Министерство образования Республики Беларусь
Учреждение образования
«Белорусский государственный университет
информатики и радиоэлектроники»

Кафедра информационных технологий автоматизированных систем

М. П. Ревотюк

ОБЪЕКТНО-ОРИЕНТИРОВАННОЕ ПРОГРАММИРОВАНИЕ И ПРОЕКТИРОВАНИЕ

В 2-х частях

Часть 1

ТЕХНОЛОГИИ ОБЪЕКТНОГО ПРОГРАММИРОВАНИЯ

Рекомендовано УМО по образованию в области информатики и радиоэлектроники в качестве учебно-методического пособия для специальности 1-53 01 02 «Автоматизированные системы обработки информации»

Минск БГУИР 2014

УДК 004.42(075.8)
ББК 32.973.26-018я73
Р32

Р е ц е н з е н т ы:

кафедра прикладной информатики учреждения образования «Белорусский государственный аграрный технический университет» (протокол №11 от 06.06.2013 г.);

заведующий кафедрой информационных систем и технологий Белорусского национального технического университета, доктор технических наук, профессор А. А. Лобатый

Ревотюк, М. П.

Р32 Объектно-ориентированное программирование и проектирование. В 2 ч. Ч. 1 : Технологии объектного программирования : учеб.-метод. пособие / М. П. Ревотюк. – Минск : БГУИР, 2014. – 194 с. : ил.

ISBN 978-985-543-009-5 (ч. 1).

Представлены технологические особенности и приемы объектно-ориентированного программирования на современных версиях языка C++. Материал иллюстрируется показательными демонстрационными примерами программ.

Предназначено студентам всех форм обучения при изучении дисциплины «Объектно-ориентированное программирование и проектирование».

УДК 004.42(075.8)
ББК 32.973.26-018я73

ISBN 978-985-543-009-5 (ч. 1)
ISBN 978-985-543-010-1

СОДЕРЖАНИЕ

ВВЕДЕНИЕ

Среди многих современных языков программирования язык C++, являющийся расширением языка процедурного программирования C, выделяется ориентацией на объектно-ориентированное программирование. В настоящее время созданы многие другие языки объектно-ориентированного программирования, например, Java или C# [13], использующие концепции C++. В предлагаемом пособии изложение сути технологии объектного программирования будет вестись на примере языка C++ [9], остающегося наиболее представительным по набору возможностей, востребованным и развиваемым стандартизованным языком.

Объектно-ориентированное программирование – технология создания и использования новых типов объектов, наследующих некоторые черты ранее созданных типов объектов (объект может включать в любом сочетании данные, функции и определения типов). Определяемые пользователем типы объектов в языке C++ называют классами.

Технология объектно-ориентированного программирования явилась результатом эволюции методов программирования в направлении ярко выраженного конструктивного использования принципов структуризации, модульности и абстракции [1–9].

Ключевые понятия объектно-ориентированного программирования:

- пакетирование (encapsulation) – связывание в единое целое объектов некоторых существующих типов и функций доступа к ним с целью определения объектов нового типа;
- наследование (inheritance) – использование элементов данных, функций и собственных типов ранее определенных объектов при образовании иерархии производных объектов;
- полиморфизм (polymorphizm) – возможность ассоциации некоторого имени с множеством уникальных для каждого уровня иерархии производных объектов понятий.

Рассмотрим предварительно идеи поддержки перечисленных понятий, предполагая освоенным процедурный стиль программирования, например, на языке C.

В процедурном программировании [8] известны два типа модулей:

1) функция – программный модуль, связывающий входные параметры и результат;

2) файл – совокупность функций и глобальных переменных, где глобальные переменные определяют контекст состояния процесса использования функций.

Использование функций – старый проверенный способ надежного программирования, весьма привлекательный для задач математического характера или допускающих формализацию.

Пример использования модулей-функций:

```
#include <stdio.h>
#include <window.h>
#include <conio.h>
#include <time.h>

/* Отображение текущего времени */
void show_time() {
 struct tm *newtime;
 long ltime;
 time(&ltime);
 newtime = gmtime(&ltime);
 printf("The current time is: %2d:%02d:%02d.%02d\n",
         newtime->tm_hour, newtime->tm_min, newtime->tm_sec);
}
void main() {
 show_time();
 Sleep(1000);
 show_time();
}
```

Достоинство модулей-функций – описание функции и известные значения аргументов достаточны для прогнозирования поведения программы. Исходный текст программы вполне достаточен для объяснения ее работы.

Недостаток модулей-функций – каждая операция вызова функции требует полного представления задачи списком параметров, разработчик должен владеть алгоритмом решения всей задачи.

Однако наличие множества библиотек функций в разных системах программирования подтверждает жизнеспособность рассмотренного метода.

Реальные программы чаще соответствуют модулям-файлам, т. к. даже многие стандартные функции используют глобальные переменные.

Пример использования модулей-файлов:

```
/* ************** Файл my_time.c ***************** */
/* Подсистема отображения текущего времени */
#include <stdio.h>
#include <windows.h>
#include <conio.h>
      void _show_time() { show_time(); }
      void set_time(int x, int y, int z) { x=h; y=m; z=s; }
}

/* ************** Файл my_test.c ***************** */
/* Использование подсистемы отображения файлов */
#include "my_time.c"
      void main() {
      show_time();
      set_time();
      show_time();
      int x, y, z;
      x++, y++; z++;
```

```
        set_time_xy(x, y, z);
        show_time();
}
```

Достоинства модулей-файлов – сокращение записи и реализации операции вызова функции, работа с понятием контекста, разделение труда программистов.

Проблемы использования модулей-файлов – диагностика ошибок требует анализа всех глобальных переменных, а также слежения за операциями вызова функций, разработчик должен заниматься синхронизацией вызова функций. Исходный текст недостаточен для объяснения работы программы, требуется трассировка. Для этого система программирования должна включать средства отладки разного уровня детализации процесса исполнения программы.

Общепризнанный факт практики современного программирования – эволюционный характер процесса создания и поддержки жизненного цикла программ [11–12]. Очевидно, что технологии программирования на основе модулей-функций и модулей-файлов нуждаются в совершенствовании. Линейность представления исходного текста программы, который может быть весьма громоздким, затрудняет анализ ее структуры читателем (программистом) с целью развития программы независимыми разработчиками или даже автором спустя некоторое время после создания работающей версии программы.

Поддержка эволюционного характера процесса создания и развития программ обеспечивается включением в язык средств реализации ключевых принципов объектно-ориентированного программирования – пакетирования, наследования и полиморфизма. В языке C++ такие средства построены на расширении понятия типа, а более конкретно – понятий структуры (struct) и объединения (union), определенных стандартом языка C. Так как такие структуры и объединения предназначены для представления произвольных данных, то иногда к трем названным принципам объектно-ориентированного программирования добавляют принцип абстракции данных.

Реализация пакетирования в C++:

```
/* ************** Файл my_time.h ***************** */

#include <stdio.h>
#include <window.h>
#include <conio.h>
#include <time.h>

struct my_time {
 int x,y;

 void show_time() {
  show_time_xy(x,y);
 }

 void set_time_xy(int new_x, int new_y) {
  x=new_x;
```

```
  y=new_y;
 }

 void get_time_xy(int *px, int *py) {
  *px=x;
  *py=y;
 }

 my_time() {
  x=y=0;
 }
};

/* ************** Файл my_test.c ***************** */
#include "my_time.h"

void main() {
 my_time t;
 t.show_time();
 t.set_time_xy(10,10);
 t.show_time();
 int x, y;
 t.get_time_xy(&x,&y);
 x++, y++;
 t.set_time_xy(x,y);
 t.show_time();
}
```

Данные и функции описаны в одной языковой конструкции, компилятор может контролировать их использование. Понятие структуры языка С расширено возможностью включения в определение структуры декларации функций, работающих с ее элементами. Использование таких функций реализуется по стандартным правилам использования элементов структуры.

