цели существенно различны — то использует.

Хорошо описанный вариант использования должен иметь следующие данные:

- *Имя*, ясно говорящее о назначении соответствующего сценария.
- *Описание*. Несколько предложений, описывающих этот вариант использования.
- *Частота*. Насколько часто соответствующий сценарий возникает.
- *Предусловия*. Все условия запуска такого сценария.
- *Постусловия*. Все условия, которые выполнены по успешном выполнении сценария.

Основной сценарий работы, который используется в большинстве случаев:

- Альтернативные сценарии, возникающие иногда. Для каждого альтернативного сценария указываются условия его срабатывания.
- (Необязательно) Задействованные роли (actors).
- (Необязательно) Расширяемые варианты.
- (Необязательно) Используемые варианты.
- (Необязательно) Статус — в разработке, готов к проверке, в процессе проверки, подтвержден, отвергнут.
- (Необязательно) Допущения об окружении и ходе работ системы, использованные при разработке сценария. В полностью готовом сценарии эти допущения должны быть подтверждены и стать ограничениями системы либо давать начало различным подсценариям работы.
- Качество ПО.



.....
Как проверить что требования заданы достаточно полно? Для этого служит понятие *качества ПО*.
.....

Если попросить группу людей высказать своё мнение по поводу того, что такое качественное ПО, можно получить следующие варианты ответов:

- Его легко использовать.
- Оно демонстрирует хорошую производительность.
- В нем нет ошибок.
- Оно не портит пользовательские данные при сбоях.
- Его можно использовать на разных платформах.
- Оно может работать 24 часа в сутки и 7 дней в неделю.
- В него легко добавлять новые возможности.
- Оно удовлетворяет потребности пользователей.
- Оно надежно.
- Оно хорошо документировано.

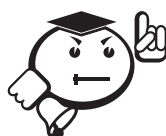
Все это действительно имеет непосредственное отношение к качеству ПО. Но все эти ответы выделяют характеристики, важные для конкретного пользователя, разработчика или группы таких лиц. Для повышения степени удовлетворения всех заинтересованных сторон, для достижения прочного положения на рынке и повышения потенциала развития важен учет всей совокупности характеристик ПО, важных для всех заинтересованных лиц: конечных пользователей, заказчиков, разработчиков, администраторов систем, в которых оно будет работать, регулирующих организаций и пр. Приведенные ответы показывают, что качество ПО может быть описано только большим количеством разнородных характеристик. Общие принципы разработки качественных продуктов во всех отраслях экономики ре-

гулируются набором стандартов ISO 9000. Наиболее важные для разработки ПО стандарты в его составе следующие.

ISO 9000:2000 Quality management systems — Fundamentals and vocabulary. Системы управления качеством — Основы и словарь. (Аналог ГОСТ Р-2001).

ISO 9001:2000 Quality management systems — Requirements. Models for quality assurance in design, development, production, installation, and servicing. Системы управления качеством — Требования. Модели для обеспечения качества при проектировании, разработке, коммерциализации, установке и обслуживании. Определяет общие правила обеспечения качества результатов во всех процессах жизненного цикла. (Аналог ГОСТ Р-2001). ISO 9002:1994 Quality systems — Model for quality assurance in production, installation and servicing. Системы качества — Модель для обеспечения качества при коммерциализации, установке и обслуживании.

ISO 9003:1994 Quality systems — Model for quality assurance in final inspection and test. Системы качества — Модель обеспечения качества при финальном инспектировании и тестировании. ISO 9004:2000 Quality management systems — Guidelines for performance improvements. Системы управления качеством. Руководство по улучшению деятельности. (Аналог ГОСТ Р-2001).



.....
Понятие качества ПО в виде совокупности атрибутов качества определяется стандартом ISO 9126. В этом стандарте выделено 6 основных *факторов качества ПО*, называемых также целями, и каждый из факторов качества описывается при помощи нескольких входящих в него *атрибутов*.
.....

Ниже приведен полный список этих атрибутов по стандарту ISO 9126:2001:

- Функциональность (определяет, что делает ПО, какие задачи оно решает).
- Пригодность к определенной работе (suitability).
- Точность, правильность (accuracy).
- Способность к взаимодействию (interoperability).
- Соответствие стандартам и правилам (compliance).
- Защищенность (security).
- Надежность («насколько надежно ПО работает»).
- Зрелость, завершенность (обратна к частоте отказов) (maturity).
- Устойчивость к отказам (fault tolerance).
- Способность к восстановлению работоспособности при отказах (recoverability).
- Соответствие стандартам надежности (reliability compliance, добавлен в 2001).
- Практичность, удобство использования (насколько удобно пользоваться ПО).
- Понятность (understandability).

