Министерство образования Российской Федерации
Владимирский государственный университет

И.Р. ДУБОВ В.В. ВЛАСЕНКО

ОБЪЕКТНО-ОРИЕНТИРОВАННОЕ ПРОГРАММИРОВАНИЕ

Учебное пособие

Владимир 2003

УДК 519.682.2
Д79

Рецензенты
Кандидат технических наук, доцент
зав. кафедрой информатики и информационных технологий
Владимирского государственного педагогического университета
Ю.А. Медведев

Доктор технических наук, профессор
зав. кафедрой информационных систем и информационного менеджмента
Владимирского государственного университета
А.В. Костров

Печатается по решению редакционно-издательского совета
Владимирского государственного университета

Дубов И.Р., Власенко В.В.

Д79 Объектно-ориентированное программирование: Учеб. пособие / Владим. гос. ун-т. Владимир, 2003. 68 с.
ISBN 5-89368-374-9

Рассматриваются основные положения объектно-ориентированного программирования и средства поддержки в языке C++. Особое внимание уделено средствам реализации позднего связывания объектов и методов, обобщенным классам и особенностям использования множественного наследования.

Предназначено для студентов специальности 220100 "Вычислительные машины, комплексы, системы и сети" при изучении дисциплины "Технология программирования" дневной и заочной форм обучения.

Ил. 6. Библиогр.: 4 назв.

УДК 519.682.2

ISBN 5-89368-374-9

© Владимирский государственный университет, 2003
© Дубов И.Р., Власенко В.В., 2003

ВВЕДЕНИЕ

При создании программных систем можно выделить три основных стадии: анализ, проектирование, программирование. Задача анализа – построение модели предметной области. Задача проектирования – разработка модели программной системы, учитывающей предметную область и будущую реализацию. Задача программирования (кодирования) – реализация программной системы с помощью конкретных языков программирования. Объектно-ориентированный подход к выполнению всех перечисленных стадий стал уже стандартом при создании сложных программных продуктов.

В настоящее время имеется обширная литература по конкретным языкам программирования, немногочисленная литература по проектированию и единичные издания, в которых затрагиваются вопросы анализа. Однако эти вопросы рассматриваются изолированно. Цель данного пособия – свести воедино некоторые вопросы проектирования и программирования. При этом осуществляется попытка рассмотреть применение средств объектно-ориентированного программирования вместе с реализацией на достаточно низком уровне.

Традиционно изучение программирования начинается со структурного программирования. Опыт преподавания показывает, что переход к объектному стилю мышления требует от обучающегося значительных усилий. В данном пособии сделана попытка сгладить острые углы этого перехода, в нем больше внимания уделяется тем вопросам, которые вызывают наибольшую сложность при изучении (позднее связывание, виртуальные базовые классы, перегрузка операторов). Таким образом, пособие следует рассматривать как дополнение к имеющейся обширной литературе по объектно-ориентированному подходу, ни в коей мере не претендующее на полноту изложения.

В качестве языка программирования выбран C++. Это связано с тем, что в этом языке реализовано большинство положений объектно-ориентированного подхода, например, такие возможности, как обобщенные классы, множественное наследование, перегрузка операторов отсутствуют в языке Object Pascal. Кроме этого, в быстро распространяющихся языках программирования Java и C# за основу был взят синтаксис языка C++. Для упрощения работы с пособием в него включены начальные сведения о языке C++ с обсуждением некоторых деталей, важных для объектно-ориентированного программирования.

Глава 1. **ОСНОВНЫЕ ЭЛЕМЕНТЫ ЯЗЫКА С++**

1.1. Примеры простейших программ

Основные элементы программы на языке С++ показаны в нижеследующем примере.

```
1:    #include <iostream. h> //директива препроцессора
2:    int main  //главная функция
3:    {   //операторная скобка
4:      cout << "Hello, world!\n";
          //cout - глобальная переменная для выходного потока
          //<< - перегруженный оператор вставки в поток
5:      return 0; //возврат из функции (здесь - возврат в ОС)
6:    }   //операторная скобка
```

Директивы include обеспечивают подключение библиотек. В данном примере библиотека iostream содержит средства ввода/вывода. В ней, например, объявлена глобальная переменная cout, связанная с устройством вывода. В этой же библиотеке объявлена переменная cin, связанная с устройстом ввода. В следующем примере приводятся объявления переменных, операторы и реализуется ввод данных.

```
7:   #include <iostream.h>
8:   int main
9:   {
10:    int integer1, integer2, sum; //объявление переменных
                                    //целого типа
11:    cout << "введите 1-е число";  //вставка в поток
12:    cin >> integer1; //извлечение из входного потока
13:    cout << "введите 2-е число";  //вставка в поток
14:    cin >> integer2;  //извлечение из входного потока
15:    sum=integer1+integer2;  //операторы сложения
                               //и присваивания
16:    cout << "сумма равна" << sum << endl;
          //endl - манипулятор потока: вывод содержимого
          // буфера и новая строка
17:  }
```

Препроцессор – это часть компилятора, выполняющая предварительную обработку исходного текста программы. Препроцессор заменяет определенным образом предназначенные для него директивы и формирует окончательный исходный модуль, который подвергается трансляции.

Директивы включения файлов

Директивы включения файлов используются для включения в исходный модуль программы заголовочных файлов библиотек. Символы <> предназначены для указания стандартных заголовочных файлов, поиск которых выполняется в системном каталоге библиотечных файлов. В симво-

лах "" указывается пользовательский заголовочный файл, поиск которого ведется сначала в текущем каталоге, а затем – в системном каталоге.

```
 18:  #include <iostream.h>
 19:  #include "myio.h"
```

Директивы условной компиляции

Директивы условной компиляции позволяют исключать из обработки фрагменты исходного текста программы. Управление трансляцией необходимо в тех случаях, когда текст содержит код программы для выполнения на различных платформах, в текст включены отладочные операторы.

```
20:  #define DEBUG //определение идентификатора, идентификатор
                   //"виден" только препроцессору
21:  #ifdef  DEBUG
22:  ...   //фрагмент транслируется, если идентификатор
           //был определен директивой #define
23:  #endif

24:  #ifdef DEBUG  //идентификатор
25:  ...   //фрагмент транслируется, если идентификатор
           //был определен
26:  else
27:  ...   //фрагмент транслируется, если идентификатор
           //не был определен
28:  #endif

29:  #ifndef DEBUG  //идентификатор
30:  ...   //фрагмент транслируется, если идентификатор
           //не был определен
31:  #endif
```

Один и тот же файл может быть включен директивой include в несколько других файлов. Директивы условной компиляции позволяют предотвратить повторое включение файлов.

```
32:   #ifndef STRINGH
33:   #define STRINGH
34:   #include "mystring.h"
35:   ...
36:   #endif
```

Операторы

В языке C++ имеются следующие основные операторы. Оператор присваивания "=" позволяет присвоить новое значение переменной, в правой его части должно быть выражение. В языке предусмотрены основные арифметические операции: "()" – скобки; "*" – умножение; "/" – деление; "%" – деление по модулю. Логические операции имеют следующие обозначения: "==" – равенство; "! =" – неравенство; ">" – больше; "<" – меньше; ">=" – больше или равно; "<=" – меньше или равно. Операции булев-

ской логики обозначаются следующим образом: "!" – логическое "НЕ"; "&&" – логическое «И»; || – логическое«ИЛИ». Логические выражения имеют результат 0 или 1, где 0 обозначает ложь, а 1 – истину.

```
37:   (1==2) //ложь
38:   (1!=2) //истина
```

Строгого различия между логическими и арифметическими операторами нет при вычислении выражения:

```
39:    i = (5==5)*3
40:    cout << i; //результат 3
```

Управляющая конструкция ветвления

Формально ветвление определяется следующим образом: *if (выражение) оператор*. Оператор выполняется тогда, когда выражение истинно, а точнее – тогда, когда выражение не равно нулю.

```
41:   #include <iostream.h>
42:   int main (){
43:     int num1, num2;
44:     num1 = 3;
45:     num2 = 3;
46:     if (num1 > num2) num1= num2;
47:     cout << num1;
48:     return 0;
49:   }
```

1.2. Типы данных

Стандартные типы данных

В C++ имеются следующие целочисленные типы, которые могут быть дополнительно объявлены беззнаковыми с помощью модификатора `unsigned`.

```
50:   char        //символьный (однобайтовый)
51:   short
      short int   //короткое целое
52:   int         //целое (слово)
53:   long
      long int    //длинное целое
```

Вещественные типы данных различаются диапазоном представления чисел и точностью.

```
54:   float машинное слово
55:   double два слова
56:   long double3 или 4 слова
```

Константы

Рассмотрим основные виды констант.

```
57:   20        //десятичная
58:   024       //восьмеричная
```

```
59:   0*14     //шестнадцатиричная
60:   128u     //целое без знака
61:   1L       //длинное целое
62:   1024UL   //длинное целое без знака
63:   1.01E-3  //экспоненциальная форма
64:   12.1F    //обычная точность
65:   1.0L     //двойная точность
```

В символьные строки можно вставлять символы, не имеющие печатаемых знаков, это так называемые escape – последовательности.

```
66:   "\n"  //новая строка
67:   "\r"  //возврат каретки
68:   "\a"  //предупреждение
69:   "\\"  //«\»
70:   "\?"  //?
71:   "\'"  //'
72:   "\""  //"
```

Возможно использование восьмеричных кодов символов в escape-последовательностях.

```
73:   \7   //bell
74:   \0   //null
75:   \12  //new line
76:   \062 //'2'
```

Инициализация переменных

Оператор объявления переменной является в C++ исполняемым оператором. Переменные могут быть объявлены без инициализации, с инициализацией в явном виде и с инициализацией в неявном виде.

```
77:   int x; //неинициализированная переменная
78:   double price = 109.9, discount = 0.16;
              //явная инициализация
79:   double salePrice(price*discount);
              //неявная инициализация
```

Константные типы

Типизированные именованные константы задаются с помощью зарезервированного слова const.

```
80:   const double pi = 2.14159; //объявление константы
81:   double x = pi; //правильно
82:   pi = x; //ошибка трансляции
```

Производный перечисляемый тип

Перечисляемый тип задает перечень констант, которые могут быть присвоены переменной данного типа. Тип обявляется с зарезервированным словом enum.

```
83:   enum TestStatus {false, pass, stop};
        //TestStatus - идентификатор для нового типа
84:   int main (){
85:     TestStatus test; //объявление переменной
```

```
                          //типа TestStatus
86:     test = pass;
87:     if (test == stop)
88:     ...
89:     enum {noon = 0, midnight = 1}; //объявление констант
                                       //без введения нового типа
90:     int time = noon;
91:     ...
92:   }
```

Производный тип структура

Структура позволяет конструировать новые агрегирующие типы данных. Формально новый тип объявляется следующим образом: `struct имя-нового-типа {объявление полей}`. Здесь struct – ключевое слово.

```
93:   struct Time{
94:     int hour;
95:     int minute;
96:     int second;
97:   }
98:   main (){
99:     Time t1, t2;
100:    Time t3 = {12, 20, 0}; //инициализация структуры
101:    t1.hour = 2;
102:    t1.minute = 30;
103:    t1.second = 10;
104:    t2.hour=3;
105:    if (t1.hour > t2.hour)...
106:    t2 = t1; //присваивание побайтное
107:    cout << t2.minute << endl;
108:  }
```

Типизированные указатели

Переменная типа указатель предназначена для хранения адреса некоторого объекта. Тип такой переменной определяется типом того объекта, на который указывает указатель. Для использования указателей имеется два оператора. Одноместный оператор обращения по адресу, его также называют оператором раскрытия ссылки или оператором разыменовывания, обозначается "*". Второй одноместный оператор, обозначаемый "&", возвращает адрес переменной.

```
109:  int *ip1, *ip2; // ip1 – переменная указатель;
                      //*ip1 – референт указателя;
110:  char * cp, cp2; // cp - указатель,
                      //но cp2 - переменная типа char
```

Получая указатель на переменную, можно обращаться к ней по адресу.

```
111:  int i = 1024, *ip = &i;
112:  int k, *kp = &k;
```

```
113:  *kp=i;
114:  k=*ip;     //1024 в k
115:  *kp = *ip; //1024 в k
116:  kp = ip;   //kp и ip будут указывать на одно место в
      памяти
```

С указателями можно выполнять следующие операции, результатом которых должно быть новое значение указателя: указатель + целое; указатель – целое; целое + указатель. В этих операциях целочисленное слагаемое указывает количество элементов, которое при вычислении смещения умножается на размер референта. Размер референта определяется по типу указателя.

```
117:  int a[] = {2,3,5};
118:  int *ip = a;
119:  cout << ip; //результат 0*00b0
120:  cout << ip - 1; //результат 0*00ae
121:  cout << ip + 1; //результат 0*00b2
```

При инициализации указателя типа char* часто используют текстовую строку. В этом случае указатель указывает на первый символ текстовой строки. Следует иметь в виду, что компилятор C++ автоматически добавляет к текстовой строке символ с нулевым кодом.

```
122:  char *st = "строка символов"; //в конец будет добавлен \0
123:  cout << st; //в поток выводится строка до символа \0
```

Ссылочный тип

Ссылкой задается синоним переменной. Адрес ссылки совпадает с адресом соотвествующей переменной.

```
124:  int val = 10;
125:  int &refVal = val; //refVal и val - синонимы
126:  RefVal = refVal + 2; // val == 12
```

Основное назначение ссылочного типа – передача параметров при вызове функций. Передача параметра по ссылке равносильна передаче адреса параметра.

Массивы

Массив – совокупность последовательно расположенных элементов одного типа. В C++ имеется только низкоуровневое понятие массива. Здесь массив не является особым типом данных. Для обращения к массиву используется адрес первого элемента массива.

```
127:  int ia[10]; //отведение места в памяти под массив
128:  const bufsize = 512;
129:  char buffer[bufsize]; //размерность массива -
                            //только константа
```

При объявлении массива можно сразу инициализировать его элементы. Количество инициализирующих значений должно быть не более коли-

чества элементов в массиве. Индексация элементов массива начинается от нуля.

```
130:  const arraysize=3;
131:  int ia[arraysize]={10,11,12};
                          //ia[0]==10
                          //ia[1]==11
                          //ia[2]==12
132:  int ib[]={10,11,12};
```

Фактически имя массива – это указатель, поэтому если ia[1] – это целое число, то ia – это указатель на целое число. С именем массива возможны любые операции, предусмотренные для указателей. Например, записи `*(buf+1)` и `buf[1]` – эквивалентны. Поэтому в C++ нет контроля выхода индекса массива за границы. Изменить значение имени массива нельзя, так как имя массива – это константа.