Реализация наследования в С++:

```
/* ************** Файл your_time.h ***************** */
#include "my_time.h"

/* Класс отображения времени с сохранением текущей позиции */
struct your_time: public my_time {
 void safe_show_time() {
 int x=wherex();
  int y=wherey();
  show_time();
  gotoxy(x,y);
 }
};

void main() {
 your_time t;
 t.show_time();
 t.set_time_xy(10,10);
 t.show_time();
 int x, y;
 t.get_time_xy(&x,&y);
```

```
 x++, y++;
 t.set_time_xy(x,y);
 t.show_time();
}
```

Здесь ранее написанный исходный текст (базовый класс) остается без изменений, старые версии работают, программу можно развивать далее в сторону расширения. Существенно, что авторское право создателя базового класса не нарушается, а количество расширенных версий не ограничено.

Реализация полиморфизма в C++:

```
#include "your_time.h"

// Организация периодического отображения времени

struct common_show_time: public your_time {
  virtual void common_show() { // Отображение текущего времени
   safe_show_time();
  }
  void time_at_line(int n) { // Периодическое отображение времени
   for (int i=0; i < n; i++)
    common_show();
  }
};

// Привязка отображения времени к реальным часам

struct show_real_time: public common_show_time {
  void common_show() {  // Имя common_show уже использовалось...
   safe_show_time();
   sleep(1);
  }
};

void main() {
 show_real_time t;
 t.time_at_line();
 common_show_time v;
 v.time_at_line();
}
```

Здесь ранее написанный исходный текст базового класса остается без изменений, но программу можно развивать далее в сторону уточнения и детализации. Однако возможность полиморфизма должна быть предусмотрена при создании базового класса.

В наибольшей степени технология объектно-ориентированного программирования проявляет свои преимущества при создании, сопровождении и развитии больших (с размером исходного текста порядка 10 тысяч операторов) программных комплексов. Достигаемая с ее применением концептуальная стройность и прозрачность программ хорошо согласуется с человеческим фактором, а накладные расходы на освоение и поддержку составляют незначительную долю.

1. РАЗВИТИЕ ЭЛЕМЕНТОВ ЯЗЫКА С В ЯЗЫКЕ С++

1.1. Соотношение языков С и С++

Язык С++ появился в 1983 г. на основе предложенных и оформленных Бьерном Страуструпом улучшений языка С с использованием наиболее полезных концепций языков SIMULA-67 и ALGOL-68. В 1998 г. был ратифицирован международный стандарт ISO/IEC 14882:1998 «Standard for the C++ Programming Language», а после технических исправлений в 2003 г. – версия этого стандарта ISO/IEC 14882:2003.

Язык С++ – компилируемый язык программирования общего назначения со строго статической типизацией данных и адресной арифметикой. Поддерживает процедурную, обобщенную и функциональную парадигмы программирования, но наибольшее внимание уделено поддержке технологии объектно-ориентированного программирования. Реализация принципов пакетирования, наследования и полиморфизма такой технологии включает:

- формирование при решении прикладных задач понятия классов операционных объектов;
- описание типов данных, характеризующих объекты класса и операции над объектами;
- программирование алгоритмов решения задачи в терминах операций над базовыми и определенными пользователем объектами в рамках традиционных операционных возможностей языка С.

Язык С++ не содержит средств обработки прерываний (обработка исключительных ситуаций сейчас обсуждается [9]), операторов параллельного программирования, ввода – вывода. Подобные средства реализуются библиотечными функциями и классами. Вместе с тем поддержка понятия определяемого пользователем класса объектов ставит язык С++ наравне с высокоуровневыми языками программирования.

Язык С++ полностью совместим с языком С. Формально можно считать язык С подмножеством языка С++. Программы на языке С практически без изменений пригодны для обработки системами программирования С++, но не наоборот.

Перечень существенных лексических расширений стандартного языка С:

- возможность определения новых типов операционных объектов (классов) посредством определения структур данных представления объекта и набора функций манипулирования объектами, причем данные определения объекта могут быть скрытыми;
- введение контроля набора и типов аргументов функций, преобразования типов и задания умалчиваемых значений аргументов;
- переопределение функций;

• переопределение операций;

• возможность встраивания тела функции в точку ее вызова подобно эквивалентному макрорасширению;

• определение объектов-констант;

• определение объектов типа ссылок;

• управление распределением памяти операторами new и delete.

Далее в настоящем разделе последовательно рассматриваются элементы языка C++, являющиеся расширением подобных в языке C.

1.2. Комментарии в C++

Комментарии – элемент пояснения аспектов реализации программы, которые не могут быть непосредственно отражены операторами языка программирования.

Язык C++ допускает два способа записи комментариев в исходном тексте программы:

• традиционное для языка C использование пар символов «/*» и «*/» для обрамления игнорируемой компилятором последовательности символов (по умолчанию комментарии такого типа не могут быть вложенными);

• однострочные комментарии, начинающиеся в любом месте строки с пары символов «//» и заканчивающиеся в конце строки.

Примеры записи комментариев в стиле языка C:

```
/* Декларация глобальных объектов */
int status; /* Состояние процесса */
/* ПРОЦЕДУРА ОБРАБОТКИ ОШИБОК */
```

Примеры записи однострочных комментариев:

```
// Декларация глобальных объектов
int status; // Состояние процесса
// ПРОЦЕДУРА ОБРАБОТКИ ОШИБОК
```

Первый способ удобен для написания многострочных комментариев и быстрой корректировки исходного текста на этапе отладки, а второй способ предпочтительнее для коротких комментариев.

Пара символов «//» может применяться для комментирования текста с символами «/*» и «*/» и, наоборот, символы «/*» и «*/» могут подавлять действие символов «//».

1.3. Идентификаторы объектов

Идентификатор как последовательность символов в исходном тексте программы на языке C++ используется для именования следующих сущностей:

- объект или переменная;
- структурированный тип – класс (class), структура (struct) или объединение (union);
- перечисление (enum);
- элементы структурированного типа или перечисления;
- функции, включая функции-элементы классов;
- новый тип, определенный оператором typedef;
- метка оператора;
- макроопределение и макрокоманда;
- параметр макроопределения и макрокоманды.

Идентификаторы перечисленных объектов в программах на языках С и С++ могут включать только символы из множества {a...z, A...Z, 0...9,_} (здесь фигурные скобки заключают элементы множества). Регистр символов имеет значение. Первый символ идентификатора должен принадлежать множеству {a...z, A...Z,_}.

Длина идентификатора в последнее время практически не ограничена. Например, в системе программирования Microsoft Visual C++ 6.0 разрешено использовать в идентификаторе до 248 символов. Идентификаторы не должны совпадать с ключевыми словами языка С++.

Набор ключевых слов определяется реализацией транслятора.

Список основных ключевых слов языка С++:

```
asm     continue    float    new        sizeof    typedef
auto    default     for      operator   static    void
break   delete      friend   private    struct    volatile
case    do          goto     protected  switch    virtual
catch   double      if       public     template  unsigned
char    else        inline   register   this      union
class   enum        int      return     throw     while
const   extern      long     short      try
```

1.4. Область действия (видимости) объекта

Имя любого объекта программы на языке С имеет атрибут локальной или глобальной области действия описания объекта. Такой атрибут определяется неявно местоположением оператора описания объекта.