- Удобство обучения (learnability).
- Работоспособность (operability).
- Привлекательность (attractiveness, добавлен в 2001).
- Соответствие стандартам практичности (usability compliance, добавлен в 2001).
- Эффективность («насколько эффективно работает ПО»).
- Временные характеристики (time behaviour).
- Использование ресурсов (resource utilisation).
- Соответствие стандартам эффективности (efficiency compliance, добавлен в 2001).
- Сопровождаемость («насколько удобно вносить изменения и поправки в ПО»).
- Анализируемость (analyzability).
- Изменяемость, удобство внесения изменений (changeability).
- Риск возникновения неожиданных эффектов при внесении изменений (stability).
- Контролируемость, удобство проверки (testability).
- Соответствие стандартам сопровождаемости (maintainability compliance, добавлен в 2001).
- Переносимость, мобильность (насколько удобно переносить ПО в другое окружение).
- Адаптируемость (adaptability).
- Устанавливаемость, удобство установки (installability).
- Способность к сосуществованию с другим ПО (coexistence).
- Удобство замены другого ПО данным (replaceability).
- Соответствие стандартам переносимости (portability compliance, добавлен в 2001).

8.5 Методы контроля качества

Как контролировать качество системы? Как точно узнать, что программа делает именно то, что нужно, и ничего другого? Как определить, что она достаточно надежна, переносима, удобна в использовании?



.....
Ответы на этот вопрос можно получить с помощью процессов верификации и валидации.
.....

Верификация — проверка того, что продукт делался правильно, т. е. проверка того, что он разрабатывался в соответствии со всеми требованиями по отношению к процессу и этапам разработки.

Валидация — проверка того, что сам продукт правилен, т. е. установление того, что он удовлетворяет требованиям, ожиданиям пользователя, заказчика и других заинтересованных сторон.

Эффективность верификации и валидации, как и эффективность разработки ПО, зависит от точности и корректности формулировки требований к программному продукту.

Выделяют следующие *методы обеспечения качества ПО*.

Тестирование — частный вариант верификации и валидации, состоящий в наблюдении за работой ПО в специальных условиях, на некоторых данных с целью проверки соответствия его свойств требованиям. Такого рода специальные условия называют тестовыми или просто тестами. Тестирование бывает следующих видов.

Функциональное (оно же черного ящика) — это тестирование проверяет соответствие системы требованиям, пользуясь при этом как для создания тестов, так и для оценки степени соответствия только самими требованиями.

Структурное (оно же белого ящика) — тесты создаются на основе знаний о структуре самой системы и о том, как она работает. Для оценки степени соответствия требованиям могут привлекаться дополнительные знания о прослеживании требований в определенные условия на значения внутренних данных системы (например, в ограничения на значения параметров вызовов, результатов и локальных переменных).

Тестирование производительности. При этом виде тестирования измеряются показатели производительности системы и определяется, насколько они соответствуют требуемым, а также, не может ли экстраполяция этих данных привести к нарушению требований в других возможных ситуациях. В случае возникновения серьезных подозрений во втором случае, тесты производительности дорабатываются для проверки этих дополнительных ситуаций.

Нагрузочное тестирование. Это особый вид тестирования производительности, целью которого является определение характеристик производительности системы при большой нагрузке на нее (большом количестве пользователей, интенсивном обмене данными с другими системами, большим объемом передаваемых/используемых данных и пр.).

Тестирование на отказ имеет целью обнаружение ситуаций, в которых система выходит из строя, хотя не должна делать этого.

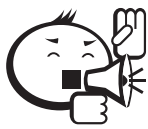
Тестирование удобства использования проверяет, насколько система удобна в использовании. Делается это на основе измерения времени, необходимого на обучение работе с системой, и производительности труда при работе с системой различных групп пользователей (уже знакомых с ней, незнакомых, опытных в данной предметной области, новичков и пр.).

Тестирование переносимости проверяет работоспособность системы в разных системных окружениях.