При инициализации массива символа следует помнить о добавлении символа \0 в конец строки символов.

```
133:  char ca[]={'A','B','C'};//массив из трех элементов
134:  char cb[]="ABC"; //массив из четырех элементов, так как
                       //добавляется символ \0
                       //cb[0]==*cb==A
135:  cb=cb+1; //ошибка трансляции - невозможно изменить
               //значение имени массива
```

Многомерные массивы объявляются и инициализируются аналогично одномерному массиву.

```
136:  int ia[3][4];  //3 строки, 4 столбца
137:  int ib[3][4]=  //внутренние скобки можно опустить
      {
        {0,1,2,3},
        {4,5,6,7},
        {8,9,10,11},
      };
138:  cout<<ib[0][0]; // 0;
139:  cout<<ib[0][2]; // 2;
140:  cout<<ib[1][2]; // 6;
```

Структура

Структура – это конструируемый тип данных, который позволяет агрегировать данные разного типа в одно целое. Формально объявление структуры записывается следующим образом: *`struct имя-типа {объявление типов}`*.

```
141:  struct Time {//объявление структуры
142:    int hour;
143:    int minute;
144:    int second;
145:  };
146:  main(){
147:    Time watch1, watch2; //объявление переменных
                             //типа структуры
```

```
148:    watch1.hour=12;
149:    watch1.minute=30;
150:    watch1.second=0;
151:    watch2=watch1; //побайтовое копирование
152:    cout << watch1.hour << endl;
153:    return 0;
154:  }
```

Синонимы типов

Синонимы типов используются для определения имен производных типов. Синонимы типов определяются с помощью оператора `typedef`.

```
155:  typedef double angle;
         //angle – новый вводимый тип
         //double – ранее определенный тип
156:  typedef char *String;
         //String - идентификатор для нового типа,
         //являющегося указателем на символьную строку
157:  typedef String Names[3];
         //Names - имя массива, в котором элементы
         //имеют тип String
158:  String congr = "Hello"; // congr – указаетель на строку
159:  Names person = {"Иванов", "Иван", "Иванович"};
         //person - массив из трех символьных строк
```

1.3. Динамическое распределение памяти

Оператор `new` обеспечивает выделение в динамически распределяемой памяти неименованной переменной заданного типа. В качестве параметра в операторе указывается тип создаваемой переменной. Оператор `new` возвращает указатель на выделенную переменную. В том случае, когда память не может быть выделена, оператор `new` возвращает 0. Освобождение памяти осуществляется оператором `delete`. В операторе `delete` задается указатель, содержащий адрес освобождаемой памяти.

```
160:  float *fptr; //объявление указателя
161:  fptr = new float; //создание неименованной вещественной
                        //переменной в куче
162:  delete fptr; //освобождение памяти
```

Оператором new возможно создание массива неименованных переменных в куче. В этом случае при освобождении памяти в операторе `delete` следует указать, что освобождается массив переменных.

```
163:  void f(){
164:  int *ia=new int[10];
165:  int ib[20];
166:  if(ia!=0)
167:    for(int i=0;i<10;i++)
168:      ia[i]=0;
169:  ...
170:  delete[]ia;
171:  }
```

Различие создания переменных еще раз поясним на примере выделения памяти под массив. В примере `ia` – это переменная, распределяемая в программном стеке. Ее значение может быть изменено. После выделения памяти под массив целых чисел в переменную `ia` был занесен адрес массива, созданного в куче. Массив с адресом `ib` создается в программном стеке. При этом `ib` – это константа типа указатель на целое число, то есть изменить `ib` невозможно. Массив из программного стека удаляется автоматически при выходе из функции в момент извлечения из стека адреса возврата.

В куче может быть выделена память под переменную любого типа, в том числе и под структуру. Для обращения к полям неименованной переменной типа структура, заданной указателем, в C++ предусмотрен специальный оператор "->".

```
172:  struct Time{ //объявление структуры
173:    int hour;
174:    int minute;
175:    int second;
176:  };
177:  void main(){
178:    Time t; //объявление переменной типа структуры,
                //она размещается в программном стеке
179:    Time *pt; //объявление указателя на структуру,
                  //самой переменной типа структура еще нет
180:    pt=new Time; //создание неименованной структуры в куче
                   //и занесение указателя на нее в переменную pt
181:    t.hour=12;
182:    (*pt).hour=11; //использование поля hour структуры
183:    pt->hour=10; //эквивалентная запись для доступа
                     //к полю структуры
184:    ...
185:    delete pt; //освобождение памяти от записи, значение
                   //переменной pt становится неопределенным
186:  }
```

1.4. Дополнительные операторы присваивания

Составной оператор присваивания

Составной оператор присваивания в общем виде может быть записан как `a op=b`, где `op` – арифметическая или логическая операция, `a` – изменяемый операнд, `b` – операнд, в котором указывается, на сколько изменяется операнд `a`. Составной оператор присваивания равносилен оператору присваивания `a=a op b`. Имеются следующие составные операторы присваивания: *=, /=, %=, +=, -=, >>=, <<=, &=, ^=, |=.

```
187:  sum+=1;       //эквивалентно sum=sum+1;
188:  product*=c;   //эквивалентно product= product*c;
```

```
189:  sum+=ia[i];  //эквивалентно sum= sum+ia[i];
```

Операторы инкремента и декремента

Операторы инкремента и декремента представляют собой унарные операции увеличения (++) и уменьшения (--) операнда на единицу. В префиксной форме данных операторов сначала выполняется действие, а потом используется полученное значение. В постфиксной форме – сначала используется значение, а потом выполняется действие над операндом.

```
190:  array[++top]=value; //префикская форма эквивалентна
                          //top=top+1; array[top]=value;
191:  value=array[top--]; //постфиксная форма эквивалентна
                          //value=array[top]; top=top-1;
```

1.5. Управляющие конструкции

Составной оператор

Чтобы задать несколько операторов там, где по синтаксису ожидается один, применяется составной оператор, также называемый блоком. Так как объявление переменных – это оператор, то в составном операторе могут быть объявлены локальные переменные. Они перестают существовать при завершении составного оператора.

```
192:  if(x>1)
193:  {           //начало составного оператора
194:    float y;  //объявление локальной переменной
195:    y=2*x;
196:    ...
197:  }
198:  x=y; //ошибка - переменная y вне блока не существует
```

Ветвление

Оператор ветвления имеет две формы: сокращенную и полную.

```
199:  if(выражение) оператор1
200:  if(выражение) оператор1 else оператор2
```

В сокращенной форме `оператор1` выполняется, если `выражение` не равно нулю. В полной форме `оператор1` выполняется, если `выражение` не равно нулю, в противном случае выполняется `оператор2`.

```
201:  if(counter==0)
202:    counter=10;
203:  else
204:    counter--;
205:  if(counter) //выражение должно быть целым
206:    counter=10;
```

Оператор выбора

Оператор выбора switch предназначен для выполнения одного из набора взаимоисключающих действий. В операторе `switch` используется оператор `break`, который прерывает выполнение оператора выбора и выполняет переход к оператору, следующему за оператором выбора.

```
207:  main (){
208:    int n;
209:    int cnt0,cnt1,cnt2; //объявление переменных
210:    ...
211:    switch(n) { //выражение целого типа
212:    case 0:   //0 – константа
          ++cnt0; break;
213:    case 1:   //1 - константа
          ++cnt1; break;
214:    case 2:
          ++cnt2; break;
215:    default:  //все остальные случаи
          ++cnt; break;
216:    }; //конец оператора switch
217:  return 0;
218:  }
```

Самая распространенная ошибка при использовании оператора `switch` – пропуск оператора `break`. Если необходимо выполнить одни и те же действия при разных значениях выражения выбора, то константы выбора должны следовать друг за другом.

```
219:  switch(n){
220:  case 0:
        ++cnt0; break;
221:  case 1:
        ++cnt1; break;
222:  case 2: //при отсутствии break будут выполняться
               //следующие case
223:  case 3:
224:  case 4:
        ++cnt; break;
225:  };
```

Операторы цикла

Оператор цикла с предусловием имеет следующую общую форму записи: `while(выражение) оператор`. `Оператор` выполняется до тех пор, пока `выражение` не равно нулю.

```
226:  //Обнуление элементов массива
227:  float a[10];
228:  int i=0;
229:  while(i<10){
230:    a[i]=0;
231:    i++;
232:  }
233:  //Определение длины символьной строки
```

```
234:  int len=0;
235:  char *tp="abcde";
236:  while(*tp++)++len;
237:  cout<<len;
```

Оператор цикла с предусловием имеет следующую общую форму записи: `do оператор while (выражение)`. Выход из цикла выполняется, если `выражение` равно нулю.

```
238:  int cnt=10;
239:  do{
240:    --cnt;
241:    a[cnt]=0;
242:  }while (cnt>0);
```

Итеративный цикл имеет следующую общую форму записи:

```
243:  for (инициализация; выражение1; выражение2) оператор
```

`Инициализация` предназначена для выполнения некоторых действий перед входом в цикл, как правило это инициализация счетчика цикла. `Инициализация` может содержать несколько операторов, разделенных запятой. `Выражение1` управляет последовательностью выполнения цикла, `выражение2` изменяет переменную цикла. Последовательность действий при выполнении оператора цикла: сначала выполняется `инициализиация оператора`, далее, если `выражение1` не ноль, то выполняется оператор, а затем вычисляется `выражение2`. Если `выражение1` равно нулю, то цикл завершается, в противном случае снова выполняется `оператор`.

```
244:  //обработка элементов массива
245:  const int sz=24;
246:  int ia[sz];
247:  for(int i=0;i<sz;++i)
248:    ia[i]=i;
249:  //вычисление факториала
250:  int k;
251:  for(k=10,product=1; k>0; k--)
252:    product=k*product;
```

Операторы безусловного перехода

Оператор `break` прекращает выполнение ближайшего вложенного оператора `while, do, for, switch`.

```
253:  location=-1;
254:  for(int ix=0;ix<size;++ix)
255:    if(val==ia[ix]){
256:      location=ix;
257:      break; //выход из цикла
258:    }
```

Оператор `continue` прекращает выполнение только текущей итерации и передает управление на конец тела цикла.

```
259:  while{
260:    if(sum==0) continue; //обход последующих операторов
261:    ++counter;
```

```
262:  };
```

Оператор безусловного перехода имеет вид `goto метка`. `Метка` – это идентификатор. Оператор `goto` не должен выполнять переход вперед через объявление переменных.

```
263:  while{
264:    if(sum==0) goto m; //обход последующих операторов
265:    ++counter;
266:  };
267:  m: counter=0;
```

1.6. Функции

В объявлении функции указывается тип возвращаемого значения, идентификатор фукнции, список формальных аргументов и тело функции.

```
268:  тип идентификатор([список аргументов]) {тело функции}
```

Выход из функции и возврат значения выполняются оператором `return`. Если предполагается, что функция не должна ничего возвращать, то в ней указывается тип возвращаемого значения `void`. Тип возвращаемого значения может быть любым стандартным типом (например, `int` или `double`), производным типом (например, `int&` или `double*`), пользовательским типом (полученным, например, с помощью `enum` или `struct`).

Параметры передаются либо по значению (в том числе можно передавать значение адреса переменной), либо по ссылке.

```
269:  #include <iostream.h>
270:  double max(double x1, double x2){
                        //передача параметров по значению
271:    if(x1>x2)
272:      return x1; //выход из функции и возврат значения
273:    else
274:      return x2; //выход из функции и возврат значения
275:  }
```

При вызове функции указывается ее идентификатор и фактические параметры. Функция, возвращающая значение, обычно вызывается в выражении.

```
276:  int main(){
277:    double a1=3;a2=2;a3;
278:    a3=max(a1,a2); //вызов функции,
                       //a1 и a2 - фактические параметры
279:    cout<<a3;
280:    return 0;
281:  }
```

Если функция не возвращает значения, то используется оператор `return` без параметров. В этом случае оператор `return` может быть опущен. Передача адреса переменной позволяет изменять значение внешних переменных внутри функции.

```
282:  void swap(double *a, double *b){ //передача параметров
            //по значению, но значениями являются адреса a и b
```

```
283:    double c;
284:    c=*a;
285:    *a=b;
286:    *b=c;
287:  }
288:  main(){
289:    double x=5,y=1;
290:    swap(&x,&y);
291:    return 0;
292:  }
```

Передача параметра по ссылке реализуется как передача адреса параметра. Так как ссылка, по определению, есть синоним переменной, то при вызове функции формальный параметр становится синонимом фактического параметра. Поэтому изменения параметра, переданного по ссылке, вызовут изменения внешнего фактического параметра.

```
293:  void swap(double &a, double &b){
294:    double c;
295:    c=a;
296:    a=b;
297:    b=c;
298:  }
299:  main() {
300:    double x=5, y=1;
301:    swap(x,y); //в качестве фактических параметров
                   //передаются синонимы
302:    return 0;
303:  }
```

Так как в C++ отсутствует высокоуровневое понятие массива, то передача массива в качестве параметра сводится к передаче указателя на начальный элемент массива. Следствием этого является необходимость передачи в качестве параметра функции размера массива.

```
304:  double sum(int n, double *a) {
305:    double s=0.0;
306:    for(int i=0;i<n;i++)
307:      s=s+a[i];
308:    retutn s;
309:  }
310:  int main(){
311:    double arr[3]={3,2,1};//arr - указатель на массив
312:    cout<<sum(3,arr);
313:    return 0;
314:  }
```

Для передачи адреса массива возможны следующие эквивалентные формы записи:

```
315:  double sum(int n, double a[10]){...}
316:  double sum(int n, double a[]){...}
317:  double sum(int n, double *a){...}
```

Прототип функции

Прототип функции – это заголовок со списком типов параметров, объявляемый до полного определения функции (предварительное объявление функции). В объектно-ориентированном программировании прототип функции используется для предварительного объявления методов класса.