Рассмотрим фрагмент исходного текста программы на языке С:

```
int error_n=0; // глобальная переменная
void f1() {
   int x; // локальная переменная
   // ...
}

struct complex { // глобальное описание типа данных
   float re,im;
};
```

```
void f2(int x) {
   int y;
   complex z; // локальная переменная типа struct complex
   // ...
   if (!scanf(" %d",&y))
      error_n++; // изменение глобальной переменной
   z.re=(float)y, z.im=0;
   // ...
}
```

Можно утверждать, что границей разделения областей действия имен объектов в языке С является блок определения функции. Имена локальных объектов, описанных в функции до первого исполнимого оператора, действуют в пределах тела функции. Локальными являются и аргументы функции. Имена глобальных объектов, описанных вне функции, действуют от точки описания до конца файла исходного текста программы.

В языке С++ границей разделения областей действия имен является любой операторный блок, ограниченный символами фигурных скобок «{» и «}». Имена объектов, определенные вне блока, остаются в блоке доступными, но любое ранее определенное имя можно переопределить для ссылки на другой объект. После выхода из блока переопределенное имя вновь обретает свой прежний смысл. Другими словами, локальное имя объекта скрывает старое определение объекта и, по существу, образуется иерархия локальных областей действия.

```
// ПРИМЕРЫ СКРЫТИЯ ИМЕН ОБЪЕКТОВ
int x; // глобальное имя x
void f() {
   int x; // локальное имя x скрывает глобальное имя x
   x = 1; // присваивание значения локальному x
   {
       int x; // скрытие первого локального имени x
       x = 2; // присваивание значения второму локальному x
   }
   x = 3; // присваивание значения первому локальному x
}
int* p=&x; // определение адреса глобального x
```

Скрытие имен удобно и практически неизбежно при написании больших программ. Однако при чтении текста программы можно не заметить, что некоторое имя скрыто, а возникающие вследствие этого ошибки очень трудно обнаружить. Скрытие имен рекомендуется минимизировать. Очевидно, что использование для общих переменных во вложенных блоках коротких и невыразительных имен вроде i или x провоцирует будущие неприятности. Возможность использования скрытого локального имени язык С++ не предоставляет, но скрытое **глобальное** имя становится доступным посредством применения унарной операции привязки имени, обозначаемой парой символов «::» (часто такую операцию называют операцией разрешения видимости).

```
// ПРИМЕР ИСПОЛЬЗОВАНИЯ ОПЕРАЦИИ РАЗРЕШЕНИЯ ВИДИМОСТИ
int x; // глобальная переменная x
void f() {
   int x = 1; // скрытие глобальной переменной x
   ::x = 2; // обращение к глобальной переменной x
   // ...
}
```

Операция привязки имени обладает максимальным приоритетом.

Обсудим особенности языка C++, порождаемые расширением понятия области действия описания объектов.

Область видимости имени начинается в точке описания. Последнее означает, что имя можно использовать даже в выражении инициализации именуемого объекта. Например:

```
#include <stdio.h>

double x=0;

void main() {
   printf("\n %d %ld",sizeof(x),x);
   static int x=sizeof(x);
   {
     printf("\n %d",x);
     char x=sizeof(x);
     printf("\n %d",x);
     {
        printf("\n %d",x);
        char x=x^x;
        printf("\n %d",x);
     }
   }
 }
```

Результат работы программы:

```
8 0
2
1
1
0
```

Стандарт языка C предписывает размещение операторов декларации локальных объектов в блоке функции до первого исполняемого оператора, а выражения инициализации объектов могут включать лишь операнды-константы.

Язык C++ позволяет описывать объекты в любом месте исходного текста, но вложенность определений функций не допускается. Выражения инициализации статических и нестатических объектов могут содержать любые операнды и операции. Статические объекты языка C инициализируются только константными выражениями.

Очевидно, что одно и то же имя в пределах блока практически может использоваться для обращения к двум различным объектам:

```
int x = 123; // глобальное x
void f() {
 int y = (x+1)%13; // глобальное x
 x& = 0x00ff;
 // ...
 int x=456; // переопределение имени x
 y = x; // локальное x
}
```

Здесь переменная y инициализируется значением глобального x = 123, а затем ей присваивается значение локальной переменной x = 456. Подобный стиль именования переменных затрудняет чтение программы.

Параметры функции считаются описанными в самом внешнем блоке функции. Например:

```
void f(int x) {
 int x; // ошибка, обнаруживаемая транслятором
 // ...
}
```

Здесь имя x определено ошибочно дважды в одной и той же области действия.

Возможность инициализации глобальных переменных любыми выражениями не связана с типом исполняемого модуля. Как загрузочный модуль, так и динамически подключаемая библиотека могут содержать такие операторы декларации.

Область действия имен в C++ может ассоциироваться с пространством имен (подразд. 1.14). Пространство имен (namespace) в программировании – область определения объектов (переменных, типов, констант и др.). В языке C++ пространство ограничивается созданием операторного блока. Используется для исключения конфликтов с другими именами вне заданного блока.

1.5. Атрибуты типа const и volatile

В языке C имеются следующие виды объектов-констант:

- самоопределенные значения символьных констант, целых констант и констант с плавающей точкой;
- элементы перечислений;
- имена массивов (векторов);
- имена функций.

В языке C++ ключевое слово const, добавленное к описанию объекта, объявляет этот объект константой, а не переменной:

```
const int status = 145;          // скалярная константа
const int values[] = {1,2,3,4};  // массив констант
```

По определению константе ничего нельзя присвоить, поэтому она должна быть инициализирована. Атрибут const гарантирует, что значение объекта в области его видимости не изменится:

```
status=123; // ошибка – переопределение значения константы
status++;   // ошибка – изменение значения константы
```

Слово const модифицирует тип объекта в смысле использования объекта, но не меняет способ размещения объекта в памяти. Контроль за сохранностью объекта возлагается на компилятор. Атрибут const означает наличие запрета на возможные попытки изменения объекта области его видимости (в языке С константные объекты размещались в статической памяти). Такой атрибут может иметь, в частности, и возвращаемое значение функции:

```
const char* item();
```

В этом случае становятся недопустимыми выражения вида

```
item()[i]='=';
(*item())++;
```

Здесь объект-константа не является статически определенным.

Последний пример показывает возможность работы с объектами без имени, что характерно для **косвенной** адресации.

В левой части операций присваивания может находиться только прямая или косвенная ссылка на область объекта-переменной (объекты такого вида иногда называют lvalue). Атрибут const предписывает невозможность использования объекта в левой части оператора присваивания.

Очевидно, что описание указателей имеет отношение к двум объектам – собственно указателю и адресуемому им объекту. Префикс описания указателя const объявляет константой только адресуемый объект:

```
const char* p = "0123456789";  // указатель на константу
p[2] = 'a';                    // ошибка
p = "ghjk";                    // изменение указателя
```

Описание константного указателя производится в постфиксной форме:

```
char *const q = "abcdef";  // константный указатель
q[3] = '_';                // изменение поля данных
q = "ghjk";                // ошибка
```

При необходимости объявления константами обоих связанных с указателем объектов модификатор const применяется дважды:

```
const char *const
 r="0123456789abcdef"; // константный указатель на константу
r[3] = 'a';           // ошибка
r = "ghjk";           // ошибка
```

Указателю на константу можно присваивать адрес переменной ввиду безобидности последствий. Однако присвоить адрес константы указателю на переменную нельзя, т. к. это позволило бы изменить значение объекта:

```
int x = 1;
const c = 2; // Здесь атрибут типа в описании опущен,
const *p1 = &c; // т. к. по умолчанию предполагается
const *p2 = &x; // тип int
int *p3 = &c; // ошибка
*p3 = 7; // ошибка
```

Атрибут const полезно использовать для указателей – параметров функции – с целью установки запрета на изменение функцией адресуемого объекта:

```
char* strcpy(char* p, const char* q);
```

(здесь библиотечная функция копирования строк не может менять исходную строку).