Тестирование на соответствие проверяет строгое соответствие требованиям любого рода, зафиксированным в достаточно строгом и полном виде (спецификациям, стандартам, корпоративной политике, и пр.).

Модульное, интеграционное и системное тестирование — различаются масштабом целей. Модульное тестирование предназначено для проверки правильности работы отдельных модулей, вне зависимости от окружения, интеграционное — для проверки того, что модули корректно работают вместе (используют друг друга, не нарушая взаимных ограничений на такое использование), системное — для проверки правильности работы системы в целом и ее способности решать заявленные задачи в различных ситуациях.

8.6 Ошибки в ПО



.....
Ошибками в ПО, вообще говоря, являются все возможные несоответствия между демонстрируемыми характеристиками его качества и предписанными требованиями и, иногда, ожиданиями пользователей.

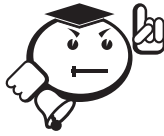
В англоязычной литературе используется несколько терминов, часто переводящихся как «ошибка» на русский язык:

- defect — самое общее нарушение каких-либо требований или ожиданий, не обязательно проявляющееся вовне (к дефектам относятся и нарушения стандартов кодирования, недостаточная гибкость системы и пр.);
- failure — нарушение требований, проявляющееся при каком-то реальном сценарии работы ПО, это, скорее, проявление ошибки;
- fault — ошибка в коде программы, вызывающая нарушения требований при работе (failures), то место, которое надо исправить. Хотя это понятие используется довольно часто, оно, вообще говоря, не вполне четкое, поскольку для устранения нарушения можно исправить программу в нескольких местах. Что именно надо исправлять, зависит от дополнительных условий, выполнение которых мы хотим при этом обеспечить;
- error — используется в двух смыслах. Первый — это ошибка в ментальной модели программиста, которая заставляет его делать ошибки в коде (faults). Второй смысл — это некорректные значения данных (выходных или внутренних), которые возникают при ошибках в работе программы.

Первое место в неформальном состязании за место «самой дорого обошедшейся ошибки в ПО» долгое время удерживала ошибка, приведшая к неудаче первого запуска ракеты Ариан-5 4 июня 1996 года, стоившая около 500 М. После произошедшего 14 августа 2003 года обширного отключения электричества на северо-востоке Северной Америки, стоившего экономике США и Канады от 4 до 10 миллиардов долларов, это место закрепилось за вызвавшей его ошибкой в системе управления электростанцией.

При разработке ПО основная нагрузка лежит на человеке как с точки зрения как такового программирования, так и предоставления соответствующих физических, экономических, химических или математических моделей описания решаемых

мой задачи. Таким образом интеллектуальная возможность человека или коллектива является решающей при разработке ПО и основной при ошибочных действиях или выборе модели.



.....
Дейкстра выделяет три интеллектуальные возможности человека, используемые при разработке ПО:

- способность к перебору,
 - способность к абстракции,
 - способность к математической индукции.
-

Способность человека к перебору связана с возможностью последовательного переключения внимания с одного предмета на другой, позволяя *узнавать* искомый предмет. Эта способность весьма ограничена — в среднем человек может уверенно (не сбиваясь) перебирать в пределах 1000 предметов (элементов). Человек должен научиться действовать с учетом этой своей ограниченности. Средством преодоления этой ограниченности является его способность к абстракции, благодаря которой человек может объединять разные предметы или экземпляры в одно понятие, заменять множество элементов одним элементом (другого рода). Способность человека к математической индукции позволяет ему справляться с бесконечными последовательностями.

При разработке ПО человек имеет дело с системами. Под *системой* будем понимать совокупность взаимодействующих (находящихся в отношениях) друг с другом элементов. ПО можно рассматривать как пример системы. Логически связанный набор программ является другим примером системы. Любая отдельная программа также является системой. Понять систему — значит, осмысленно перебрать все пути взаимодействия между ее элементами. В силу ограниченности человека к перебору будем различать простые и сложные системы.

Под *простой* будем понимать такую систему, в которой человек может уверенно перебирать все пути взаимодействия между ее элементами, а под *сложной* будем понимать такую систему, в которой он этого делать не в состоянии. Между простыми и сложными системами нет четкой границы, поэтому можно говорить и о промежуточном классе систем: к таким системам относятся программы, о которых программистский фольклор утверждает, что «в каждой отлаженной программе имеется хотя бы одна ошибка».