Пример 1. Нахождение куба числа (передаётся указатель на целое число, и это число заменяется своим кубом).

```
318:  #include <iostream.h>
319:  //Нахождение куба числа. Число заменяется своим кубом
320:  void cubeByReference(int *);//прототип функции
321:  main(){
322:    int number=5;
323:    cubeByReference(&number);
324:    cout<<"В кубе="<<number;
325:    return 0;
326:  }
327:  void CubeByReference(int *nptr) { //объявление функции
328:    *nptr=*nptr * *nptr * *nptr;
329:  }
```

Перегрузка функций

В C++ допускается объявление функций с одинаковыми именами, но различными списками формальных параметров. При трансляции выполняется так называемое декорирование (искажение) имен. Оно заключается в том, что транслятор искажает имена, формируя сигнатуры функций. *Сигнатура функции* – это внутреннее имя функции, которое образуется добавлением к заданному имени функции обозначений типов параметров. Например, для функций `int square(int)`, `double square(double)`, `Complex& square(Complex)` будут сформированы сигнатуры приблизительно следующего вида: `square$i`, `square$d`, `square$Complex`. Конкретный способ искажения имен зависит от версии транслятора. В сигнатуру не включается тип возвращаемого значения, так как в контексте вызова функции невозможно однозначно определить необходимый тип возвращаемого значения как, например, в выражении `cout<<square(2)`.

```
330:  struct Complex{double Re; double Im};
331:  int square (int x){return x*x;}
          //функция для целых чисел
332:  double square (double x) {return x*x;}
          //функция для вещественных чисел
333:  Complex square(Complex x){
          //функция для комплексных чисел
334:    Complex w;
335:    w.Im=2*x.Im*x.Re;
336:    w.Re=x.Im*x.Im-x.Re*x.Re;
```

```
337:    return w;
338:  }
339:  main(){
340:    double z=3.1
341:    cout<<square(z);
342:    return 0;
343:  }
```

Шаблоны функций

Шаблоны функций позволяют создавать описания функции без указания фактических типов параметров и по мере необходимости генерировать из этих описаний функции с конкретными типами параметров.

```
344:  temlate<список формальных типов (параметров)> функция
```

Каждый раз, когда транслятор встречает в исходном тексте программы вызов функции, имеющей имя, совпадающее с именем шаблона, он транслирует шаблон функции, подставляя вместо формальных типов параметров фактические типы. Формальные типы могут заменяться любыми фактическими, в том числе пользовательскими типами.

```
345:  //шаблон функции определения максимального
      //из трех чисел
346:  temlate<class T>
347:  T maximum(T value1, T value2, T value3){
348:    T max=value1;
349:    if(value2>max)
350:      Max=value2;
351:    if(value3>max)
352:      Max=value3;
353:  return max;
354:  }
355:  main(){
356:  int i1, i2, i3, ir;
357:  double d1=3.14, d2=0.15, d3=8.1, dr;
358:  ir=maximum(i1, i2, i3); //целые параметры
359:  dr=maximum(d1, d2, d3); //вещественные параметры
360:  ...
361:  return 0;
362:  }
```

Реализация шаблонов функций базируется на механизме перегрузки функций. При создании параметризованных функций генерируются функции с одинаковыми именами, после чего эти имена подвергаются декорированию. По этой причине нельзя задавать шаблон, в котором формальный тип объявлен только для возвращаемого значения, а не для параметров, так как такие функции были бы неразличимы.

Заголовочные файлы и файлы реализации

В C++ отстуствует поддержка модульности программ на уровне языка программирования. Модульность программного материала формируется

на уровне соглашений. Принято, чтобы прототипы функций располагались в файлах с расширением .h, а реализации функций – в файлах с расширением .cpp.

```
//файл MyMath.h
int abs(int i);

//файл MyMath.cpp
int abs(int i)
{return (I<0 ? - i:i)

//файл main.cpp
#include <iostream.h>
#include <MyMath.h>
main()
{
   cout<<abs(2*(-3));
}
```

Рекурсия

Передача параметров по значению при вызове функции предполагает, что копия параметра помещается в программный стек. Кроме этого, все локальные переменные функции размещаются в стеке при каждом входе в процедуру. Это позволяет реализовать рекурсивный вызов функций, то есть вызов функции в своем же теле.

```
363:  int fact(int k){
364:    if(k=1)
365:      return 1;
366:    else
367:      return k*fact(k-1); //рекурсивный вызов
368:  }
369:  main(){
370:     cout<<fact(3);
371:  }
```

Глава 2. **ОБЪЕКТНАЯ МОДЕЛЬ ПРОГРАММЫ**

Рассмотрим определения понятия объект с разных точек зрения:

1) с точки зрения *человеческого восприятия*, объект – это осязаемый или видимый предмет, имеющий определенное поведение;

2) с точки зрения *исследователя* в некоторой предметной области объект – это опознаваемый предмет или сущность (реальная или абстрактная), имеющие важное функциональное значение в данной предметной области;

3) с точки зрения *проектирования* системы, объект – это сущность, обладающая состоянием, поведением и индивидуальностью.

4) с точки зрения *программирования*, объект – это совокупность данных (переменная), существующая в машинном представлении как единое целое, допускающее обращение по имени или указателю.

Далее будем рассматривать объект с точки зрения проектирования системы. Группы объектов могут иметь схожие черты, которые составляют понятие класса. *Класс* – это *описание* структуры и поведения схожих объектов. Будем использовать понятия *"экземпляр класса"* и *"объект"* как синонимы.

2.1. Объявление классов и создание объектов в программе на C++

Тип класс задаёт модель атрибутов (поля структуры) и операций (функции, связанные со структурой). В классе можно объявлять или реализации методов, или прототипы методов с внешним определением их реализации.

```
372: class Time{
373: private:
374:    int hour;
375:    int minute;
376:    int second; //объявление полей
377: public:
378:    Time(); //конструктор
379:    void setTime(int, int, int);
380:    void print24();
381:    void print12();
382: }
```

Зарезервированные слова `private, public` – это модификаторы доступа, они обеспечивают управление доступа к элементам класса. Указание модификатора `private` запрещает доступ к полям и методам извне класса, `public` – позволяет доступ извне класса. Реализация метода, записанная вне объявления класса, должна содержать название класса.

```
383: Time::Time(){ //инициализирует поля данного класса
384:    hour=0;
385:    minute=0;
386:    second=0;
387: }
388: void Time::setTime(int h, int m, int s) {
389:    hour=h;
390:    minute=m;
391:    second=s;
392: }
393: void Time::print24(){
394:    cout<<hour<<":"<<minute<<":"<<second;
395: }
396: void Time::print12(){
397:    if(hour==0||hour==12)
398:    cout<<12;
399: else
```

```
400:    cout<<hour%12;
401: cout<<":"<<minute<<":"<<second;
402: }
```

Объект класса – это переменная данного класса. При выполнении оператора объявления переменной вызывается конструктор класса.

```
403: //головная функция
     int main(){
404:    Time t; //вызов конструктора класса Time
405:    t.setTime(14,32,11);
406:    t.print24();
407:    t.print12();
408:    Time *tptr;
409:    tptr=new Time; //динамическое выделение памяти под
                       //объект и вызов конструктора
410:    (*tptr).setTime(1, 2, 3);
411:    delete tptr; //удаление объекта, на который
                     //указывает tptr
412:    return 0;;
413: }
```

Если в классе не объявлен конструктор, то он создаётся автоматически. Если в классе имеется несколько конструкторов, то используется механизм перегрузки функций, то есть конструкторы должны различаться списками параметров.

2.2. Состояние

Состояние объекта – перечень всех возможных свойств (обычно статических) объекта и текущих значений (обычно динамических) каждого из этих свойств. Перечень свойств объекта (абстракции сущности) – как правило статический, так как эти свойства отражают неизменяемую основу природы объекта. Свойства объекта характеризуются своими значениями, которые могут быть либо простыми количественными значениями, либо представляют собой состояния других объектов.

Например, технологический производственный процесс, как объект, имеет простые количественные характеристики: количество выпускаемой продукции в единицу времени, количество работников и т.д. В то же время он характеризуется состоянием таких объектов, как технологические агрегаты, транспортные средства и так далее, то есть тех объектов, которые могут существовать независимо от технологического процесса и над ними могут совершаться определенные действия.

2.3. Поведение

Поведение характеризует то, как объект воздействует или подвергается воздействию других объектов с точки зрения изменения состояния этих

объектов и передачи сообщений. *Операция* (действие, сообщение, метод, функция) – это определенное воздействие одного объекта на другой с целью вызвать соответствующую реакцию.

Категории операций над объектами:

1) *функция управления* – операция, которая изменяет состояние объекта;

2) *функция доступа* – операция, дающая доступ для определения состояния объекта без его изменения (операция чтения);

3) *функция реализации* – операция над информацией из внешних источников, не изменяющая состояния данного объекта;

4) *вспомогательная функция* – операция, используемая вышеперечисленными операциями, и не предназначенная для самостоятельного использования другими объектами;

5) *конструктор* – операция создания и инициализации объекта;

6) *деструктор* – операция разрушения объекта и освобождения занимаемой им памяти.

Совокупность всех методов, относящихся к конкретному объекту, образует протокол объекта. Протокол определяет, охватывает (инкапсулирует) внутреннее состояние объекта. Наличие внутреннего состояния означает, что порядок изменения состояния определяется и последовательностью операций над объектом, и предшествующим состоянием. Под термином *инкапсуляция* подразумевают указанное сокрытие внутренней структуры данных и реализации методов объекта от остальной программы.

Рассмотрим пример, в котором конструируется класс "Стек". Методы класса реализуют следующие функции: `push` – добавление элемента в стек; `pop` – возврат значения элемента из вершины и удаление его; `isEmpty` – возвращает 1, если стек пуст; `isFull` – возвращает 1, если стек заполнен; `copy` – создает копию другого стека.

```
414: class Stack{
415: private:
416:    int * s;  //указатель на первый элемент массива
417:    int top;  //указатель на вершину стека
418:    int size; //размер стека
419: public:
420:    void push(int i);     //управление
421:    int pop();            //управление
422:    int isEmpty();        //доступ
423:    int isFull();         //доступ
424:    void copy(Stack *st); //реализация
425:    Stack(int sz);        //конструктор
426:    ~Stack();             //деструктор
427: };
```

В C++ имя конструктора совпадает с именем класса. Можно объявить несколько конструкторов, отличающихся списками параметров. Деструктор объявляется как имя класса с символом "~". Он всегда один и не имеет параметров.

Основное назначение конструктора – выделить под объект память, объём которой равен суммарному объёму полей объекта. Второе, необязательное, назначение – выполнить некоторые действия, например по инициализации полей. Если ни один конструктор не объявлен в классе, то компилятор сам генерирует код конструктора без параметров, который может выполнять только основное своё назначение – выделение памяти под объект.

Основное назначение деструктора – освобождение памяти от занимающих её полей объекта. Второе, необязательное, назначение – выполнение некоторых действий по освобождению ресурсов. Если деструктор не объявлен в классе, то компилятор сам генерирует код деструктора, который может только освобождать память.

Существует два способа создания объектов: объявление локальных переменных и распределение объектов в динамически распределяемой памяти с помощью оператора `new`. Локальные объекты создаются в программном стеке и удаляются автоматически при завершении функции, то есть в момент извлечения из программного стека адреса возврата. Динамически распределенные объекты должны удаляться явно при помощи оператора `delete`, в противном случае неудаленные объекты будут оставаться в памяти как "мусор".

Рассмотрим продолжение предыдущего примера со стеком. Ниже приводится реализация методов класса и головная функция, создающая локальный объект и динамически распределенный объект.

```
428: Stack::Stack (int sz) {
429: size=sz;
430: s=new int[size];//массив элементов(s - указатель на него)
431: top=-1;        //несуществующий индекс массива
432: };
433: Stack::~Stack(){
434: delete []s;
435: };
436: int Stack::pop(){
437: int last=s[top]; //значение на вершине стека
438: top=top-1;
439: return last;//возвращение значения выталкиваемого
                 //элемента
440: };
441: int Stack::isEmpty(){
442:   if (top>=0) return 0;
```

```
443:   else return 1;
444: };
445: int Stack::isFull(){
446:   if (top>=size-1) return 1;
447:   else return 0;
448: };
449: void Stack::Copy (Stack *st){
450:   while(!st->isEmpty())
451:     st->pop();
452:   if(int I; i<=top; i++)
453:     st->push(s[i]);
454: }
455: void Stack::push(int i){
456:   if(!isFull()){
457:     top=top+1;
458:     s[top]=I;
459:   }
460: void main(){
461:   Stack st1(10); //создание локального объекта st1
462:   Stack *pst2;
463:   pst2=new Stack(10); //создание объекта в куче
464:   st1.push(77);
465:   pst2->push(33);
466:   pst2->push(st1.pop())//изменение состояния стека
467:   pst2->copy(&st1);
468:   delete pst2; //вызов деструктора Stack::~Stack()
469: } //здесь вызывается деструктор для локального объекта
```

2.4. Индивидуальность

Индивидуальность – это совокупность тех свойств объекта, которые отличают его от всех других объектов. Структурная неопределённость – это нарушение индивидуальности объектов.

```
470: void main(){
471:   Stack st1(10), st2(10);
472:   st1=st2;
473: }//ошибка освобождения памяти при выполнении
      //деструктора второго объекта
```

Для создания тождественных объектов используются *копирующие конструкторы*. Копирующий конструктор – это конструктор, в списке формальных параметров которого стоит ссылка на объект того же класса.

```
474: class Stack{
475:   int size; int top; int *s; int sz;
476:   Stack(){
477:     size=sz; top=-1;
478:     s=new int [size];
479:   }
480: Stack(const Stack &st){//ссылка на объект того же класса
481:   size=st.size;
482:   top=st.top;
483:   for(int i=0; i<=top; i++) //копирование массива
484:      s[i]=st.s[i];
485: }
486: void f(Stack st){
```

```
    //при вызове функции создается таблица адресов и
    //значений. В нее помещается копия фактического
    //параметра - объекта класса Stack
...
}
int main(){
  Stack st1(100);
  ...
  Stack st2(st1); //вызов конструктора копирования
  f(st1);
}
```

2.5. Отношения между объектами

Отношение определяет способы взаимодействия объектов. При взаимодействии один объект "знает" операции, которые можно выполнять над другим объектом. Выделяют два типа отношений объектов: отношение использования и отношение включения (отношение агрегатирования).