Введение атрибута const наряду с автоматизацией контроля за использованием объектов позволяет компилятору иногда оптимизировать программу с учетом процедуры инициализации и механизмов доступа к памяти. Изменение значения константы возможно посредством использования операции приведения типа

```
const int x = 123;    // инициализация константы
*(int *)&x = 256;     // принудительное изменение константы
```

Таким образом, объекты-константы не защищены от модификации, но эта модификация явно определяется программистом в исходном тексте.

Атрибут volatile уведомляет компилятор о том, что объект может использоваться одновременно многими процессами. Например, если переменная count разделяется фоновой программой и программой обработки прерываний, то ее описание может иметь вид

```
volatile int count = 0; // счетчик прерываний

void main () {
 while (count < 10) {
  //…
 }
}
```

Значения переменных с атрибутом volatile при вычислении выражений каждый раз будут выбираться из памяти и сохраняться немедленно после присваивания. Хранение копий значений таких переменных в регистрах на этапе оптимизации кода транслятором не предусматривается.

Сказанное ранее относительно описания указателей с модификатором const можно отнести и к модификатору volatile. Например, оператор

```
volatile int * volatile varptr;
```

объявляет доступность для разделяемого доступа указатель и адресуемых им данных.

1.6. Ссылки (обращение по адресу)

Единственным способом связи параметров и аргументов функций в языке С является передача значений параметров, а не их адресов. Это делает невозможным изменение параметров в вызывающей функции. При необходимости изменения функцией некоторых объектов программист вынужден использовать указатели этих объектов. В языке С++ введен дополнительный модификатор типа, обозначаемый символом «&», позволяющий ссылаться на объект по его адресу без применения указателя.

Если type – некоторый тип объекта, то модифицированный тип type& называют **ссылкой** на объекты типа type.

Ссылка представляет собою константный указатель, определяемый на этапе компиляции и построения загрузочного модуля программы. Отсюда следует, что никакие действия над ссылками не могут быть явно запрограммированы. Использование ссылок в выражениях означает обращение к адресуемым этими ссылками объектам.

Можно выделить два вида применений ссылочных типов:

- организация передачи значений параметров и возвращаемых результатов функций;
- создание псевдонимов переменных и констант.

Сначала рассмотрим использование ссылок для построения функций, изменяющих значение параметра.

Пусть записан фрагмент программы:

```
int x = 1;
void incr(int& a) { a++; }
// ...
incr(x);                          // x = 2
```

Здесь параметр a объявлен ссылкой на объект типа int, а оператор a++ предписывает инкремент этого объекта.

Представим эту же программу без использования понятия ссылки. С точки зрения удобочитаемости исходного текста программы предпочтительнее явно возвращать значение функции:

```
int x = 2;
int next(int p) { return p+1; }
// ...
x = next(x);                      // x = 3
```

Аналогичный результат достигается при использовании в качестве параметра указателя на изменяемую переменную:

```
int x = 3;
void inc(int* p) { (*p)++; }
// ...
inc(&x);                              // x = 4
```

Последние два варианта программы теряют свою привлекательность, если параметр функции является массивом или структурой.

Ссылки удобно применять для определения функций, которые могут использоваться как в левой, так и в правой части операции присваивания. Такая возможность реализуется посредством возврата функцией результата ссылочного типа.

Например, определим ассоциативный массив, где строка символов имеет некоторое ассоциированное с ней целое значение. Элемент массива представим структурой

```
struct pair {
   struct pair *next; // Слово struct здесь необязательно
   char* name;
   int val;
};
```

Пусть поиск целочисленного значения по заданному значению строки осуществляется функцией value(), которая отыскивает подходящий элемент, а в случае неудачного поиска расширяет текущий массив. Пример реализации функции поиска:

```
static struct pair *head=NULL;
int& value(char* p) {
   for (struct pair *x=head; x; x=x->next)
       if (!strcmp(x->name,p)) return x->val;
       x=malloc(sizeof(struct pair));
       x->next=head, head=x;
       x->name=malloc(strlen(p)+1);
       strcpy(x->name,p);
       x->val = 0;
       return x->val;
}
```

Представим программу подсчета количества повторений, введенных с клавиатуры слов. Введенной строке здесь соответствует количество повторений ее содержимого.

```
void main() {
   char buf[80]; // Буфер ввода строк
/* Создание ассоциативного массива */
   while (fgets(buf,sizeof(buf),cin)) value(buf)++;
/* Вывод ассоциативного массива */
   for (struct pair *x=head; x; x=x->next)
    printf("\n %s: %d",x->name,x->val);
}
```

Результатом работы программы для исходной последовательности (aa, bb, bb, aa, cc, aa, bb, aa, aa) является

```
cc: 1
bb: 3
aa: 5
```

Рассмотренный пример можно реализовать на языке С с использованием, например, указателей:

```
int *value(char *); // другое определение функции
//...
(*value(buf))++;    // инкремент количества повторений
```

Таким образом, использование ссылок позволяет отказаться от операций над указателями и повысить лаконичность записи исходного текста программы. Передача параметров и возвращаемых результатов функций по ссылке оказывается весьма эффективным приемом исключения копирования больших по размеру структур. Если вызываемая функция не меняет значение параметра, передаваемого по ссылке, то легко запретить его модификацию добавлением атрибута const, например:

```
f(const Large_Object& nonmodified_parameter) { /* ... */}
```

Рассмотрим применение ссылок для назначения псевдонима или другого имени объекта.

```
int  i = 1;
int& r = i;    // r и i ссылаются на одно поле типа int
int  x = r;    // x=1

r = 2; // i=2
r++; // Значение i увеличивается на 1
```

Здесь оператор r++ не увеличивает ссылку, а операция инкремента применяется к адресуемым ею данным типа int, которыми оказывается поле переменной i.

Ссылка, определяющая псевдоним, обязательно должна быть инициализирована, т. е. должен существовать объект, которому назначается новое имя. Инициализация ссылки необязательно требует обращения к объекту, к которому допустима операция определения адреса. Более того, тип данных такого объекта может не совпадать с типом данных ссылки. В таких случаях значением ссылки становится адрес временной переменной, куда после преобразования типа помещается инициализирующее значение. Например, описание псевдонима константы

```
float& s = 1;
```

интерпретируется компилятором следующим образом:

```
float* s;          // представление ссылки указателем
float tmp;         // объявление временной переменной
tmp = (float)1;    // инициализация временной переменной
s = &tmp;          // инициализация ссылки
```

Использование временной переменной исключает возможность изменения области объекта, который использовался для инициализации ссылки:

```
// ПРИМЕР НЕЖЕЛАТЕЛЬНЫХ ПОСЛЕДСТВИЙ ПРИМЕНЕНИЯ ПСЕВДОНИМА
int ix = 1;
char& cx = i; // Типы cx и i не совпадают, поэтому будет
     // создана временная переменная типа char
     // с начальным значением (char)ix и имеющая
     // псевдоним cx
cx=2; // Значение ix не меняется, меняется значение
     // временной переменной

// ПРИМЕР НЕЖЕЛАТЕЛЬНЫХ ОСОБЕННОСТЕЙ СВЯЗИ ПАРАМЕТРОВ ПО ССЫЛКЕ
void f(int& ix) { ix=2005; }
// ...
char cx;
// ...
f(cx); // Типы параметра и аргумента не совпадают,
       // поэтому будет сформирована последовательность
 // {int temp=(int)cx;
 // f(temp);
 //}
 // В результате переменная cx останется
 // без изменения
```

Последние примеры показывают, что избежать недоразумений можно лишь тщательным учетом последствий интерпретации типа объекта транслятором.