При разработке ПО мы не всегда можем уверенно знать о всех связях между ее элементами из-за возможных ошибок. Поэтому полезно уметь оценивать сложность системы по числу ее элементов: числом потенциальных путей взаимодействия между ее элементами, т. е. $n!$, где n — число ее элементов. Систему назовем *малой*, если $n < 7$ ($6! = 720 < 1000$), систему назовем *большой*, если $n > 7$. При $n = 7$ имеем промежуточный класс систем. Малая система всегда проста, а большая может быть как простой, так и сложной. Задача технологии программирования — научиться делать большие системы простыми.

Полученная оценка простых систем по числу элементов широко используется на практике. Так, для руководителя коллектива весьма желательно, чтобы в нем не было больше шести взаимодействующих между собой подчиненных.

При разработке и использовании ПС мы многократно имеем дело с преобразованием (переводом) информации из одной формы в другую (см. рис. 8.7). Заказчик формулирует свои потребности в ПО в виде некоторых требований. Исходя из этих требований, разработчик создает внешнее описание ПО, используя при этом спецификацию (описание) заданной аппаратуры и, возможно, спецификацию базового программного обеспечения. На основании внешнего описания и спецификации языка программирования создаются тексты программ ПО на этом языке. По внешнему описанию ПС разрабатывается также и пользовательская документация. Текст каждой программы является исходной информацией при любом ее преобразовании, в частности при исправлении в ней ошибки. Пользователь на основании документации выполняет ряд действий для применения ПО и осуществляет интерпретацию получаемых результатов. Везде здесь, а также в ряде других процессов разработки ПО имеет место указанный перевод информации.

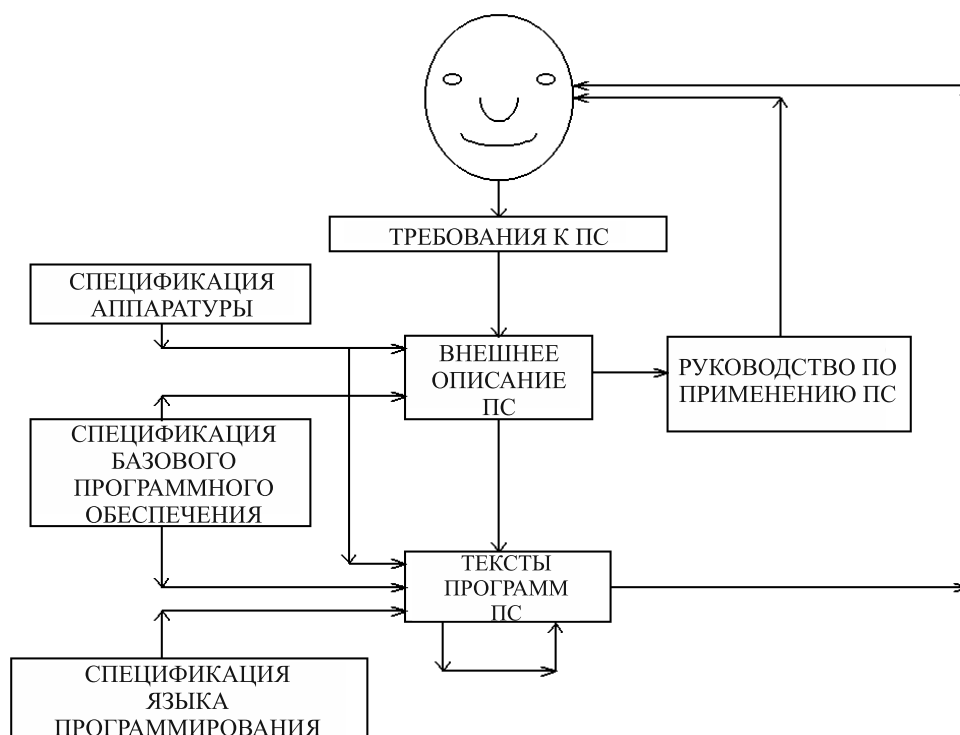


Рис. 8.7 – Грубая схема разработки и применения ПС

На каждом из этих этапов перевод информации может быть осуществлен неправильно, например из-за неправильного понимания исходного представления информации. Возникнув на одном из этапов разработки ПО, ошибка в представлении информации преобразуется в новые ошибки результатов, полученных на последующих этапах разработки, и, в конечном счете, окажется в ПС.