Отношение использования возникает в том случае, когда один объект посылает сообщение другому объекту. При этом взаимодействующие объекты создаются и уничтожаются независимо друг от друга. В головной функции примера объект класса Contorller использует объект класса Sensor и объект класса Valve.

```
class Sensor{
public:
  int getSignal();
};
class Valve{
public:
  void switchOn(int Delta);
  void switchOff(int Delta);
};
class Controller{
public:
  void run(Sensor &s, Valve &v) {
  if(s.getSignal()>10)
     v.switchOn(5);
  else
     v.switchOff(5);
}
void main(){
  Sensor sensor;
  Valve value;
  Controller controller;
  ...
  controller.run(sensor, value);
}
```

Агрегатирование – это отношение объектов, при котором объект класса более высокого уровня использует в своем составе объекты других классов (в качестве составных частей). В этом случае время существова-

ния агрегирующего объекта определяет время существования включаемых объектов: при создании объекта-агрегата должны создаваться автоматически и включаемые объекты, при удалении объекта-агрегата предварительно должны быть удалены включаемые объекты.

```
519: class Mouth{
520:   float capacity;
521:   Mouth(float l);
522:   ~Mouth();
523: };
524: class Eye{
525:   float radius;
526:   int color;
527:   Eye(float r, int c);
528:   ~Eye();
529: };
530: Mouth::Mouth(float c){
531:   capacity=c;
532: }
533: Mouth::~Mouth(){;}
534: Eye::Eye(float r, int c){
535:   radius=r;
536:   color=c;
537: }
538: Eye::Eye(){;}
539: class Face{
540:   Eye leftEye;
541:   Eye rightEye;
542:   Mouth mouth;
543:   Face (float eRadius, int eColor, float mCapacity);
544:   ~Face(){;}
545: }
546: Face::Face(float eRadius, int eColor, float
     mCapacity)::leftEye(eRadius, eColor), rightEye(eRadius,
     eColor), mouth(mCapacity) {
547:   ...
548:   leftEye=rightEye;
549:   ...
550: }
551: Face::~Face{;}
552: int main(){
553:   Face face(5, 1, 10); //здесь вызываются конструктор
                           //агрегирующего объекта и конструкторы
                           //агрегируемых объектов
554: } //здесь автоматически вызываются деструкторы
      //агрегируемых объектов и деструктор
      //агрегируемого объекта
```

Глава 3. **ОТНОШЕНИЯ МЕЖДУ КЛАССАМИ**

Класс – это описание структуры и поведения объектов, связанных отношением общности. Между классами существует три типа отношений: наследование, использование, наполнение (обобщение).

3.1. Отношение наследования

Наследование – это такое отношение между классами, при котором один класс повторяет структуру и поведение другого класса (простое наследование) или нескольких других классов (множественное наследование). Отношение наследования задаёт на множестве классов иерархию по номенклатуре, в которой определяются пары суперкласс – подкласс. В подклассе структура соответствующего суперкласса дополняется новыми полями, а поведение может дополняться новыми методами, существующие методы могут переопределяться и уточняться.

В C++ наследование обозначается следующим образом.

```
555:  class C: public A{...} // простое наследование
556:  class C: public A, public B{...}
          // множественное наследование
```

При проектировании не для всех классов создаются объекты. Некоторые классы предназначены только для указания общих свойств своих подклассов на определённом уровне иерархии. Потом эти свойства реализуются в конкретных подклассах. Классы, для которых не определены реализации объектов, называются абстрактными классами. Те конкретные методы, которые не реализуются в абстрактном классе, называются абстрактными методами. В C++ они объявляются со спецификатором "=0".

```
557: class Figure{
558: private:
559:   Point center;
560:    ...
561: public:
562:   virtual void draw()=0; // реализация отсутствует
563: ...
564: };
565: class Circle:public Figure{
566: private:
567:   int radius;
568: public:
569:   void draw(){ //реализация абстрактного метода
                    //в подклассе
570:     ...
571:   }
572: };
```

В зависимости от модификатора доступа, указанного в наследовании, уровень доступа полей и методов может быть изменен.

Доступ в суперклассе	Модификатор доступа при наследовании	Уровень доступа в подклассе
Открытый	public	Открытый
Закрытый	public	Закрытый
Защищённый	public	Защищённый
Открытый	private	Закрытый
Закрытый	private	Недоступный
Защищённый	private	Закрытый
Открытый	protected	Защищённый
Закрытый	protected	Недоступный
Защищённый	protected	Защищённый

Пример добавления метода.

```
573: class Circle{
574: public:
575:   void show();
576: }
577: class Eye:public Circle{
578: public:
579:   void close();
580: ...
581: }
582: int main(){
583:   Circle c;
584:   Eye e;
585:   e.show();
586:   c.show();
587:   e.close();
588:   c.close(); //ошибка, этот метод имеется
                 //только в подклассе
589:   ...
590: }
```

Переопределение методов

Переопределение (замещение) – объявление в подклассе метода с таким же именем и списком параметров, как и в суперклассе. При этом новая реализация метода в подклассе полностью заменяет реализацию метода в суперклассе.

Переопределение может быть невиртуальным либо виртуальным. Способ переопределения метода оказывает влияние на то, какой именно метод будет вызываться для объекта подкласса при выполнении программы. Поэтому говорят о раннем связывании и позднем связывании объекта и выполняемого метода.

Невиртуальное переопределение метода приводит к раннему связыванию объекта и метода, то есть к связыванию на этапе трансляции. Вызываемый метод для указателя на объект определяется по типу указателя, а

не по типу фактического объекта. Это возможно делать на этапе компиляции.

```
591: class Circle{
592: public:
593:   void show();
594:    ...
595: }
596: class Eye: public Circle{
597:   void show();
598:   ...
599: }
600: int main(){
601:   Circle *pC; //указатели на объекты суперкласса
602:   pC=new Circle; //фактический объект - класса Circle
603:   pC->show(); //вызов метода Circle::show()
604:   delete pC;
605:   pC=new Eye(); //фактический объект - класса Eye
606:   pC->show(); //вызов метода Circle::show()
607:   delete pC;
608:   return 0;
609: }
```

Виртуальное переопределение приводит к тому, что метод, вызываемый для объекта, на который указывает указатель суперкласса, определяется во время выполнения программы по фактическому типу объекта. Поэтому в данном случае говорят о позднем связывании объекта и метода.

```
610: class Figure{
611:   int x,y;
612:   virtual void show();
613:   virtual void hide();
614:   virtual void move(int newX, int newY); {
615:     hide();
616:     x=newX; y=newY;
617:     show();
618: };
     //в классах потомках не переопределяется метод move
619: class Circle:public Figure{
620:   int radius;
621: public:
622:   virtual void show();
623:   virtual void hide();
624: };
625: class Face: public Circle{
626:   int eyeColor;
627: public:
628:   void show();
629:   void hide();
630:   virtual void closeEyes(); //добавленный метод
631: };
632: int main(){
633:   circle *cPtr;
634:   cPtr=new Face; //фактический объект класса Face
635:   cPtr->show(); //вызывается метод Face::show()
636:   cPtr->move(10,20);
        //Вызывается Figure::move(), так как он наследуется.
```

```
        //А в нем вызываются методы Face::show() и
        //Face::hide() в соответствии с фактическим
        //классом объекта *cPtr.
637:   cPtr->closeEyes();
638:   delete cPtr;
639:   ...
640: }
```

Таблица виртуальных методов

Позднее связывание может быть реализовано различными способами. Наиболее распространенный – это реализация при помощи таблицы виртуальных методов (VMT – virtual method table).

На этапе трансляции транслятор строит VMT для всех классов. Количество элементов в VMT равно количеству виртуальных методов в классе, включая наследуемые виртуальные методы. Невиртуальные методы в VMT не заносятся. Таблицы виртуальных методов – это глобальные константы, представляющие собой массивы адресов методов.

Установление связи между объектом и соответствующими его фактическому классу методами осуществляется на этапе выполнения в момент создания объекта в памяти. В создаваемом объекте выделяется дополнительное поле для указателя на VMT. Конструктор заносит в это поле адрес VMT соотвествующего класса. Вызов виртуального метода, например,

```
641: cPtr->move(10,20);
```

преобразуется в косвенный вызов функции:

```
642: (*(cPtr->v_ptr[2]))(cPtr, 10, 20);
```

Как видно, при вызове метода в список параметров всегда добавляется необъявленный параметр – указатель на объект, для которого выполняется метод. Этот указатель необходим для доступа к полям объекта.

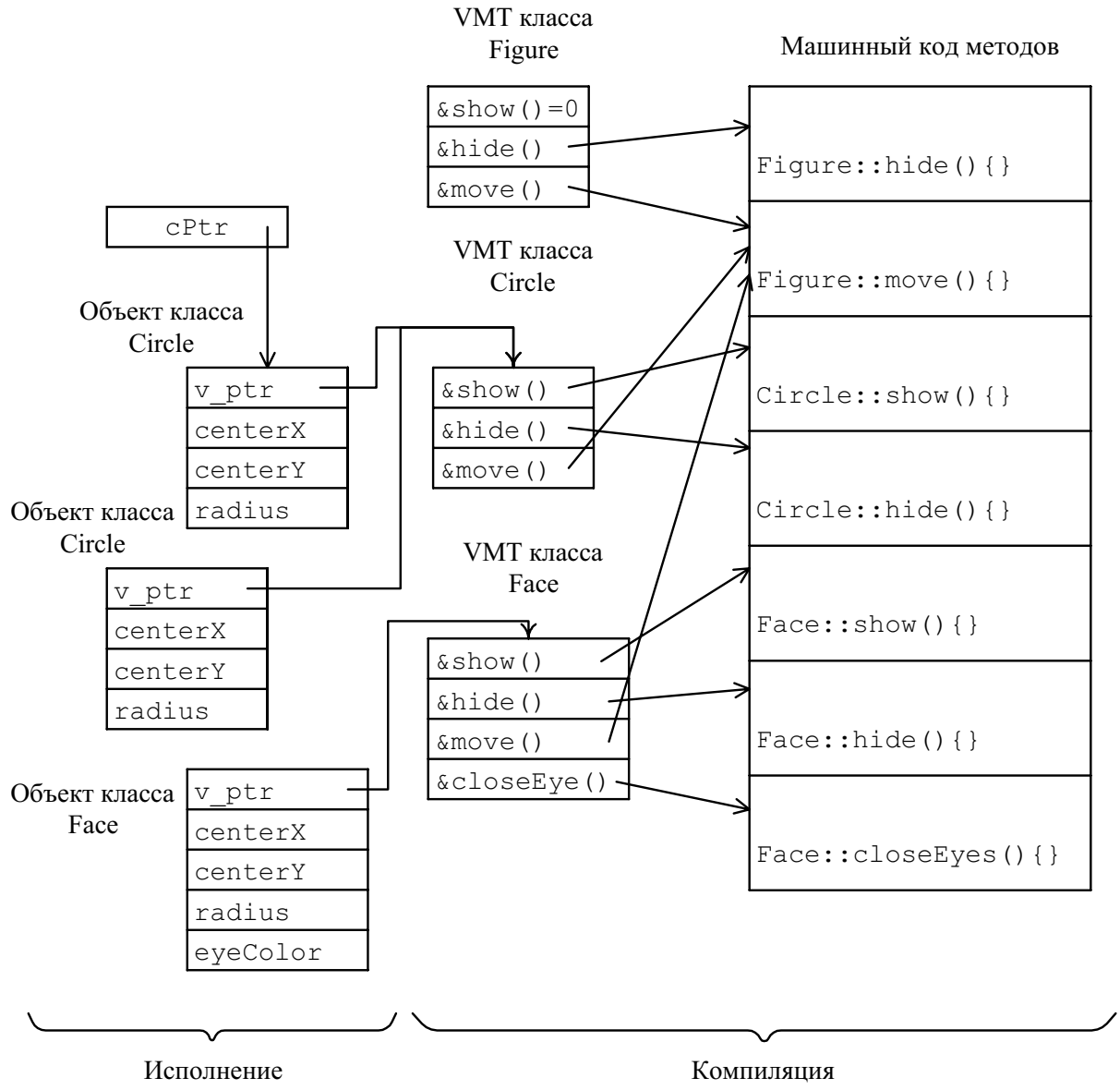

Рис. 1. Формирование таблиц виртуальных методов

Особенности работы с конструкторами и деструкторами

Конструктор играет особую роль, он никогда не является виртуальным. Конструктор не возвращает значение. В классе может быть объявлено несколько конструкторов, тогда реализуется перегрузка конструкторов. Конструкторы не наследуются. При создании объектов сначала выполняются конструкторы суперкласса, начиная с базового класса. Каждый конструктор обрабатывает выделенную под объект память как объект "своего класса" (например инициализирует поля объекта). По этой причине использование виртуальных методов в конструкторе может приводить к ошибкам. А вот невиртуальные методы можно использовать без ограничений.

```
643:  class A {...}
644:  class B:public A{...}
645:  class C:public B{...}
646:  main(){
647:  cPtr=new C; //здесь вызываются конструкторы всех предков
                  //A::A(); B::B(); C::C()
648:  ...
649:  }
```

Деструктор выполняет действия по уничтожению объекта. Деструкторы суперклассов выполняются в порядке, обратном выполнению конструкторов. Сначала выполняются команды тела деструктора, потом разрушаются поля, добавленные к родительскому классу подклассом. После этого выполняются деструкторы родительских классов вплоть до базового. Использование виртуальных методов в деструкторе родительского класса недопустимо, так как поля подкласса к этому моменту уже разрушены.

Деструкторы не наследуются. Если в объявление класса не добавлен деструктор, то он генерируется компилятором. Деструкторы могут быть объявлены виртуальными.

```
650: class B{
651: public:
652:   B();
653:   ~B();
654: };
655: class D:public B{
656: public:
657:   D();
658:   ~D();
659: }
660: void f(){
661: B *p=new D();  //создается объект класса D
662: delete p;      //уничтожается объект класса B
663: }
```

В данном случае надо объявить виртуальный деструктор.

```
664: class B{
665: B();
666: virtual ~B();
667: };
```

Уточнение методов

Уточнение метода при наследовании осуществляется в том случае, когда метод подкласса добавляется к существующему методу суперкласса, но при этом полностью использует код метода суперкласса. В этом случае код методов суперклассов используется повторно, а не копируется от метода к методу.

```
668: class Figure{
669: public:
670:   virtual void show(){...}
671: };
```

```
672: class Circle:public Figure{
673: public:
674:   virtual void show(){
675:     Figure::show(); //вызов метода суперкласса
676:     ... //рисование окружности
677:   }
678: }
679: class Face:public Circle{
680: public
681:   virtual void show(){
682:     Circle::show(); //вызов метода суперкласса
683:     ... //рисование других элементов
684:   }
685: }
686: void main(){
687:   Face F;
688:   F.show();    // будут вызваны три метода:
                   // Figure::show();
                   // Circle::show();
                   // Face::show();
689: }
```

В C++ автоматическое уточнение методов реализовано для конструкторов и деструкторов.