1.7. Набор и приоритет операций в языке C++

Введем обозначения:

cid – идентификатор структуры, объединения или класса;

id – идентификатор объекта в памяти;

pe – выражение-указатель;

e – выражение;

ie – целочисленное выражение;

el – список выражений (элементы списка разделены запятыми);

t – идентификатор типа объекта;

lv – выражение, обозначающее непостоянный объект, пригодный для использования в качестве левого операнда операции присваивания (lvalue).

Сведения о составе и приоритете операций языка C++ представлены в табл. 1.

Таблица 1

Список операций языка С++

Обозначение	Наименование операции	Схема операции
::	Привязка имени элемента класса	cid::id
::	Привязка глобального имени	::id
.	Прямой выбор элемента класса	cid.id
->	Выбор элемента по указателю объекта класса	pe->id
[]	Индексация	pe[e]
()	Вызов функции	e(el)
()	Построение значения	t(el)
sizeof	Размер в байтах поля объекта или результата вычисления выражения	sizeof e
sizeof	Размер в байтах поля объекта указанного типа	sizeof(t)
++	Приращение после использования	lv++
++	Приращение до использования	++lv
--	Уменьшение после использования	lv--
--	Уменьшение до использования	--lv
~	Дополнение (побитовое НЕ)	~ie
!	Логическое НЕ	!e
-	Унарный минус	-e
+	Унарный плюс	+e
&	Адрес объекта	&lv
*	Косвенный выбор по указателю	*pe
new	Создание объекта	new t
delete	Уничтожение объекта	delete pe
delete[]	Уничтожение массива объектов	delete[] pe
()	Приведение (преобразование) типа	(t)e
*	Косвенный выбор элемента класса	cid.*pe
->*	Косвенный выбор элемента класса для косвенно адресуемого объекта	pe->*pe
*	Умножение	e1*e2
/	Деление	e1/e2
%	Деление по модулю (остаток)	ie1%ie2
+	Сложение	e1+e2
-	Вычитание	e1-e2
<<	Сдвиг влево	ie1<<ie2
>>	Сдвиг вправо	ie1>>ie2
<	Меньше	e1<e2
<=	Меньше или равно	e1<=e2
>	Больше	e1>e2
>=	Больше или равно	e1>=e2
==	Равно	e1==e2
!=	Не равно	e1!=e2
&	Побитовое И	e1&e2
^	Побитовое исключающее ИЛИ	ie1^ie2
|	Побитовое включающее ИЛИ	ie1|ie2
&&	Логическое И	e1&&e2
||	Логическое ИЛИ	e1||e2

Окончание табл. 1

Обозначение	Наименование операции	Схема операции
? :	Условное вычисление	e1? e2:e3
=	Простое присваивание	lv=e
*=	Умножение и присваивание	lv*=e
/=	Деление и присваивание	lv/=e
%=	Деление по модулю и присваивание	lv%=ie
+=	Сложение и присваивание	lv+=e
-=	Вычитание и присваивание	lv-=e
<<=	Сдвиг влево и присваивание	lv<<=ie
>>=	Сдвиг вправо и присваивание	lv>>=ie
&=	Побитовое И и присваивание	lv&=ie
|=	Побитовое включающее ИЛИ и присваивание	lv|=ie
^=	Побитовое исключающее ИЛИ и присваивание	lv^=ie
,	Следование	e1, e2

В каждой отчерченной части таблицы находится группа операций с одинаковым приоритетом. Приоритет операций нижерасположенных групп меньше вышерасположенных. Порядок выполнения операций можно регулировать круглыми скобками. Для объектов, определенных пользователем, содержание операции устанавливается функцией с атрибутом operator.

Можно заметить, что набор операций языка С в языке С++ дополнен операциями «::», «.*», «->*», new и delete. Смысл таких операций рассматривается далее в подразд. 2.1–2.3.

1.8. Операция преобразования типа

Операции над объектами разных типов порождают вопрос корректности и типа результата. В языках С и С++ допустимы операции над любыми комбинациями арифметических типов (int, short, long, unsigned int, unsigned short, unsigned long, float, double). Символы (char, unsigned char) преобразуются к типу int. Результат арифметической операции в математическом отношении будет корректным в рамках конкретных аппаратных возможностей. Например:

```
unsigned char x=150, y=128;
unsigned char z=x+y;          // z!=150+128
int  a=x+y;                   // a==150+128
```

Корректными и прогнозируемыми являются операции сложения и вычитания над парами операндов (указатель_объекта_типа_X – арифметическое_значение), а также вычитания указателей объектов одного типа. Результат вычитания в этом случае является целым числом.

Язык С предоставляет пользователю операцией приведения типа (тип)X явно указать требуемый вид преобразования объекта X:

```
#include <stdio.h>
#include <stdlib.h>

// ПРОЦЕДУРА ПЕЧАТИ 10 СЛУЧАЙНЫХ ЧИСЕЛ В ДИАПАЗОНЕ 0...1

void main(void) {
   static n=32767;
   for (int i=0; i<10; i++)
       printf("\n %f",(float)random(n)/(float)n);
}
```

Потребность в операции приведения возникает в случае появления неопределенности типа операнда и при необходимости обхода запретов языка на комбинации операндов. Неопределенность типа операнда возникает, например, при передаче значения аргумента функции при отсутствии в ее описании описания списка параметров. Компилятор предупреждает программиста о подобных ситуациях, но иногда неопределенность порождена отсутствием достаточной информации. Рассмотрим пример программы с неопределенностью связи аргумент – параметр.

```
#include <stdio.h>
void main() {
      float f=125.0;
      printf("\n??? %f %d %x %f",f,f,f,f);
      printf("\n*** %f %d %x %f",f,(int)f,(int)f,f);
}
```

Результаты работы программы (Borland C++):

```
??? 125.000000 0 405f4000 125.000000
*** 125.000000 125 7d 125.000000
```

Источником ошибок при вычислении выражения может быть неправильное представление промежуточных результатов.

Пусть необходимо установить способ представления в памяти конкретной вычислительной системы чисел типа long:

```
#include <stdio.h>
void main() {
   long lx=0;
   for (int i=0; i<sizeof(lx); i++)
       lx|=((long)i)<<(i<<3);
   unsigned char *x=(unsigned char *)&lx;

   printf("\n %d, %0*lx, ", sizeof(lx),sizeof(lx)<<1,lx);
   for (i=0; i<sizeof(lx); i++)
      printf(" %02x",x[i]);
}
```

Результат работы программы на ПЭВМ класса IBM PC XT/AT:

```
Размер 4, значение 03020100, представление 00 01 02 03
```

Примеры запрещенных видов операции присваивания:

```
указатель=арифметическое_значение;
арифметическая_переменная=указатель;
указатель_объекта_типа_1=указатель_объекта_типа_2,
если (тип_1!=тип2)&&(тип_1!=void)&&(тип_2!=void).
```

Многие задачи системного и прикладного программирования вынуждают использовать перечисленные операции. При этом приходится непосредственно сталкиваться с особенностями реализации вычислительной среды, а программы легко теряют свойство мобильности.

В языке С++ допустимы два способа записи операции явного преобразования типа:

а) традиционная для языка С запись операции приведения типа в форме (*тип*)*выражение*, например:

```
(float)i – приведение простой переменной,
(long)(x+y) – приведение значения выражения,
(char *)&x – приведение значения адреса;
(int *)0x100 – приведение значения константы;
```

б) функциональная запись операции приведения типа в форме *тип*(*выражение*), где «тип» должен обозначаться простым именем, например:

```
float(i),     // int i;
long(x+y),    // int x,y;
double(125);  // int z=125;
```

Ограничение на имя типа легко преодолеть определением нового имени типа, используя оператор typedef.