Чтобы понять природу ошибок при переводе, рассмотрим модель, изображенную на рис. 8.8. На ней человек осуществляет перевод информации из представления А в представление В. При этом он совершает четыре основных шага перевода:

- он получает информацию, содержащуюся в представлении А, с помощью своего читающего механизма R;
- он запоминает полученную информацию в своей памяти М;
- он выбирает из своей памяти преобразуемую информацию и информацию, описывающую процесс преобразования, выполняет перевод и посылает результат своему пишущему механизму W;
- с помощью этого механизма он фиксирует представление В.

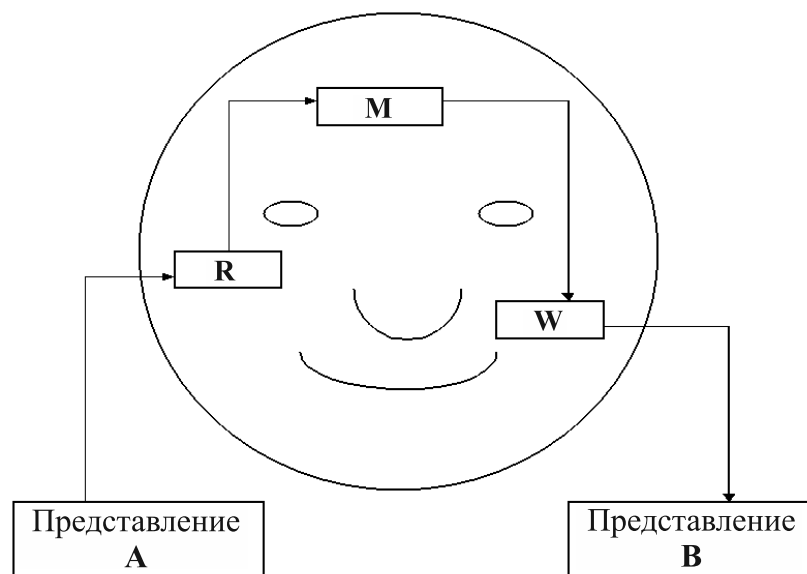


Рис. 8.8 – Модель перевода

На каждом из этих шагов человек может совершить ошибку разной природы. На первом шаге способность человека «читать между строк» (способность, которая часто оказывается полезной, позволяя ему понимать текст, содержащий неточности или даже ошибки) может стать причиной ошибки в ПО. Ошибка возникает в том случае, когда при чтении документа А человек, пытаясь восстановить недостающую информацию, видит то, что он ожидает, а не то, что имел в виду автор документа А.

В этом случае лучше было бы обратиться к автору документа за разъяснениями. При запоминании информации человек осуществляет ее осмысливание (здесь важен его уровень подготовки и знание предметной области, к которой относится документ А). И если он поверхностно или неправильно поймет, то информация будет запомнена в искаженном виде. На третьем этапе забывчивость человека может привести к тому, что он может выбрать из своей памяти не всю преобразуемую информацию или не все правила перевода, в результате чего перевод будет осуществлен неверно. Это обычно происходит при большом объеме плохо организованной информации. И наконец, на последнем этапе стремление человека быстрее зафиксировать информацию часто приводит к тому, что представление этой информации оказывается неточным, создавая ситуацию для последующей неоднозначной ее интерпретации.

Учитывая рассмотренные особенности действий человека при переводе, можно указать следующие пути борьбы с ошибками:

- сужение пространства перебора (упрощение создаваемых систем),
- обеспечение требуемого уровня подготовки разработчика (это функции менеджеров коллектива разработчиков),
- обеспечение однозначности интерпретации представления информации,
- контроль правильности перевода (включая и контроль однозначности интерпретации).



Контрольные вопросы по главе 8

- 1) Что такое качество ПО?
- 2) Что такое требования к ПО?
- 3) Что такое потребности при разработке ПО?
- 4) На сколько этапов разбита формулировка потребностей?
- 5) Что регламентируют правила работы с требованиями к ПО?
- 6) Что называют вариантом использования (use case)?
- 7) Какие методы контроля качества вы знаете?
- 8) Что такое методы контроля качества верификации?
- 9) Что такое методы контроля качества эффективности?
- 10) Что такое метод обеспечения качества ПО?