Формы наследования

Порождение подклассов для специализации (порождение подтипов). В этом случае полностью выполняется принцип подстановки, то есть новый класс является специализированной формой родительского класса и поддерживает его протокол.

Порождение классов для спецификации. Родительский класс – абстрактный класс. Подклассы должны обеспечивать реализацию поведения, заложенную в суперклассе. Таким образом абстрактный суперкласс задает (но не реализует) спецификации подклассов.

Порождение с целью конструирования. Подкласс наследует поведение и структуру родительского класса, но отсутствует необходимость в установлении отношения "это есть" (отношения супертип-подтип). Например, конструирование абстрактных типов данных на базе массива или списка.

```
690: #include<iostream.h>
691: class DynamicArray{
692: private:
693:   int *h;
694:   int size;
695: public:
696:   DynamicArray(int sz){
697:     size=sz;
698:     h=new int[sz]; //выделение памяти
699:   };
```

```
700:   void sort(){;}  //сортировка
701:   int getSize(){return size;}
702:   DynamicArray(){delete [] h;}
703: };
704: class Stack:private DynamicArray { //закрытое наследование
705: private:
706:   int top;
707: public:
708:   Stack(int sz):DynamicArray(sz),top(-1){;}
709:   int isFull(){
710:     if(top>=getSize()-1)) return 1;
711:     else return 0;
712:   }
713: };
714: int main(){
715:   Stack st(10);
716:   cout<<st.isFull()<<endl;
717:   cout<<st.getSize()<<endl;
                //ошибка из-за закрытого наследования
718:   st.sort();
                //ошибка из-за закрытого наследования
719:   return 0;
720: }
```

Порождение класса для обобщения. Метод противоположный специализации. Предок является более общим классом. Например, класс Windows позволяет отображать белые символы на черном фоне, а класс ColouredWindows, являющийся потомком Windows, позволяет отображать цветные символы на цветном фоне. Необходимость в этой форме может возникнуть при проектировании новой иерархии классов на базе существующей иерархии классов. Обобщения следует избегать.

Порождения класса для расширения. Добавление совершенно новых свойств, при этом не переопределяется ни один метод предка.

Порождение класса для ограничения. Возможности подкласса более ограничены, чем в родительском классе. Например, пусть в библиотеке классов имеется класс Deque (double ended queue) – очередь с двумя концами. На базе этого класса требуется создать класс стек. Тогда следует модифицировать методы выполнения действий над одним концом очереди так, чтобы они выдавали сообщения об ошибке или сделать их недоступными. Следует избегать эту форму наследования, так как в данном случае нарушается правило подстановки.

Порождение класса для варьирования. Дочерний и родительский класс являются вариациями на одну тему и связь "суперкласс-подкласс" – произвольна.

Порождение класса для комбинирования. Комбинирование реализуется множественным наследованием. В одном классе сочетаются свойства

различных суперклассов, относящихся к различным предметным областям. Например, класс "Усилитель" может быть потомком классов "Элемент электрической схемы" и "Микросхема".

3.2. Множественное наследование

Множественное наследование используется в том случае, когда один и тот же класс рассматривается как сочетание нескольких независимых свойств и протоколов, которые оказываются существенными в различных приложениях. Если в различных приложениях есть базовые классы, которые объединяют в себе некоторые существенные свойства, то может быть создан подкласс, объединяющий в себе структуру и поведение нескольких суперклассов. При реализации множественного наследования возникает две основных проблемы: неопределённость в наименовании элементов суперклассов и повторное наследование.

Неопределенность в наименовании

Неопределённость в наименовании возникает в том случае, когда в разных суперклассах одного подкласса используются одинаковые имена для элементов интерфейсов (полей или методов).

```
721: class Amplifier{
722: protected:
723:   float maxPower; //максимальная мощность на нагрузке
724:   ...
725: };
726: class Chip{
727: protected:
728:    float maxPower; //максимальная потребляемая мощность
729:    ...
730: };
731: class DCAmplifier: public Amplifier, public Chip{
732: public:
733:   void f(float x);
734:   ...
735: };
736: void DCAmplifier:: f(float x) {
737: maxPower=x;    //неопределённость - какое из полей
                    //суперклассов должно быть изменено
738: ...
739: }
```

Для решения проблемы в C++ используется дополнительный квалификатор, указывающий на соотвествующий суперкласс.

```
740: void DCAmlifier:: f (float x){
741:   Chip:: MaxPower=x;
742:   Amplifier:: MaxPower=0.5*x;
743: }
```

Решение проблемы неопределённости наименования для методов класса.

```
744: #include <iostream.h>
745: void printA(){ //внешняя глобальная функция
746:   cout<<"это внешняя printA"<<endl;
747: };
748: class Base1{
749: public:
750:   int a;
751:   Base1(int m) {a=m;};
752: void printA(){
753:   cout<<"это Base1::a"<<endl;
754: };
755: class Base2{
756: public:
757:   float a;
758:   Base2(float r) {a=r};
759:   void printA(){
760:     cout<<"это Base2::a"<<a<<endl;
761:   }
762:   void print2A(){
763:     cout<<"это 2*Base2::a"<<2*a<<endl;
764:   }
765: };
766: class Derived: public Base1, public Base2{
767: public:
768:   char a;
769:   Derived (char l, int m, float r): Base1(m), Base2(r)
770:     {a=l;};
771:   void printA(){
772:     cout<<"это Derived::a"<<a<<endl;
773:   }
774:   void printAll(){
775:     printA();
776:     Base1::printA();
777:     Base2::printA();
778:     ::printA();
779:   }
780: };
781: main(){
782:   Base1 b1(10), *base1Ptr;
783:   Base2 b2(2.3), *base2Ptr;
784:   Derived d('z', 5, 7.1);
785:   b1.printA();
786:   b2.printA();
787:   d.printA(); //вызывается Derived::PrintA()
788:   d.print2A(); //вызывается Base2::print2A(),
                   //так как он не переопределен
789:   d.Base1::printA();
790:   d.Base2::printA();
791:   base1Ptr=&d;
792:   base1Ptr->printA(); //вызывается Base1::printA(),
                          //так как методы невиртуальные
793:   base2Ptr=&d;
794:   base2Ptr->printA();//вызывается Base2::printA()
795: }
```

Повторное наследование

Повторное наследование имеет место в тех случаях, когда суперклассы некоторого подкласса, в свою очередь, являются подклассами одного и того же суперкласса более высокого уровня.

```
796:  class Element {...};
797:  class Amplifier : public Element {...};
798:  class Chip : public Element {...};
799:  class DCAmplifier : public Amplifier, public Chip {...};
```

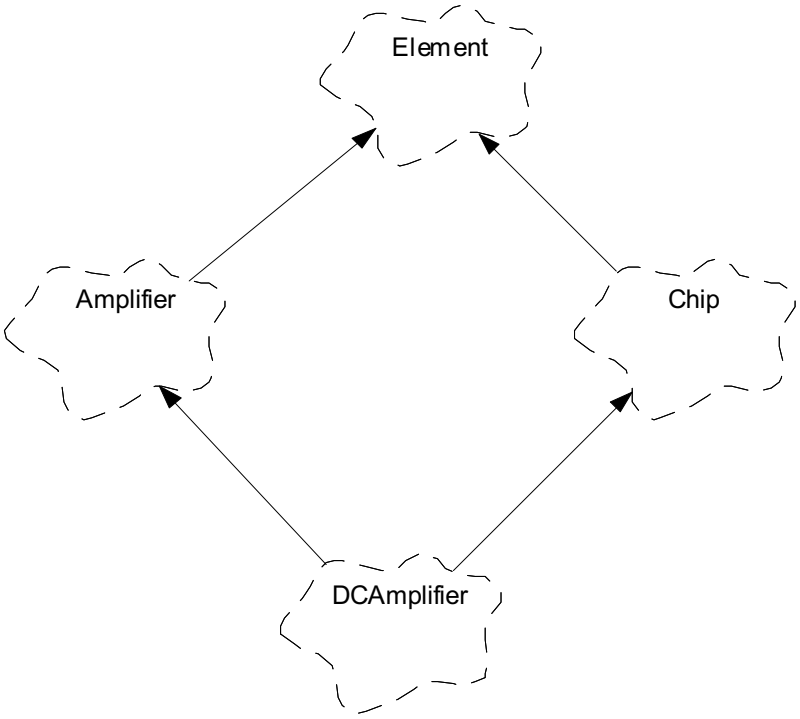

Рис. 2. Диаграмма классов для повторного наследования

Внешне проблема повторного наследования напоминает проблему неопределённости в наименовании, однако на самом деле имеется принципиальное отличие: сколько бы раз подкласс не наследовал структуру и поведение общего суперкласса, он детализирует свойства общего суперкласса, то есть только одного суперкласса.

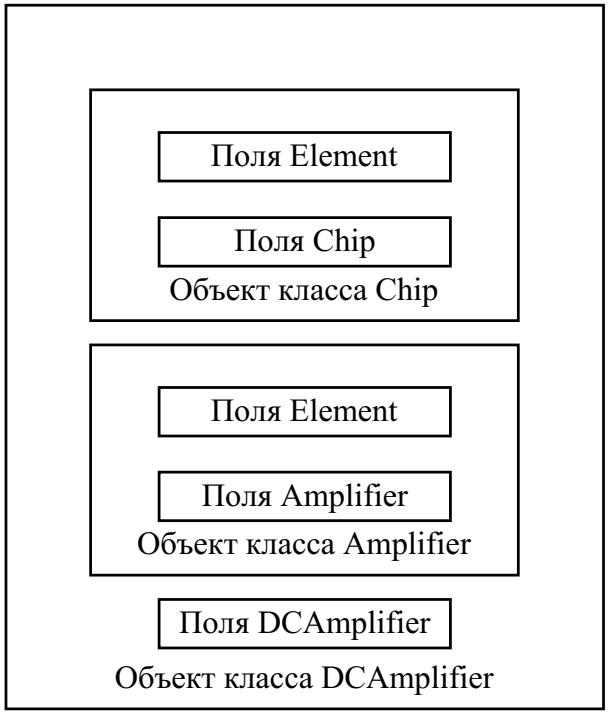

Рис. 3. Повторное наследование полей

В примере класс DCAmplifier дважды наследует поля класса Element. В этом случае оказывается, что методы, наследуемые от первого непосредственного суперкласса, обращаются к одной копии полей базового класса, а методы, наследуемые от второго суперкласса, обращаются ко второй копии полей суперкласса. Для исключения дублирования структуры базового класса в C++ реализуется механизм виртуального наследования. Этот механизм указывается в описании подкласса следующим образом.

```
800:  class Element{...};
801:  class Amplifier: virtual public Element{...}
```

При виртуальном наследовании в объекте подкласса создаётся указатель на структуру, наследуемую от виртуального базового класса. Это показано на рис. 4.

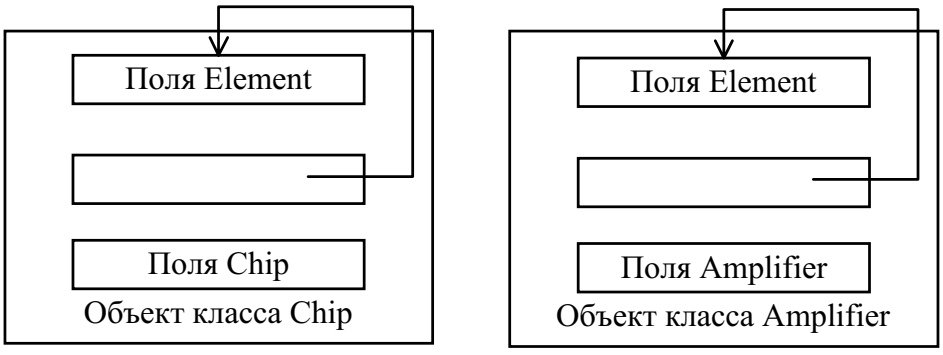

Рис. 4. Измененение структуры объектов класса при виртуальном наследовании

Для объектов классов `Chip` и `Amplifier` принципиально ничего не изменилось, но доступ к наследуемым полям осуществляется через указатель. При множественном наследовании указатели дают ссылку на структуру виртуального базового класса, которая наследуется только один раз.

```
802:  class DCAmplifier: public Amplifier, public Chip{...};
```

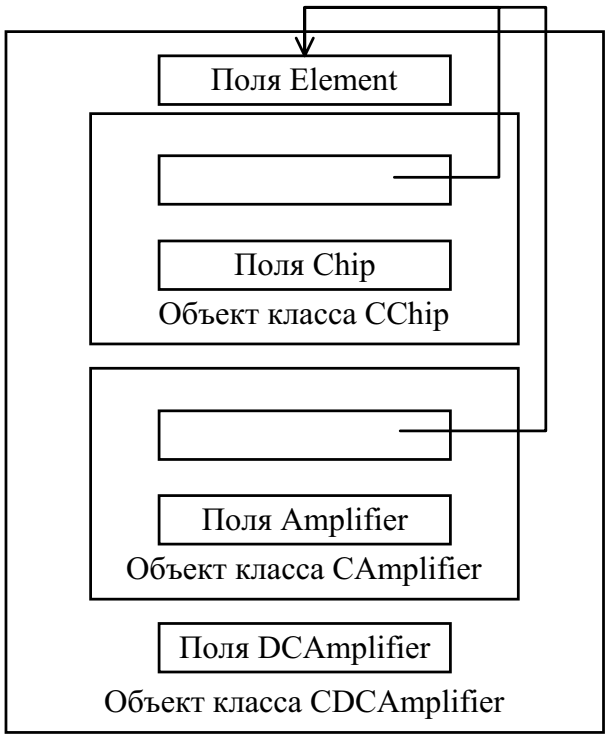

Рис. 5. Структура класса при повторном наследовании и объявлении базового класса виртуальным

Методы, наследуемые классом `DCAmplifier` от непосредственных предков, будут манипулировать одними и теми же полями структуры, наследуемой от базового класса.