Например, оператор декларации

```
char *p=(char *)0777; // традиционное приведение
```

в функциональной форме записи операции приведения типа можно представить следующим образом:

```
typedef char * string; // идентификация типа char *
                       // новым именем string
char* p = string(0777); // функциональное приведение к типу
                        // string либо обращение к функции
                        // string?
```

Функциональная запись улучшает читаемость текста программы. Например, рассмотрим эквивалентные выражения

```
marker pn = base(xn->tp)->buf;    // функциональная запись
marker pn = ((base)xn->tp)->buf; // традиционная запись
```

Операция «->» имеет больший приоритет, чем приведение, поэтому последнее выражение интерпретируется как

```
marker pn = ((base)(xn->tp))->buf; // здесь 3 пары скобок!
```

Функциональный стиль записи операции приведения особенно удобен в C++ для определяемых пользователем типов (классов). Операция приведения в этом случае реализуется процедурой создания объекта – конструктором (имя процедуры совпадает с именем типа, а набор параметров характеризует исходное состояние объекта). Явное приведение типов указателей позволяет получить адрес объекта любого типа:

```
any_type* p = (any_type*)&some_object;
```

Известное значение указателя p позволяет работать с некоторым объектом some_object как объектом типа any_type (см. пример программы вывода представления поля типа long). Явное преобразование типов рекомендуется использовать лишь в крайних случаях. Появление неожиданных результатов операций во внешне правильных выражениях и исходных данных – сигнал для анализа и коррекции хода преобразований.

1.9. Объявление и переопределение функций

В классическом языке C при описании функции достаточно объявить лишь тип возвращаемого значения. Язык C++ требует полной определенности при объявлении функций. Объявление функции задается в виде ее описания и/или определения.

Описание функции задает имя функции, тип возвращаемого функцией значения (если такое есть), а также число и типы используемых при вызове параметров.

Примеры описания функций:

```
extern double sqrt(double);
extern elem* next_elem();
extern char* strcpy(char* to, const char* from);
int printf(char *,...);
extern void exit(int);
```

В описании функции для удобства чтения можно приводить имена аргументов, но компилятор их игнорирует. Описание иногда называют **прототипом** функции, подчеркивая его роль при конструировании компилятором элементарных операций обращения к функции (преобразование фактических параметров, формирование списка их значений, вызов функции по соответствующему имени адресу).

Определение функции – полное описание функции, в котором представлено тело функции.

Пример описания и определения функции:

```
extern void swap(int *, int *); // описание функции
void swap(int *x, int *y) { // определение функции
   int t=*x;
   *x=*y, *y=t;
}
```

(имена аргументов в определении функции обязательны).

Определение функции в исходном тексте программы может отсутствовать, но наличие до первого использования имени функции ее описания в этом случае обязательно. Появление в исходном тексте определения функции делает излишним ее описание в остатке текста до конца файла. Отсутствие определения в файле программы свидетельствует о том, что будет использован объектный модуль, получаемый на основе другого файла или из библиотеки. Описания функций рекомендуется помещать в заголовочные файлы.

В языке C++, в отличие от языка C, функции идентифицируются строкой, элементы которой прямо или в кодированном виде представляют:

- имя функции;
- если функция является элементом класса, то имя класса, а также все имена включающих классов;
- имя пространства имен (namespace);
- типы параметров функции;
- схема операции вызова функции;
- тип возвращаемого результата.

Правила формирования или декорирования имени устанавливаются при разработке компилятора и компоновщика и могут быть системно-зависимыми.

Программист не может менять правила декорирования, однако следствием их учета в языке C++ разрешено использовать в одной программе функции с одинаковыми именами, но различающиеся списками аргументов по количеству и/или типу. Говорят, что такие функции являются **переопределенными** (иногда встречаются термины «перезагрузка» или «повторная загрузка» функций).

Переопределенные функции весьма практичны для программирования одинаковых в некотором смысле действий над объектами разных типов.

Пример использования переопределенных функций:

```
#include <stdio.h>

int sum(int x, int y) { // Вариант 1
   printf("\nInt - Int :");
   return x+y;
}

char sum(char x, char y) { // Вариант 2
   printf("\nChar - Char :");
   return x+y;
}
float sum(float x, float y) { // Вариант 3
   printf("\nFloat - Float :");
```

```
   return x+y;
}
int sum(int *x, int n) { // Вариант 4
 printf("\nInt[]:");
 for (int j=0, i=0; i<n; j+=x[i++]);
 return j;
}
int sum(int *x, int *y, int n) {  // Вариант 5
   printf("\nInt[] - Int[] :");
   for (int j=0, i=0; i<n; j+=x[i]+y[i], i++);
   return j;
}

void main() {
   int i=1, j=2;
   char x=3, y=4;
   float a=5.0, b=6.0;
   static int c[]={1,2,3,4,5}, d[]={1,2,3,4,5};

// Примеры вызова переопределенных функций

   printf("%d (1)", sum(i,j));
   printf("%d (2)", sum(x,y));
   printf("%f (3)", sum(a,b));
   printf("%f (4)", sum(a,1));
   printf("%f (5)", sum(a,(float)1));
   printf("%d (6)", sum(c,sizeof(c)/sizeof(*c)));
   printf("%d (7)", sum(c,d,sizeof(c)/sizeof(*c)));

// Инициализация указателя именем переопределенной функции

   int (*act)(int,int)=sum;
   printf("%d (8)", (*act)(i,j));
   printf("%d (9)", (*act)(int(a),int(b)));
   printf("%d (10)", (*act)(a,b));
}
```

Результаты работы программы:

```
Int - Int :3 (1)
Char - Char :7 (2)
Float - Float :11.000000 (3)
Int - Int :0.000000 (4) // Ошибка, но возможно из-за формата
Float - Float :6.000000 (5)
Int[] :15 (6)
Int[] - Int[] :30 (7)
Int - Int :3 (8)
Int - Int :11 (9)
Int - Int :11 (10)
```

Имена переопределенных функций могут использоваться в любых операциях как имена других функций. Выбор компилятором подходящей версии функции производится поэтапно до выявления сопоставимости типов аргументов и параметров – вначале по условию точного совпадения, а затем по условию совпадения после применения стандартных преобразований. По отноше-

нию к объектам определяемых пользователем классов этот вопрос детально будет рассмотрен в подразд. 3.3.

Возможность переопределения функций в языке C++ реализуется расширением транслятором имени функции символьной информацией о количестве и типе параметров. Такое расширение может оказаться нежелательным, например, при включении в программу на языке C++ объектных модулей из файлов и/или библиотек, созданных транслятором языка C.

Функции, подлежащие интерпретации в стиле языка C, следует описывать и/или определять со специальным модификатором типа extern "C":

```
extern "C" long f1(void); // отдельная функция
extern "C" { // группа функций
   void f2(int);
   double f3(double, int);
};

extern "C" void f4(int z) { // определение функции
   // ...
}
```

Определение функции f4 в приведенном виде позволяет использовать ее объектный модуль в системах программирования на языке C. Очевидно, что для использования библиотек стандартных программ, созданных на языке C и представленных в заголовочном файле, например, library.h, достаточно записать

```
extern "C" {
   #include "library.h"
};
```

При описании и/или определении функции язык C++ позволяет задать значения параметров по умолчанию, а также представить функции с переменным числом параметров. Такие возможности обсуждаются в подразд. 1.10 и 1.11. Кроме этого, в подразд. 1.12 рассматривается возможность замены операции вызова функции подстановкой тела функции в точку вызова.

1.10. Установка умалчиваемых значений параметров функций

Использование умалчиваемых значений параметров улучшает читаемость текста программы за счет исключения в операторе вызова функции предопределенных выражений.