ЗАКЛЮЧЕНИЕ

Все программы для компьютера сложны. Сложность реального мира уже сама по себе достаточно веская причина для сложности программного обеспечения, призванного решить какую-то его задачу. Дополнительная сложность обусловлена изменением требований к системе в процессе ее разработки. То, что система представляет собой в проекте, очень сильно может отличаться от своей реализации. Организация коллективной работы над системой — еще одна причина сложности программ. Чем больше разработчиков, тем сложнее организовать совместную работу.

Для разработки программ необходим язык общения человека с компьютером. По аналогии общения людей, этих языков много и каждый решает свои задачи. Одним из компьютерных языков, который широко распространен в мире программистов, является язык C++.

Объектно-ориентированная технология программирования основывается на так называемой объектной модели. Основными свойствами этой модели являются: абстрагирование, инкапсуляция, модульность, иерархичность, типизация, параллелизм и сохраняемость. Это не новые принципы, но в совокупности они встречаются только в объектном моделировании. Объектная модель является основой трех китов разработки: объектно-ориентированного анализа, объектно-ориентированного проектирования и объектно-ориентированного программирования.

Данное пособие посвящено краткому введению в мир объектно-ориентированного программирования на языке C++.

ЛИТЕРАТУРА

- [1] Майерс Г. Надежность программного обеспечения / Г. Майерс. — М. : Мир, 1980.
- [2] Дал У. Структурное программирование / У. Дал, Э. Дейкстра, К. Хоор. — М. : Мир, 1975.
- [3] Держинский Ф. Я. Концепция и опыт реализации методических средств программной инженерии / Ф. Я. Держинский, И. М. Калиниченко. — М. : ЦНИИ информации и технико-экономических исследований по атомной науке и технике, 1988.
- [4] Турский В. Методология программирования / В. Турский. — М. : Мир, 1981.
- [5] Буч Г. Объектно-ориентированное проектирование с примерами применения / Г. Буч. — М. : Конкорд, 1992.
- [6] Брукс Ф. П. мл. Как проектируются и создаются программные комплексы / Ф. П. мл. Брукс. — М. : Наука, 1979.
- [7] Хьюз Дж. Структурный подход к программированию / Дж. Хьюз, Дж. Мичтом. — М. : Мир, 1980.
- [8] Характеристики качества программного обеспечения / Б. Боэм [и др.]. — М. : Мир, 1981.
- [9] Липаев В. В. Качество программного обеспечения / В. В. Липаев. — М. : Финансы и статистика, 1983.
- [10] Шнейдерман Б. Психология программирования / Б. Шнейдерман. — М. : Радио и связь, 1984.
- [11] Кауфман В. Ш. Языки программирования. Концепции и принципы / В. Ш. Кауфман. — М. : Радио и связь, 1993.
- [12] Требования и спецификации в разработке программ. — М. : Мир, 1984.
- [13] Агафонов В. Н. Спецификация программ: понятийные средства и их организация / В. Н. Агафонов. — Новосибирск : Наука (Сибирское отделение), 1987.

- [14] Липаев В. В. Мобильность программ и данных в открытых информационных системах / В. В. Липаев, Е. Н. Филипов. — М. : Научная книга, 1997.
- [15] Жоголев В. А. Введение в технологию программирования (конспект лекций) / В. А. Жоголев. — М. : ДИАЛОГ-МГУ, 1994.
- [16] Зелковец М. Принципы разработки программного обеспечения / М. Зелковец, А. Шоу, Дж. Гэннон. — М. : Мир, 1982.
- [17] Зиглер К. Методы проектирования программных систем / К. Зиглер. — М. : Мир, 1985.
- [18] Фокс Дж. Программное обеспечение и его разработка / Дж. Фокс. — М. : Мир, 1985.
- [19] Характеристики качества программного обеспечения / Б. Боэм [и др.]. — М. : Мир, 1981.
- [20] Фуксман А. Л. Технологические аспекты создания программных систем / А. Л. Фуксман. — М. : Статистика, 1979.
- [21] Ван Тассел Д. Стил, разработка, эффективность, отладка и испытание программ / Д. Ван Тассел. — М. : Мир, 1985.
- [22] Безбородов Ю. М. Индивидуальная отладка программ / Ю. М. Безбородов. — М. : Наука, 1982.
- [23] Горбунов-Посадов М. М. Конфигурации программ. Рецепты безболезненных изменений / М. М. Горбунов. — М. : Малип, 1994.
- [24] Фути К. Языки программирования и схемотехника СБИС / К. Фути, Н. Судзуки. — М. : Мир, 1988.
- [25] Фаулер М. UML в кратком изложении / М. Фаулер, К. Скотт. — М. : Мир, 1999.
- [26] Крачтен Ф. Введение в Rational unified process / Ф. Крачтен. — М. : Вильямс, 2002.
- [27] Кауфман В. Ш. Языки программирования. Концепции и принципы / В. Ш. Кауфман. — М. : Радио и связь, 1993.
- [28] Бен-Ари М. Языки программирования. Практический сравнительный анализ / М. Бен-Ари. — М. : Мир, 2000.
- [29] Константайн Л. Разработка программного обеспечения / Л. Константайн, Л. Локвуд. — СПб. : Питер, 2004.
- [30] Торрес Р. Дж. Практическое руководство по проектированию и разработке пользовательского интерфейса / Р. Дж. Торрес. — М. : Вильямс, 2002.
- [31] Приемы объектно-ориентированного проектирования. Паттерны проектирования / Э. Гамма [и др.]. — СПб. : Питер-ДМК, 2001.