В случае простого наследования при создании объекта выполняются конструкторы всех суперклассов, начиная с базового класса и заканчивая конструктором данного класса. При множественном наследовании эта последовательность для непосредственных суперклассов уточняется в списке инициализаторов в релизации конструктора. Инициализаторы позволяют указать параметры, с которыми будут вызываться конструкторы суперклассов. Важная проблема заключается в том, что в конструкторах непосредственных суперклассов могут вызываться различающиеся конструкторы базового класса. Поэтому в том классе, в котором возникает проблема повторного наследования, необходимо явно указать, какой конструктор виртуального базового класса должен быть вызван.

```
803: class A{
804: public:
805:    A(){...};
806:    A(int i){...};
807:    A(float d){...};
808: };
809: class B: virtual public A{
810: public:
811:     B(): A(7){...}; //конструктор для A с целым параметром
812: };
813: class C: virtual public A{
814: public:
815:    C():A(3.141){...}; //конструктор для A
                          //с действительным параметром
816: }
817: class D: public B, public C{
818: public:
819:    D(): A(3), B(), C(){...};
820: };
821: main(){
822:   D d();  //вызов конструктора класса A в конструкторах
               // B и C игнорируется, вызывается A(3),
               //потом выполняется B::B(), потом C::C()
823: ...
824: }
```

Это единственная ситуация, в которой внутри класса разрешается использовать конструктор другого класса, который не является непосредственно предшествующим предком. Конструкторы виртуальных классов должны обязательно вызываться первыми, то есть до вызова конструкторов невиртуальных классов. В противном случае может возникнуть неопределенность, какие параметры (или какой вариант конструктора) надо использовать при инициализации объекта при выполнении конструктора виртуального базового класса. Деструкторы выполняются в порядке, обратном выполнению конструкторов.

3.3. Отношение использования

Отношение использования – это такое отношение между классами, при котором один класс не может быть описан без упоминания другого (используемого) класса.

Перечислим некоторые ситуации, в которых возникает отношение использования классов:

1) методу данного класса передается объект используемого класса, или указатель на такой объект в качестве параметра;

2) поле данного класса является объектом используемого класса или указателем на такой объект;

3) в методе данного класса создаются локальные объекты используемого класса;

4) метод данного класса выполняет действия над глобальным объектом используемого класса.

В качестве примера рассмотрим программную модель некоторого электронного устройства. Сконструируем следующие классы: `Device` – электронное устройство; `Chip` – микросхема, которая может входить в состав устройства; `Signal` – электронный сигнал; `Beep` – звуковой сигнал; `ostream` – выходной поток, определенный в стандартной библиотеке iostream.

```
825: #include <iostream.h>
826: class Chip{
827: public:
828:   void initiate();//привести в начальное состояние
829:   ...
830: };
831: class Signal{
832: public:
833:   float getValue(){...};//получить значение сигнала
834: };
835: class Beep{
836: public:
837:   Beep(int seconds){...}
838: };
839: class Device{
840: private:
841:   Chip chip; //объект - поле класса
842: public
843:   void output(Signal &s){ //вывод значения сигнала
844:     cout<<s.getValue(); //доступ к глобальной переменной
845:   }
846:   void reset(){
847:     Beep *pbeep; //создание локального указателя на объект
848:     pbeep=new Beep(10);
849:     delete pbeep;
850:     chip.initiate();
851:   }
852: };
853: int main(){
854:   Device divice;
855:   Signal signal;
856:   ...
857:   device.output(signal);
858:   ...
859: }
```

В данном примере класс `Device` использует классы `Chip`, `Signal`, `Beep` и `ostream`. Класс `Signal` используется в интерфейсной части, то есть объект этого класса должен быть виден так же, как и объект класса `Device` в момент выполнения метода `output()`, и передаваться в каче-

стве параметра метода. Объект `chip` класса `Chip` объявляется полем класса `Device`. Объект `*pbeep` является локальным в реализации метода `reset()`. Объект cout класса ostream оказывается глобальным по отношению к классу `Device` и используется в методе `output()`.

Видимость на уровне классов и на уровне объектов

При рассмотрении отношения использования возникает вопрос: может ли один объект некоторого класса использовать неоткрытые поля и методы другого объекта *того же класса*. В C++ такое использование является корректным, то есть реализуется видимость на уровне классов.

```
860: class A{
861: private:
862:   int m;
863: public:
864:   void f(A &p){
865:    p.m=0; //изменяется закрытое поле объекта p
866:   };
867: };
868: main(){
869:   A p1,p2;
870:   p2.f(p1); //изменение поля p1.m другим объектом
                 //того же класса
871: }
```

Важно различать понятия "видимость" и "доступ к элементам классов". Элементы классов могут быть видимы, но недоступны. Приведем пример.

```
872: int m; //глобальная переменная
873: class A{
874: private:
875:   int m
876: }
877: class B:public A{
878:   void f(){m=0;} //ошибка, A::m описана как private
879: }
```

Если бы спецификаторы доступа управляли видимостью, то поле `A::m` было бы невидимым, и была изменена глобальная переменная `m`.

Дружественные отношения

Отношение использования ограничивается квалификаторами доступа и последовательностью объявления классов. Гибкий механизм ограничения доступа, точнее – ослабления ограничения, даёт объявление дружественных классов. Для этого в определении класса указывается, какие другие классы, методы классов или функции-утилиты могут иметь доступ к закрытым элементам данного класса.

```
880: class Device; //предварительное объявление класса
```

```
class Chip {
friend class Device; //объявление дружественного класса
private:
  void f(); //закрытый метод
  ...
};
class Device{
...
Chip chip;
public:
   void g();
   ...
};
...
void Device::g(){
  chip.f(); // закрытая функция здесь доступна
}
```

Аналогично можно объявлять дружественные функции и отдельные методы классов.

```
class Complex{
private:
  double rp;
  double ip;
public:
  Complex(double a, double b);
friend double abs(Comlex& x);
};
Complex::Complex(double a, double b)
  {rp=a; ip=b};
double abs(Complex& x)
  {return sqrt(x.rp*x.rp+x.ip*x.ip);} //доступны закрытые
                                      //поля класса
main(){
  Complex x(3.1-0.5);
  cout<<abs(x);
};
```

3.4. Отношение наполнения

Отношение наполнения – это отношение между классами, при котором классы формируются из общей структуры, называемой параметризованным (обобщённым) классом, путём указания фактических типов элементов параметризованного класса. Термины "параметризованный класс", "обобщённый класс", "шаблон класса" являются синонимами.

Параметризованный класс представляет собой описание класса с неопределёнными предварительно типами, которые указываются в виде формальных параметров. Обобщённый класс обобщает классы с одинаковыми структурой и поведением. Например, динамически распределяемые массивы целых чисел и действительных чисел отличаются друг от друга

только типом элементов массива, а общая структура классов и алгоритмы методов одинаковы.

```
914: class IntArray{//целочисленный массив
915:   int *a;
916:   int size;
917: public:
918:   IntArray(int sz); //создание массива
919:   ~IntArray();//удаление массива
920: };
921: class FloatArray{//массив вещественных чисел
922:   float *a;
923:   int size;
924: public:
925:   FloatArray(int sz);
926:   ~FloatArray();
927: };
928: IntArray::IntArray(int sz){
929:   size=sz;
930:   a=new int[size];
931: }
932: FloatArray::FloatArray(int sz) {
933:   size=sz;
934:   a=new float[size];
935: InitArray::~InitArray(){
936:   delete[] a;
937: }
938: FloatArray::~FloatArray(){
939:   delete[] a;
940: }
941: void main(){
942:   IntArray intArray(10);
943:   FloatArray floatArray(20);
944:   ...
945: }
```

Шаблон класса позволяет обобщить оба класса в единую запись. Тогда объявления конкретных классов с заданным типом элементов массива могут быть получены из шаблона путем указания конкретного типа.

```
946: template <class T>  class Array{
947:   T* a;
948:   int size;
949: public:
950:   Array(int sz);
951:   ~Array();
952:   void initiate();//инициализация элементов
953: };
954: template <class T> Array <T>::Array(int sz){
955:   size=sz;
956:   a=new T[size];
957: }
958: template <class T> Array <T>::~Array()
959:   {delete[] a;}
960: template <class T> void Array <T>::initiate(){
961:   int i;
962:   for(i=0; i<size; i++)
```

```
963:     a[i]=0;
964: }
965: void main(){
966:   Array<int> intArray(10);//создание варианта
                              //целочисленного массива
967:   Array<float> floatArray(20);//создание варианта
                              //массива действительных чисел
968:   intArray.Initiate();
969:   floatArray.Initiate();
970: }
```

Специализация методов обобщённого класса

Пусть в примере предполагается параметризация шаблона символьным типом, но при этом начальными значениями должны быть пробелы, а не символы с кодом 0. Тогда необходимо добавить специальную реализацию метода `initiate()`.

```
971: void Array <char>::initiate(){
972:   int i;
973:   for(i=0; i<size; i++)
974:      ia[]=’ ’;
975: };
976: void main(){
977:   Array <float> floatArray(20);
978:   Array <char> charArray(10);
979:   floatArray.initiate(); //инициализация нулями
980:   charArray.initiate();  //инициализация пробелами
981: };
```

Эквивалентность типов

Два конкретных класса, полученных из обобщённого класса, являются эквивалентными, если совпадают шаблоны и фактические параметры имеют одинаковые значения. Константные типы и формальные параметры могут использоваться одновременно.

```
982:  template <class T, int SIZE>
983:  class Array{
984:    T *a;
985:    Array();
986:    ~Array();
987:    void initiate();
988:  }
989:  template <class T, int SIZE>
990:  Array <T, SIZE>::Array()
991:    {a=new T[SIZE];};
992:  template <class T, int SIZE>
993:   void Array <T, SIZE>::initiate(){
994:    int i;
995:    for(i=0; i<SIZE; i++)
996:      a[i]=0;
997:  }
998:  void Array <char, 10>::initiate(){
999:    int i;
```

```
1000:   blank=’ ’;
1001:   for(i=0; i<SIZE; i++)
1002:     a[i]=blank;
1003: };
1004: main(){
1005:   Array <float, 15> floatArray;
1006:   Array <char, 10> charArray10;
1007:   Array <char, 6> charArray6;
1008: //классы объектов charArray10 и charArray6 различны
1009:   floatArray.initiate();
1010:   charArray10.initiate(); //инициализация пробелами
1011:   charArray6.initiate(); //инициализация нулями
1012:   return 0;
1013: };
```

Отношение использования для обобщённых классов

Использование обобщённого класса в обычном (конкретном) классе:

```
1014: class AgrStack{
1015:   int top;
1016:   Array <int> arr;
1017: public:
1018:   AgrStack(int n):arr(n) {top=-1;}
1019:   ~AgrStack(){;}
1020: void init()
1021:   {arr.initiate()};
1022: main{
1023:   AgrStack st(4);
1024:   st.init();
1025:   return 0;
1026: }
```

Использование обобщенного класса другим обобщенным классом и дружественные отношения между обобщенными классами

Часто шаблоны классов используются для описания абстрактных типов данных, например связных списков.

```
1027: template<class Type>class List;
                //предварительное объявление шаблонного класса
1028: template <class InfoType> class Element{ //элемент списка
1029: friend class List<InfoType>;
                //список - дружественный класс
1030: protected:
1031:   InfoType info;
1032:   Element *next;
1033: public:
1034: Element(InfoType *t):Info(t)
1035:    {next=0;}
1036: };
1037: template <class Type> class List{
1038: protected:
1039:   Element <Type> *head;
1040: public:
1041:   List() {head=0;}
1042:   ~List();
```

```
1043:   void add(Type &val);
1044: };
1045: template <class Type> List <Type>::~List(){
1046:   Element <Type> *ptr;
1047:   while(hHead!=0){
1048:     ptr=head;
1049:     head=head->next;
1050:     delete ptr;
1051:   }
1052: }
1053: template <classType> void List <Type>::Add(Type &val){
1054:   Element <Type> *ptr;
1055:   ptr=new Element<Type>(val);
1056:   ptr->next=head;
1057:   head=ptr;
1058: }
1059: //первый вариант головной функции
1060: int main(){
1061:   List <int> lst1; //локальные объекты
1062:   List <char> lst2;
1063:   int i=3;
1064:   char ch='V';
1065:   lst1.add(i);
1066:   lst2.add(ch);
1067:   ...
1068:   List <int> *plist;
1069:   plist=new List<int>; //динамический объект
1070:   int i=3;
1071:   plist->add(i);
1072:   delete plist;
1073:   return 0;
1074: }
```

Отношение наследования обобщенных классов

В отношении наследования возможны различные сочетания обобщенных и конкретных классов. Рассмотрим пример, в котором обобщенный класс является подклассом конкретного суперкласса.

```
1075: class Base{
1076: float x;
1077: public:
1078:   Base(float newX)
1079:     {x=newX;}
1080: };
1081: template <class Type> class Derived:public Base{
1082:   Type y;
1083: public:
1084:   Derived(float newX,Type &newX):Base(newX),y(newY){;}
1085: };
1086: int main(){
1087:   char c='z';
1088:   Derived <char> d(3.0,c);
1089:   return 0;
1090: }
```

В следующем примере конкретный класс является подклассом обобщенного класса.

```
1091: class IntStack:public Array <int>{
1092:   int top;
1093: public:
1094:   IntStack(int n):Array <int>(n){top=-1;}
1095:   int isEmpty(){return top==-1;}
1096:   void initiate(){
1097:     Array<int>::initiate();
1098:     top=-1;
1099:   }
1100: };
1101: void main(){
1102:   IntStack st(10);
1103:   st.initiate();
1104: }
```

В следующем примере обобщенный класс является подклассом обобщенного класса.

```
1105: template <class Type> class Stack : public Array <Type>{
1106:   int top;
1107: public:
1108:   Stack(int n):Array<Type>(n),top(-1) {;}
1109:   int isEmpty(){return top==-1;}
1110:   void initiate(){
1111:     Array<Type>::initiate();
1112:     top=-1;
1113:   }
1114: };
1115: void main(){
1116:   Stack <char> st(15);
1117:   st.initiate();
1118: }
```

Существуют отдельно поставляемые библиотеки обобщённых классов. Обычно это "заготовки" абстрактных типов данных: односвязные и двусвязные списки, бинарные деревья, очереди, стеки, массивы, множества. Объекты этих классов содержат некоторую совокупность объектов, типы которых определяются при конкретизации, поэтому такие классы называют контейнерными. Библиотеки содержат также функции для работы с контейнерами, например, функции сортировки, выборки и так далее.