Например, пусть определена функция

```
void f_name(int par1, int par2=1,int par3=2) { /* ... */ }
```

Возможные операторы вызова функции f_name и их интерпретация в стиле языка C:

```
f_name(x)      ⇔  f_name(x,1,2)
f_name(x,y)    ⇔  f_name(x,y,2)
f_name(x,y,z)  ⇔  f_name(x,y,z)
f_name()  // ошибка
```

Значения параметров функции по умолчанию могут задаваться в ее определении и/или описании. Все параметры такого вида необходимо размещать в конце списка параметров. Умалчиваемое значение параметра устанавливается в списке параметров подобно оператору инициализации

тип_параметра имя_параметра = выражение

(в операторе описания функции имя параметра необязательно, но разделитель между элементом «тип_параметра» и символом «=» должен быть). Значение выражения вычисляется в момент вызова функции. Выражение может включать любые допустимые операции, в том числе и вызов функций. Имена глобальных объектов использовать можно, но имена из списка параметров нельзя (параметры являются локальными объектами блока функции). Возможность установки значений параметров в описании функции позволяет использовать в разных блоках различающиеся выражения и наборы умалчиваемых параметров:

```
#include <stdio.h>
#include <string.h>

void test(char *p1="x1-?",char *p2="x2-?",char *p3="x3-?");
void test(char *p1,char *p2,char *p3) {
   printf("\n %s %s %s",p1,p2,p3);
}

char *zx="P1 ???";
char *dsprpl(int n=80) {
   char *x=new char[(n<40)? 40:n];
   sprintf(x,"p3=empty, size=%d\0",n);
   return x;
}

void omega() {
   void test(char *p1=zx, char *p2="q2-?", char *p3=dsprpl());
   test();
   test("e");
}

void main () {
   test();
   test("a");
   test("a","b","c");
   void test(char *p1=zx, char *p2="p2-?", char *p3="p3-?");
   test();
   test("a","b");
   omega();
}
```

Результаты работы программы:

```
x1-? x2-? x3-?
a x2-? x3-?
a b c
P1 ??? p2-? p3-?
a b p3-?
P1 ??? q2-? p3=empty, size=80
e q2-? p3=empty, size=80
```

1.11. Функции с переменным числом параметров

Переопределение функций или задание умалчиваемых параметров по умолчанию часто не позволяет определять функции с неопределенным количеством и типом параметров. В таких ситуациях приходится отказываться от контроля компилятором корректности обращения к функции, указывая лишь факт наличия переменного количества параметров. Ответственность за правильное использование параметров возлагается на программиста. Признаком функции с переменным количеством параметров является завершение списка ее параметров многоточием (...). Например, описание библиотечной функции

```
int printf(char *,...);
```

означает, что при вызове printf должен быть по меньшей мере один параметр типа char*, а остальные могут быть лишь по потребности:

```
printf("СПИСОК СОТРУДНИКОВ\n");
printf("КАФЕДРА %s, ФАКУЛЬТЕТ %s\n",k_n,f_n);
printf("%d + %d = %d\n",2,3,5);
```

Здесь количество и типы параметров должны соответствовать управляющим символам форматной строки, но компилятор этого учесть не может:

```
char *format="ФАМИЛИЯ,И.О.: %40s Оклад: %6d";
// Синтаксически правильные операторы
printf(format);                          // ???
printf(format,"Иванов",1000);            // правильно
printf(format,"Иванов","Василий",1000); // ???
```

Функции с переменным числом параметров должны иметь по крайней мере один аргумент для привязки стека значений параметров (в языке С параметры передаются по значениям). Доступ к значениям параметров переменной части списка можно организовать по указателям, начиная с последнего фиксированного параметра. Зная для любого параметра его адрес и тип, адрес следующего параметра определяется смещением на величину sizeof(тип). Однако действительное значение адреса следующего параметра в некоторых системах может быть скорректировано с учетом правил выравнивания. По последней причине рекомендуется строить механизм выборки параметров на основе си-

стемно независимых макрокоманд, определенных в библиотечном файле stdarg.h. Файл stdarg.h содержит следующие макроопределения для организации доступа к списку значений параметров функции:

```
void va_start(va_list ap, lastfix);
type va_arg(va_list ap, type);
void va_end(va_list ap);
```

Здесь lastfix – имя последнего фиксированного параметра, ap – указатель очередного параметра типа type.

Тип va_list указателя ap определен в файле stdarg.h оператором

```
typedef void *va_list;
```

Макрокоманда va_start инициализирует указатель ap значением адреса параметра, следующего за последним фиксированным параметром lastfix. Отсюда следует, что такая макрокоманда обязательно должна использоваться до вызова макрокоманд va_arg или va_end.

Примеры реализаций макрокоманды va_start:

```
// C++
#define va_start(ap, lastfix) (ap=...)

// C
#define va_start(ap, lastfix) \
 ((void)((ap)=(va_list)\
 ((char*)(&lastfix)+((sizeof(lastfix)+1)&0xFFFE))))
```

Макрокоманда va_arg предназначена для выборки очередного параметра типа type и установки указателя ap на следующий параметр.

Примеры реализаций макрокоманды va_arg:

```
// C++
#define va_arg(ap, type) (*((type *)(ap))++)

// C

#define va_arg(ap, type)\
 (*(type *)(((*(char **)&(ap))+=\
 ((sizeof(type)+1)&0xFFFE))-(((sizeof(type)+1)&0xFFFE))))
```

Макрокоманда va_end выполняет «закрытие» списка аргументов для исключения возможности его использования без повторного использования команды va_start. Примеры реализаций макрокоманды va_arg:

```
// C++
#define va_end(ap) ((void)0)

// C

#define va_end(ap)
```

Рассмотрим пример функции с переменным числом параметров целого типа:

```
#include <stdio.h>
#include <stdlib.h>
#include <starg.h>

int sum(int n,... ) {
   va_list a; // Указатель текущего параметра
   va_start(a,n);
   for (int i=0, b=0; i < n; i++) b+=va_arg(a,int);
   va_end(a);
   return(b);
}

void main () {
   printf("\n1) %d",sum(5,1,2,3,4,5));
   printf("\n2) %d",sum(4,6,7,8,9));
   printf("\n3) %d",sum(4,6,7.0,8,9));
   printf("\n1) %d",sum(5));
}
```

Результаты работы программы:

```
1) 15
2) 30
3) 6   <---- Ошибка преобразования типа по вине программиста
4) 5
```

Количество элементов списка здесь указывается фиксированным в полном списке аргументов параметром n. Возможны другие приемы пометки количества параметров (например, некоторое особое значение последнего параметра). Типы параметров переменной части списка могут быть в общем случае любыми. Рекомендуется минимизировать применение функций с переменным числом параметров ввиду отсутствия со стороны компилятора контроля за правильностью их преобразования.

1.12. Встраиваемые функции

Использование функций связано с потерями машинного времени на выполнение действий по сохранению и восстановлению регистров общего назначения, формированию списков параметров и т. д. (пролог и эпилог функции). Для относительно небольших функций величина таких потерь может оказаться соизмеримой с временем выполнения основных операций тела функции.

При программировании в стиле языка С небольшие, но часто используемые выражения обычно оформляют как макроопределения с параметрами. Однако использование механизма макросредств имеет два недостатка:

- отсутствие контроля типов параметров макрокоманд;
- наличие побочных эффектов в макрорасширениях.