- [32] Брауде Э. Дж. Технология разработки программного обеспечения / Брауде Э. Дж. — СПб. : Питер, 2004.
- [33] Страуструп Б. Язык программирования C++ / Б. Страуструп. — Киев : Диасофт, 2001.
- [34] Бабэ Б. Просто и ясно о BorlandC++. Версии 4.0 и 4.5 / Б. Бабэ. — М. : Бином, 2001.
- [35] Вайнер Л. С. C++ изнутри / Р. Вайнер, Л. С. Пинсон. — Киев : Диасофт, 1993.
- [36] Дьюхарст С. Программирование на C++ / С. Дьюхарст. — Киев : Диасофт, 1993.
- [37] Лукас П. C++ под рукой / П. Лукас. — Киев : Диасофт, 1993.
- [38] Намиот Д. Е. Основные особенности языка программирования C++. Реализация Turbo C++ / Д. Е. Намиот. — М. : Память, 1991.
- [39] Подбельский В. В. Язык C++ : учеб. пособие / В. В. Подбельский. — М. : Финансы и статистика, 1995.
- [40] Рассохин Д. От Си к Си++ / Д. Рассохин. — М. : Эдель, 1993.
- [41] Романов В. Ю. Программирование на языке C++. Практический подход / В. Ю. Романов. — М. : Компьютер, 1993.
- [42] Страуструп Б. Язык программирования C++ / Б. Страуструп. — Киев : Диасофт, 1993. — Ч. 1.
- [43] Страуструп Б. Язык программирования C++ / Б. Страуструп. — Киев : Диасофт, 1993. — Ч. 2.
- [44] Фейсон Т. Объектно-ориентированное программирование на Borland C++ 4.5 / Т. Фейсон. — Киев : Диалектика, 1996.
- [45] Эллис М. Справочное руководство по языку программирования C++ с комментариями / М. Эллис, Б. Страуструп. — М. : Мир, 1992.
- [46] Соммервилл И. Инженерия программного обеспечения / И. Соммервилл. — М. : Вильямс, 2002.
- [47] Ройс У. Управление проектами по созданию программного обеспечения / У. Ройс. — М. : Лори, 2002.
- [48] Якобсон А. Унифицированный процесс разработки программного обеспечения / А. Якобсон, Г. Буч, Дж. Рамбо. — СПб. : Питер, 2002.
- [49] Брауде Э. Дж. Технология разработки программного обеспечения / Э. Дж. Брауде. — СПб. : Питер, 2004.
- [50] Бек К. Экстремальное программирование / К. Бек. — СПб. : Питер, 2002.

- [51] Леффингуэлл Д. Принципы работы с требованиями к программному обеспечению. Унифицированный подход / Д. Леффингуэл, Д. Уидриг. — М. : Вильямс, 2002.
- [52] Иванова Г. С. Объектно-ориентированное программирование / Г. С. Иванова, Г. Н. Ничушкина, Е. К. Пугачев. — М. : МГТУ им. Баумана, 2001.
- [53] Ишкова Э. А. С++ начала программирования / Э. А. Ишкова. — М. : БИНОМ, 2001.
- [54] Ирэ П. Объектно-ориентированное программирование / П. Ирэ. — Киев : Диасофт, 1995.
- [55] Бугг Г. Объектно-ориентированное программирование с примерами применения / Г. Бугг. — М. : Финансы и статистика, 1992.
- [56] Дал У. Структурное программирование / У. Дал, Э. Дейкстра, К. Хоор. — М. : Мир, 1975.