Глава 4. **ОСОБЕННОСТИ ОБЪЕКТНО-ОРИЕНТИРОВАННОГО ПРОГРАММИРОВАНИЯ НА C++**

4.1. Передача параметра по ссылке с запретом его модификации

Передача параметра по ссылке во многих случаях оказывается более предпочтительной, чем передача параметра по значению. При этом экономится память, повышается быстродействие и отпадает необходимость в

разработке конструктора копирования для класса передаваемого параметра. Запрет модификации параметра повышает надежность реализации.

```
1119: struct Time{
1120:   int hour;
1121:   int minute;
1122:   int second;
1123: };
1124: void Output(const Time& t) {
1125:   cout<<t.hour;// правильно
1126:   t.hour=0;    // ошибка
1127: }
```

4.2. Константные методы, константные поля и объекты-константы

Константными объявляются методы, которые не должны изменять поля объекта. Как правило константными являются методы доступа к свойствам объекта.

```
1128: int A::getValue() const {return Value}
```

Компилятор позволяет выявить следующие ошибки реализации: константный метод изменяет поля, константный метод вызывает неконстантный метод. Использование константных методов повышает надёжность программы.

Объекты могут быть объявлены константами. Такой объект не может быть изменён присваиванием, поэтому все поля должны быть сразу инициализированы конструктором. Очевидно, что для объекта-константы допустимо использование только конструкторов, деструктора и константных методов.

```
1129: class Time{
1130: private:
1131:    int hour;
1132: public:
1133:    void setTime(int hr);
1134:    int getHour() const;
1135:    Time(int hr);
1136: }
1137: void Time::setTime(int hr){
1138:   hour=hr;
1139: }
1140: int Time::getHour() const{
1141:   return Hour;
1142: }
1143: Time::Time(int hr) {
1144:   hour=hr
1145: };
1146: main(){
1147:   const Time noon(24);
1148:   noon.setTime(0); //ошибка
1149:   cout<<noon.getHour(); //правильно
1150: }
```

Если поле объекта не должно изменяться на всём протяжении существования объекта, то оно может быть объявлено константным. Значение константного поля устанавливается только инициализатором, указываемым в конструкторе.

```
1151: class Inc{
1152: private:
1153:   int value;
1154:   const int increment;
1155: public:
1156:   inc(int v, int i);
1157:   void addIncrement();
1158: }
1159: Inc::Inc(int v, int i):increment(i) {
1160:   value=v;
1161: }
1162: Inc::addIncrement(){
1163:   value=value+increment;
1164: //  increment=2; //ошибка
1165: };
1166: main(){
1167:   Inc counter(10, 2);
1168:   counter.addIncrement();
1169:   return 0;
1170: }
```

4.3. Статические методы и статические поля класса

Если имеется необходимость совмещения полей различных объектов одного класса, то это поле объявляется статическим. Статическое поле часто называют полем класса, а нестатическое поле – полем объекта. Статическое поле по своей сути – глобальная переменная, связанная с классом, поэтому оно может использоваться до создания объектов. Статический метод – это такой метод класса, который выполняет действия только со статическими полями. Статический метод также может быть использован до создания экземляров класса.

```
1171: class Student{
1172:   int id; //порядковый номер студента
1173:   static int counter; //каждый студент знает сколько
                            //всего студентов
1174: public:
1175:   Student(int I) {
1176:     counter=counter+1;
1177:     id=counter;
1178:   }
1179:   static int howMany()//статический метод
1180:     {return counter;}
                  //используется статическое поле
1181:   void getId(){
1182:     cout<<id<<endl;
1183:     cout<<counter<<endl;
1184:   }
```

```
1185: };
1186: int Student::counter=0; //использование статического поля
                              //до создания экземляров класса
1187: int main(){
1188:   cout<<Student::howMany();
1189:   Student st1, st2, st3;
1190:   cout<<Student::howMany();
1191:   cout<<St2.howMany();
1192:   St2.GetId();
1193:   return 0;
1194: }
```

4.4. Конструктор копирования

Для объекта любого класса по умолчанию определена операция поэлементного копирования. Однако такое копирование может приводить к структурной неопределённости (нарушение индивидуальности объекта). Поэтому для копирования необходимы специальные методы. Особый случай – копирование при создании объекта. Необходимость в копировании возникает при создании именованного объекта по образу другого объекта, а также при создании неименованного (временного) объекта, например при передаче объекта в качестве параметра функции по значению.

Имеются особенности использования конструкторов при наследовании классов и при агрегации объектов. В последнем случае различаются следующие ситуации:

1) поля агрегирующего объекта имеют конструкторы копирования, сам объект – не имеет;

2) поля и агрегирующий объект имеют конструкторы копирования.

Конструктор копирования по умолчанию вызывает конструктор копирования родительского класса. Если для дочернего класса явно задан конструктор копирования, то его реализация должна явным образом вызывать конструктор копирования родительского класса. В противном случае не будет выполнено копирование полей, объявленных в родительском классе.

Конструктор копирования по умолчанию агрегирующего объекта вызывает конструкторы агрегируемых объектов. Если в агрегирующем классе явно задан конструктор копирования, то он должен содержать инициализаторы для копирования полей. В противном случае не будет выполнено копирование агрегированных объектов.

```
1195: class Base{
1196: public:
1197:   char name;
1198:   int id;
1199:   Base(){name='U', id=0;} //конструктор без параметров
1200:   Base(const Base& sb) { //конструктор копирования
1201:     name=sb.name; id=sb.id;
```

```
1202:   }
1203: };
1204: class Array:public Base{
1205: public:
1206:   int size;
1207:   float *ptr;
1208:   Array(int sz){
1209:      size=sz;
1210:      str=new float[size];
1211:   }
1212:   Array(const Array& sa); //конструктор копирования
1213: };
1214: Array::Array(const Array& sa):Base(sa){
        //если явно не вызывать конструктор копирования предка,
        //то копирование будет неправильным
1215:   size=sa.size;
1216:   ptr=new float[size];
1217:   for(int i; i<size; i++){
1218:     ptr[i]=sa.ptr[i];
1219: }
1220: class TwoArrays{
1221: public:
1222:   Array fArr;
1223:   Array sArr;
1224:   TwoArrays():fArr(3), sArr(4){;}
          //нет явно заданного конструктора копирования
          //в агрегирующем классе
1225: };
1226: class NewTwoArrays{
1227: public:
1228:   Array fArr;
1229:   Array sArr;
1230:   NewTwoArrays():fArr(3), sArr(4){;}
1231:   NewTwoArrays(const NewTwoArrays & nta):
          fArr(nta.fArr), sArr(nta.sArr){;}
            //конструктор копирования агрегирующего класса
1232: };
1233: void func(Array param){//функция-утилита, параметр,
                            //передаваемый по значению - объект
1234:   for(int i=0; i<param.size;i++)
1235:     param.ptr[i]=0;
1236: }
1237: main(){
1238:   Base b1;
1239:   Base b2(B1);
1240:   Array x1(3);
1241:   Array x2(x1);
1242:   TwoArrays t1;
1243:   TwoArrays t2=T1;
1244:   NewTwoArrays n1;
1245:   NewTwoArrays n2=n1;
1246:   func(x2);
1247:   return 0;
1248: }
```

4.5. Перегрузка операторов

Для выполнения действий над переменными встроенных типов в C++ предусмотрен набор операторов (+, -, /, *, ++, -- и др.). Операторы могут быть бинарными (двуместными) и унарными (одноместными) в зависимости от количества аргументов. Одному и тому же символу операции даже для встроенных типов соответствуют различные операции. Например, оператор умножения "*" для целых и для действительных чисел реализуется различными последовательностями машинных команд. Такие операторы называются перегружаемыми или совместно используемыми. C++ предусматривает перегрузку операторов для новых конструируемых типов данных со следующими ограничениями:

1) невозможно изменить старшинство операторов;

2) запрещено изменять назначение операторов для встроенных типов, так как это может привести к изменению языка;

3) запрещено конструировать новые операторы. Например, объявить оператор "**" нельзя.

При создании нового типа автоматически генерируется код для оператора присваивания "=" и оператора вычисления адреса "&". Оба эти оператора могут быть перегружены явно.

Функция-оператор может быть или утилитой (в том числе дружественной каким-либо классам), или методом какого-либо класса. При перегрузке операторов "()", "[]", "->", "=" функция-оператор должна быть методом класса, для других операторов это необязательно. Если функция-оператор является методом класса, то самый левый операнд должен быть объектом (или ссылкой) данного класса. Если же самый левый операнд должен быть переменной встроенного типа, а правый – объектом класса, то функция-оператор может быть только утилитой. Для перегрузки операторов используется зарезервированное слово `operator`.

Ниже приведен пример перегрузки операторов для типов, не являющихся классом.

```
1249: struct Complex{float re,im;};
1250: Complex operator+(Complex c1,Complex c2) {
1251:   Complex c;
1252:   c.re=c1.re+c2.re;
1253:   c.im=c1.im+c2.im;
1254:   return c;
1255: };
1256: Complex operator*(Complex c1,Complex c2){
1257:   Complex c;
1258:   c.re=c1.re*c2.re-c1.im*c2.im;
1259:   c.im=c1.im*c2.re+c2.im*c1.re;
```

```
1260:   return c;
1261: };
1262: main(){
1263:   Complex x1={1,2};
1264:   Complex x2={3,4};
1265:   Complex x3;
1266:   x3=x1+x2;
1267:   x1=x1+x1; //в сложении передается параметр по значению,
                  //а не по ссылке
1268:   x3=x1*x2;
1269:   return 0;
1270: }
```

Перегрузка операторов для класса имеет некоторые особенности. Оператор присваивания не наследуется аналогично тому, как не неследуются конструкторы и деструкторы.

```
1271: class Comp{
1272: protected:
1273:   float re,im;
1274: public:
1275:   Comp(){re=0.0;im=0.0;}
1276:   Comp(float x,float y)
1277:     {re=x;im=y;}
1278: friend Comp operator+(const Comp& c1, const Comp& c2);
1279:   int operator==(const Comp& c)const
1280:     {return ((re==c.re)&&(im==c.im));}
1281:   Comp operator!()const { //комплексно сопряженное число
1282:     Comp c;
1283:     c.re=re;
1284:     c.im=-im;
1285:     return c;
1286:   }
1287:   float& operator[](int i) { //оператор скобки
1288:     if(i==1)return re;
1289:     else return im;
1290:   }
1291: };//класс Comp
1292: Comp operator+(const Comp& c1, const Comp& c2){
1293:   Comp c;
1294:   c.re=c1.re+c2.re;
1295:   c.im=c1.im+c2.im;
1296:   return c;
1297: }
1298: //наследование
1299: class NComp:public Comp{
1300:   char name;
1301: public:
1302:   NComp(){name='U';}
1303:   NComp(float x,float y,char n):Comp(x,y){name=n;}
1304: };
1305: //головная программа
1306: main(){
1307:   Comp y1(1,2);
1308:   Comp y2(3,4);
1309:   Comp y3;
1310:   y1=y1+y1;
```

```
1311:   if(y1==y2)
1312:     y3=y1+y2;
1313:   y1=!y1;
1314:   y1[1]=0.0;
1315:   y1[2]=y1[1];
1316:   NComp z1(1,2,'D');
1317:   NComp z2(1,2,'A');
1318:   if(z1==z2) //функция-оператор == наследуется
1319: //  z1=z1+z2; //ошибка типизации
1320:   else
1321:     z1[1]=2*z1[2]; //функция-оператор [] наследуется
1322:   return 0;
1323: }
```

4.6. Потоковый ввод/вывод в C++

В языке C++ отсутствуют какие-либо средства ввода/вывода. Эти возможности реализуются различными стандартными библиотеками. Одна из библиотек, называемая iostream, использует возможности объектно-ориентированного программирования. В ней используются следующие понятия: *поток* – последовательность байтов, *входной поток* – средство перемещения байтов от внешнего устройства в оперативную память, *выходной поток* – средство перемещения байтов из оперативной памяти к устройству.

В библиотеке потокового ввода/вывода обеспечиваются возможности низкого уровня (неформатный ввод/вывод, управление передачей отдельных байтов) и высокого уровня (совокупность байтов имеет свойства, определяемые типами – форматный ввод/вывод).

Средства ввода/вывода распределены по следующим библиотечным файлам:

1) *iostream.h* – основная библиотека со стандартными потоками ввода/вывода.

2) *iomanip.h* – библиотека с манипуляторами для форматного ввода/вывода.

3) *fstream.h* – библиотека файловой системы ввода/вывода.

4) *strstream.h* – библиотека внутреннего форматного ввода/вывода для выполнения операций над строками символов таким образом, как-будто они представлены в виде файлов.

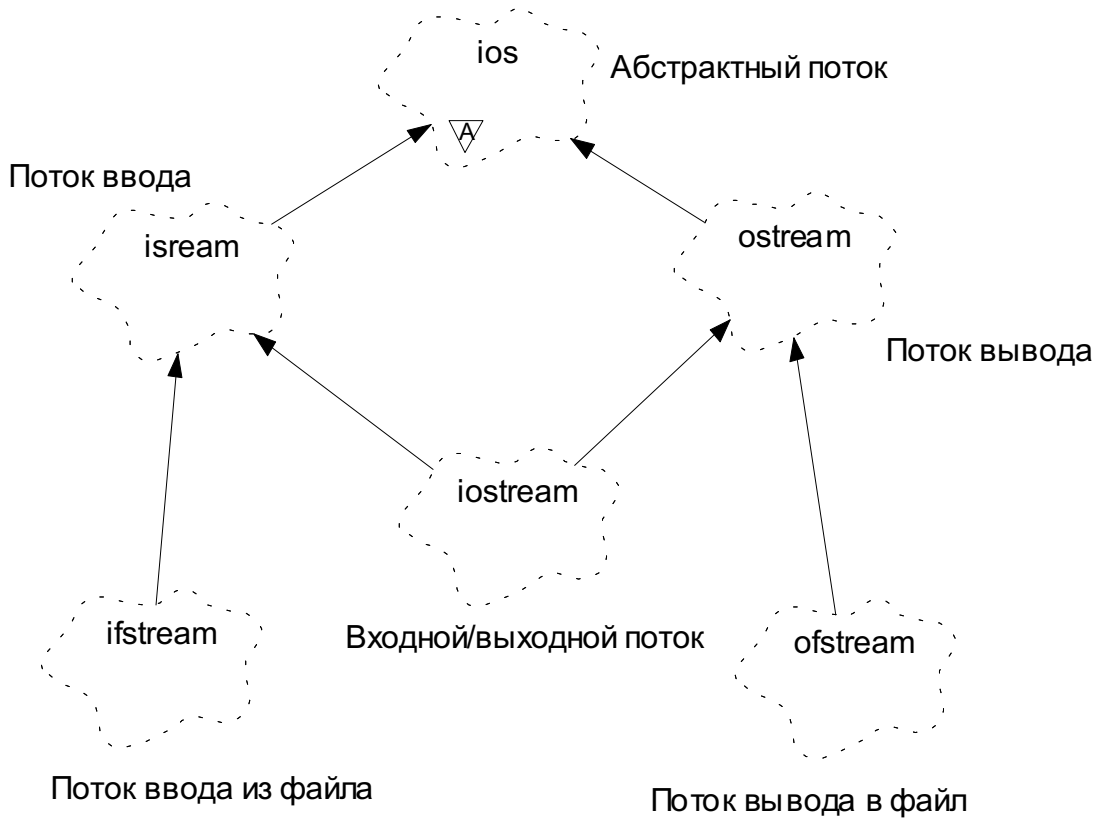

Рис. 6. Диаграмма наследования основных классов для организации ввода/вывода

В библиотеке iostream.h объявлены следующие *объекты*, которые используются в программах в качестве глобальных переменных:

`cin` – поток, связанный со стандартным входным устройством (обычно клавиатура);

`cout` – поток, связанный со стандартным выходным устройством (обычно экран);

`cerr` – поток, связанный со стандартным устройством вывода сообщений об ошибке (небуферизованный быстрый вывод);

`clog` – поток, связанный со стандартным устройством вывода сообщений об ошибке (буферизованный поток, который выводится после заполнения буфера).