ГЛОССАРИЙ

Абстрактный объект (Abstract object) — базовый объект, который затем может быть наследован другими объектами.

Библиотека классов (Class Library) — набор готовых объектов общего назначения.

Виртуальный метод (virtual method) — метод, адрес которого известен только в момент выполнения программы.

Дерево классов (class tree) — синоним термина «иерархия объектов».

Деструктор (destructor) — специальная процедура для освобождения памяти, занятой объектом.

Иерархия классов (class hierarchy) — позволяет создавать классы-потомки существующих классов таким образом, что классы-потомки наследуют все данные и методы классов-предков.

Инкапсуляция (Incapsulation) — объединение в типе данных ОБЪЕКТ полей данных и методов (процедур и функций), работающих с этими данными.

Класс (Class) — синоним термина тип данных ОБЪЕКТ.

Конструктор (Constructor) — специальная процедура, инициализирующая экземпляр объекта, содержащий виртуальные методы, путем установки связи между экземпляром объекта и таблицей виртуальных методов.

Метод (Method) — процедура или функция, определенная внутри объекта для работы с данными объекта.

Множественное наследование (Multiple inheritance) — наличие у объекта нескольких родителей.

Наследование (Inheritance) — процесс получения данных и методов от объекта-предка. Возможно многоуровневое наследование.

Объект (Object) — тип данных ОБЪЕКТ — совокупность данных и процедур-операций над этими данными (называемых методами).

Объект-потомок (Descendant object) — объект, наследующий методы и данные от объекта-предка.

Объект-предок (Ancestor object) — объект, методы и данные которого наследует объект-потомок.

Объекты-контейнеры (Container object) — объекты, способные хранить в себе другие объекты.

Параметр Self (Self parametr) — формальный параметр, передаваемый каждому объекту. Содержит указатель на конкретный экземпляр объекта.

Передача сообщения (Message passing) — вызов метода для конкретного экземпляра объекта.

Переопределение (Override) — процесс создания у объекта-потомка новых методов с теми же именами, что и у объекта-предка, но с новыми функциями, это делает возможным расширение объектов.

Подкласс (Subclass) — это класс-потомок.

Позднее связывание (Late binding) — ситуация, при которой адрес вызываемого метода неизвестен до момента выполнения программы.

Полиморфизм (Polimorphism) — возможность использования методов с одинаковыми именами для работы с различными типами данных.

Раннее связывание (Early binding) — ситуация, при которой адрес вызываемого метода известен в момент компиляции/компоновки.

Расширяемость (Extendibility) — метод, который изменяет или расширяет функции унаследованного метода и заменяет его в объекте-потомке.

Связывание (Binding) — процесс, в результате которого вызывающая программа/подпрограмма получает адрес вызываемой подпрограммы.

Статистический метод (Static method) — метод, вызываемый с использованием раннего связывания (адрес которого известен в момент компиляции/компоновки).

Суперкласс (Superclass) — в иерархии объектов каждый класс имеет только одного непосредственного предка, называемого «предком» или «суперклассом».

Таблица виртуальных методов (Virtual method table) — таблица, хранимая в сегменте данных.

Экземпляр (Instance) — это объявленная переменная типа объект.

Учебное издание

Катаев Михаил Юрьевич

**ОБЪЕКТНО-ОРИЕНТИРОВАННОЕ
ПРОГРАММИРОВАНИЕ**

Учебное пособие

Корректор Осипова Е. А.

Компьютерная верстка Перминова М. Ю.

Подписано в печать 24.06.13 Формат 60х84/8.

Усл. печ. л. 24,65. Тираж 300 экз. Заказ

Издано в ООО «Эль Контент»

634029, г. Томск, ул. Кузнецова д. 11 оф. 17

Отпечатано в Томском государственном университете
систем управления и радиоэлектроники.

634050, г. Томск, пр. Ленина, 40

Тел. (3822) 533018.