Глобальные объекты принадлежат специальным классам, являющимся потомками основных классов потоков: `cin` принадлежит классу, производному от `istream`; `cout, cerr, clog` принадлежат классу, производному от `ostream`.

Потоковый вывод

Класс ostream обеспечивает форматный и неформатный вывод при помощи перегруженного оператора включения в поток "<<", вывод символов при помощи метода `put`, неформатный вывод при помощи метода `write`, использование манипуляторов для вывода целых чисел в десятичный, восьмиричный и шестнадцатиричный форматы записи, выравнивание по ширине, указание формата представления чисел.

Оператор вставки в поток << является совместно используемым оператором (для стандартных типов данных оператор "<<" – это оператор сдвига). В библиотеке имеется ряд определений операторов вставки в поток данных различных стандартных типов. Приведем для примера некоторые заголовки.

```
1324: ostream& ostream::operator<<(const int&){...}
1325: ostream& ostream::operator<<(int){...}
```

Выражению `cout<<121` соответствует перегруженный оператор (метод класса) с параметром типа `int`. В соответствии с объявлением это выражение возвращает ссылку на первый параметр – объект потока `cout`. Возврат ссылки на поток в операторе вставки в поток позволяет использовать конкатенацию операторов.

```
1326: cout<<121<<0<<13; //равносильно последовательному
                        //выполнению cout<<121; cout<<0; cout<<13;
```

Имеется особая реализация оператора вставки в поток для типа `char*`, которая выводит не адрес символа, а строку символов до первого символа с нулевым кодом.

```
1327: char *string="test"; //строка оканчивающаяся нулём
1328: cout<<sting; //выводятся символы "test",
                   //начиная с указателя string
```

Формат представления данных при выводе изменяется либо методами потока, либо специальными программами-утилитами, которые называются манипуляторами.

```
1329: double f=sqrt(2.0);
1330: cout<<f<<endl;                       //результат 1.414214
1331: cout.precision(3);                   //метод для потока
1332: cout<<f<<endl;                       //результат 1.414
1333: cout<<setprecision(2)<<f<<endl;      //результат 1.41
```

В рассмотренном примере используется манипулятор `endl`. Прототип манипулятора задается в библиотеке приблизительно следующим образом.

```
1334: ostream& endl(ostream&);
```

В поток вставляется адрес функции-манипулятора, после чего выполняется вызов функции. Манипулятор `endl` вставляет в поток символ перевода строки и освобождает буфер. В библиотеке имеются также и другие манипуляторы: `ends`, `flush`, `dec`, `hex`, `oct`, `ws`.

Потоковый ввод

В библиотеке имеется набор перегруженных операторов извлечения из входного потока ">>" для различных стандартных типов данных. Приведем некоторые из них.

```
1335: istream& istream::operator>>(int&);
1336: istream& istream::operator>>(char&);
```

Благодаря возврату ссылки на тот же поток становится возможной конкатенация операторов извлечения из потока.

```
1337: cin>>x>>y; //равносильно последовательному выполнению
                 //операторов cin>>x; cin>>y;
```

В библиотеке предусмотрен оператор приведения типа возвращаемого из оператора извлечения возвращаемого значения к типу `void*`. Благодаря преобразованию возвращаемого типа в указатель средствами базового класса оператор извлечения может использоваться в логическом выражении.

```
1338: while(cin>>x)  //цикл ввода данных
1339:   {if(x>y)...}
```

Класс `istream` предусматривает методы для чтения, например метод `get()` возвращает код символа; метод `eof()` – возвращает 0, если не был обнаружен символ конец файла, и 1 в противном случае.

```
1340: while((c=cin.get())!=EOF){...} //здесь EOF - символ
                                    //конца файла
```

При вводе текстовых строк следует учитывать, что запись информации в память не контролируется. Поэтому размер буфера для текстовой строки должен быть выбран с запасом. Строка вводится до первого пробела. После окнончания чтения потоком к строке добавляется символ с нулевым кодом.

```
1341: char buffer[1024];
1342: cin>>buffer; //чтение
1343: cout<<buffer; //вывод
```

Перегрузка операторов потокового ввода и вывода

Для организации ввода и вывода конструируемых типов данных (например объектов какого-либо класса) следует перегрузить соответствующие операторы извлечения из потока и вставки в поток.

```
1344: class Complex{
1345: friend ostream &operator<<(ostream&, const Complex&);
1346: friend istream &operator>>(istream&, Comlex&);
1347: private:
1348:   float re;
1349:   float im;
1350: };
1351: ostream &operator<<(ostream &output, const Complex &comp){
1352:   output<<"re="<<comp.re<<" im="<<comp.im;
```

```
1353:   return output;
1354: }
1355: istream &operator>>(istream &input, Complex &comp){
1356:   input<<comp.re;
1357:   input>>comp.re;
1358:   return input;
1359: }
1360: void main(){
1361:   Complex A, B;
1362:   cin>>A>>B;
1363:   cout<<A<<B;
1364:   ...
1365: }
```

Перегружаемые операторы << и >> при создании новых типов объектов объявляются как утилиты. Они не могут быть объявлены как методы класса, так объект нового класса появляется только в правой части списка параметров.

Работа с файлами последовательного доступа

Для создания файла последовательного доступа необходимо объявить переменную класса `ofstream` или производного от него класса. Связывание физического файла с файловой переменной может быть выполнено при вызове конструктора, как показано в примере, или методом класса `ofstream`. Для вывода в файл используются операторы вставки в поток, наследуемые от класса `ostream`, и перегруженные функции-операторы для сконструированных классов.

```
1366: #include <iostream.h>
1367: #include <fstream.h>
1368: main(){
1369:   ofstream outputFile("x.dat", ios::out);
1370:   if(!outputFile)
1371:     cerr<<”ошибка”<<endl;
1372:   else{
1373:     int index;
1374:     float fNum;
1375:     while(cin>>index>>fNum)
1376:       outputFile<<index<<’’<<fNum<<endl;
1377:   }
1378:   return 0;
1379: }//выполнение деструктора, закрытие файла
```

Первый параметр конструктора – имя физического файла, второй параметр – константа, определяющая режим открытия файла. Эти константы объявлены внутри библиотечного класса `ios`.

Некоторые режимы создания файлов:

`ios::out` – файл для вывода;

`ios::in` – файл для ввода;

`ios::app` – добавлять данные только в конец файла;

`ios::noreplace` – если файл существует, то генерировать ошибку файловой операции;

`ios::nocreate` – если файл не существует, то не создавать новый;

`ios::trunc` – очистить файл.

Для чтения данных из файла последовательного доступа необходимо создать переменную класса `ifstream` или производного от него. Ввод данных осуществляется перегруженными операторами извлечения из потока, наследуемыми от класса `istream`, и перегруженными функциями-операторами для сконструированных классов.

```
1380: #include<iostream.h>
1381: #include<fstream.h>
1382: int main(){
1383:   int index;
1384:   float fNum;
1385:   ifstream inputFile("x.dat", ios::in);
1386:   while(inputFile >> index >> fNum)){
1387:     cout<<index<<' '<<fNum<<endl;
1388:   }
1389:   ...
1390: }
```

КОНТРОЛЬНЫЕ ВОПРОСЫ И ЗАДАНИЯ

Вопросы и задания к главе 1

1) Объяснить назначение следующих объявлений, выявить правильные и ошибочные объявления:

```
1391: int i;
1392: const int ic;
1393: const int *pic
1394: int *const cpi;
```

2) Почему следующая инициализация ошибочна?

```
1395: char st[11] = “fundamental”
```

3) В следующем фрагменте программы имеется две ошибки индексации, найдите их:

```
1396: main(){
1397:   const int arraySize = 10;
1398:   int ia[arraySize];
1399:   for (int ix = 1; ix <= arraySize; ++ix)
1400:     ia[ix] = ix;
1401: //...
1402: }
```

4) Дано объявление переменных:

```
1403: unsigned int ui1 = 3, ui2 = 7;
```

Какой будет результат вычисления следующих выражений?

```
1404: ui1 & ui2
1405: ui1 && ui2
1406: ui1 | ui2
1407: ui1 || ui2
```

5) Составить функцию, в которой определяется, входит ли подстрока символов в другую строку символов.

6) Составить функцию, в которой определяется, являются ли два целочисленных массива одинаковыми.

7) Дана функция двоичного поиска в сортированном целочисленном массиве, требуется преобразовать ее в шаблонную функцию и составить программу, в которой выполняется поиск заданного элемента в массиве целых и вещественных чисел.

```
1408: const notFound = -1;
1409: int binSearch(int *ia, int sz, int val){
1410:   int low = 0;
1411:   int high = sz-1;
1412:   while (low <= high){
1413:     int mid = (low+high)/2;
1414:     if (val == ia[mid]) return mid;
1415:     if (val < ia[mid])
1416:       high = mid-1;
1417:     else low = mid+1;
1418:   }
1419:   return notFound;
1420: }
```

Вопросы и задания к главе 2

1) Перечислите достоинства и недостатки динамического распределения объектов и локального объявления объектов.

2) Разработайте класс связного списка символьных строк. Какие отношения возникают между объектом списка и элементами списка, между отдельными элементами списка?

3) Разработайте конструктор копирования для класса `Mouth`.

Вопросы и задания к главе 3

1) Попытайтесь сформулировать правило, определяющее, в каких случаях следует применять наследование классов, а в каких – использование классов, обеспечивающее агрегатирование оъектов.

2) Приведите примеры для каждой формы наследования из какой-либо конкретной предметной области.

3) Объясните, почему в строго типизированных языках запрещено присваивать экземпляру класса потомка значение экземпляра класса предка. В каких случаях это присваивание оказывается необходимым?

4) Приведите пример множественного наследования из реальной жизни, не связанной с вычислительной техникой.

5) Можно ли значение `NULL` считать полиморфным оъектом?

6) Объясните, почему обобщенные классы часто используются для гомогенных контейнеров (контейнеров, содержащих элементы одного типа) и не используются для гетерогенных контейнеров (контейнеры, содержащие элементы разных типов).

7) Позволяет ли использование обобщенных классов сократить объем машинного кода, получаемого после трансляции программы, почему?

Вопросы к главе 4

1) В каких случаях следует использовать статические методы?

2) Имеется ли перегрузка операторов в тех языках программирования, которые не поддерживают средства объектно-ориентированного программирования.

ЗАКЛЮЧЕНИЕ

Процесс создания сложных программных систем включает в себя различные этапы: анализ, проектирование, программирование, сопровождение. На каждом из этих этапов разработчики используют различные методы работы, различные модели и различные формальные языки, в терминах которых фиксируются решения. Значительную сложность представляют переходы от одного этапа к другому, так как именно на стыке перечисленных видов деятельности необходимо выполнить преобразование полученных решений, и на основе предыдущего этапа синтезировать модели последующего этапа.

Объектно-ориентированный подход позволяет использовать на всех этапах проектирования программной системы общий стиль работы, единую систему моделей. Это в значительной степени облегчает переходы от анализа к проектированию, от проектирования к программированию. В данном учебном пособии, посвященном этапу программирования, материал изложен таким образом, чтобы кроме специальных вопросов программирования можно было получить представление о том, как связаны проектные модели с программными моделями.

Вместе с тем в пособие включены вопросы внутренней реализации объектно-ориентированных средств языка C++. Необходимость в этом вызвана тем, что при программировании необходимо представлять границы возможностей реализации проекта в виде программы. Кроме этого, только при знании технических деталей реализации возможен быстрый поиск ошибок в программе в процессе отладки и, в конечном счете, создание достаточно надежных программ.

БИБЛИОГРАФИЧЕСКИЙ СПИСОК

1. Буч Г. Объектно-ориентированное проектирование; Пер. с англ. – М.: Конкорд, 1992. – 519 с.

2. Бадд Т. Объектно-ориентированное программирование в действии; Пер. с англ. – СПб: Питер, 1997. – 464 с.

3. Эллис М., Строуструп В. Справочное руководство по языку программирования C++ с комментариями; Пер. с англ. – М.: Мир, 1992. – 445 с.

4. Lippman S.B. C++ Primer. - 2nd ed., Addison-Wesley publishing company, 1993. – 614 p.

ОГЛАВЛЕНИЕ

Учебное издание

ДУБОВ Илья Ройдович
ВЛАСЕНКО Виктория Викторовна

ОБЪЕКТНО-ОРИЕНТИРОВАННОЕ ПРОГРАММИРОВАНИЕ
Учебное пособие

Редактор Е.А. Амирсейидова
Корректор Е.В. Афанасьева
Компьютерная верстка И.Р. Дубов

ЛР № 020275. Подписано в печать 05.03.03.
Формат 60x84/16. Бумага для множит. техники. Гарнитура Таймс.
Печать офсетная. Усл. печ. л. 3,95. Уч.-изд. л. 4,22. Тираж 100 экз.
Заказ
Редакционно-издательский комплекс
Владимирского государственного университета.
600000, Владимир, ул. Горького, 87