Пример проявления последнего недостатка:

```
// Макроопределение операции возведения в квадрат
//
#define SQUARE(X)   (X)*(X)
// ...
int x,y;
// Пример макрокоманды
y=SQUARE(x++);
// ...
// Полученное макрорасширение
y=(x++)*(x++);
```

Результат вычисления здесь будет неверным, т. к. естественно ожидаемое макрорасширение должно быть эквивалентно следующему:

```
y=x*x, x++;
```

В языке C++ дополнительным атрибутом inline в определении функции можно рекомендовать компилятору, по возможности, встроить код тела функции в место вызова. Компилятор игнорирует эту рекомендацию в следующих ситуациях:

- наличие операторов организации циклов (for, do, while), переключателей (switch) и безусловного перехода (goto) в функциях, возвращающих значения;
- наличие оператора return в функциях, не возвращающих значения;
- обнаружение рекурсивного вызова функции;
- использование статических переменных (с атрибутом static).

Функции с атрибутом inline сохраняют общие свойства функций (локальные переменныые, возможность вычисления адреса, преобразование параметров и возвращаемых значений и т. п.).

Примеры использования встраиваемых функций:

```
#include <stdio.h>
inline int maxint(int x, int y=4) { return((x>y)?x:y); }
inline void swap(int &x, int &y) { int t=x; x=y, y=t; }
inline void swap(float &x, float &y) {
   float t=x;
   x=y, y=t;
}
void main () {
   int x=1, y=2;
   printf("\n %d %d %d",x,y,maxint(x,y));
   swap(x,y);
   printf("\n %d %d %d",x,y,maxint(x,y));
   int (*act)(int ,int =3)=maxint;
   printf("\n %d %d %d",x,y,(*act)(x));
   printf("\n %d %d %d",x,y,maxint(x));
   float a=1.0, b=2.0; swap(a,b);
   printf("\n %.1f %.1f",a,b);
}
```

Результаты работы программы:

```
1 2 2
2 1 2
2 1 3
2 1 4
2.0 1.0
```

Подобно макрорасширениям, встраиваемые функции увеличивают объем программы. При программировании на языке C++ рекомендуется вместо макросредств во всех возможных случаях отдавать предпочтение встраиваемым функциям. Возможность контроля типов параметров и автоматического преобразования значений аргументов упрощает процесс программирования.

1.13. Управление размещением объектов в памяти

Любой именованный объект программы размещается в статически либо автоматически распределяемой памяти. Статический объект размещается во время загрузки программы и существует в течение всего времени ее выполнения. Автоматический объект размещается каждый раз при входе в его блок и существует только до момента выхода из блока. Часто возникает потребность управляемого размещения объектов в памяти в соответствии с алгоритмом решения задачи **без привязки** к блокам программы. В языках C и C++ такие объекты могут адресоваться только косвенно по значению указателя. Указатель может иметь при этом имя, но адресуемый им объект является безымянным.

Управление размещением объектов осуществляется операциями захвата и освобождения памяти. В языке C для этих целей приходится пользоваться библиотечными функциями. Например, в файле alloc.h декларированы функции:

void *malloc(unsigned nbytes) – возврат указателя на выделенную область размером nbytes (NULL при недостатке памяти или nbytes==0);

void free(void *block_pointer) – освобождение захваченной памяти по заданному адресу block_pointer.

Операции захвата и освобождения памяти в стиле языка C имеют вид

```
указатель_на_объект=malloc(sizeof(атрибуты_типа_объекта));
free(указатель_на_объект);
```

В языке C++ операция захвата памяти записывается в следующих видах:

```
указатель_на_объект = new атрибуты_типа_объекта;
указатель_на_объект = new атрибуты_типа_объекта(значение);
```

(здесь «значение» определяет начальное состояние создаваемого объекта для некоторых типов данных).

Результат операции захвата памяти – значение указателя на выделенную для размещения объекта указанного типа область памяти. Признак нехватки памяти – пустое значение указателя (0 или NULL).

Примеры программирования захвата памяти:

```
int  *x;
char *y;
 char *z="abcdef";
 ...
 x=new int;                // создание простого объекта
 y=new char[sizeof(z)+1]; // создание массива символов
 ...
 float *s=new float[n];  // создание массива из n элементов
 int *d=new int(-12345); // создание и инициализация объекта
```

Схема выполнения последнего оператора:

```
int *d=new int;  // создание объекта – захват памяти
*d=int(-12345); // конструирование конкретного объекта
```

Очевидно, что массивы объектов подобным образом инициализироваться не могут. Созданный с помощью операции new объект существует до завершения программы, если не использована операция освобождения памяти delete (сказанное справедливо и для объектов, создаваемых библиотечными функциями).

Варианты использования операции освобождения памяти:

```
delete указатель_на_объект;
delete[] указатель_на_объект;
```

(вторая форма применяется для определяемых пользователем векторных типов данных).

Операция delete может применяться только к значению указателя, который был возвращен операцией new. Повторное освобождение памяти чревато аварийными последствиями. После освобождения памяти можно присвоить указателю нулевое значение, т. к. выполнение операции delete с нулевым операндом не вызывает никаких действий.

Пример программы считывания файлов в память:

```
#include <stdio.h>
#include <io.h>

void main(int na, char **la) {
 FILE *inpfil;
 char *buffer;
 size_t number;
 if (na>1) {
  if ((inpfil=fopen(la[1],"rb"))!=NULL) {
   long length=filelength(fileno(inpfil));
   buffer=new char[number=(size_t)length];
   if (buffer) { // Обработка данных
```

```
      fread(buffer,sizeof(*buffer),number,inpfil);
      delete buffer;
     } else printf("\nНет памяти");
   } else printf("\nФайл %s не найден",la[1]);
  } else printf("\nВызов: %s имя_файла",la[0]);
}
```

Операции управления памятью буфера в стиле языка С имели бы вид:

```
buffer=malloc(number*sizeof(*buffer)); // захват памяти
free(buffer); // освобождение памяти
```

Стандартные библиотечные функции образуют достаточно гибкий набор средств управления памятью. Например, библиотечная функция

```
void *calloc(unsigned n_elem, unsigned l_elem)
```

наряду с захватом памяти для n_elen объектов размером l_elem байт заполняет ее нулями, а функция

```
void *realloc(void *blk_ptr,unsigned nbytes)
```

изменяет размер ранее распределенного по адресу blk_ptr блока на новое значение nbytes и копирует при необходимости содержимое блока в новую область памяти.

Кажущийся недостаток гибкости операторов new и delete легко устраняется возможностью их **переопределения**. Механизм переопределения операторов в языке С++ базируется на переопределении функций и подробно рассматривается в подразд. 3.2. Поясним идею такого механизма. Операции управления размещением объектов в языке С++ с точки зрения транслятора реализуются функциями

```
void *operator new(size_t size);
void operator delete(void *pointer,size_t size);
```

(здесь size_t – тип данных для представления размера объекта в байтах).

Можно переопределить эти функции с учетом потребностей задачи, например, для использования какого-то специализированного вида памяти:

```
void *operator new(size_t size) {
  void *p=malloc(size);
  printf("\nЗапрос %d байт, адрес блока %p",size,p);
  return p;
}

void operator delete(void *pointer,size_t size) {
   printf("\nОсвобождение блока по адресу %p",pointer);
   free(pointer);
}
```

(второй параметр функции delete для объектов базовых типов игнорируется, но формально должен присутствовать).

Приведенные функции переопределяют операции new и delete глобально для объектов всех типов. В общем случае в языке C++ можно конкретизировать область применения функций, переопределяющих операторы для создаваемых программистом классов объектов.

Напомним, что многомерные массивы в языке С рекурсивно представляются массивами указателей на массивы с уменьшенным на единицу количеством измерений. Захват памяти посредством использования библиотечных функций или оператора new не влечет построения соответствующих связей, т. к. это связано с необходимостью нетривиальной инициализации полей выделенной памяти. Отсюда следует, что управление размещением в памяти массивов с более чем одним измерением при намерении последующего использования стандартных операций обращения к элементам массивов по индексам требует программирования процесса захвата и освобождения памяти элементарных одномерных массивов.

Приведем пример программы с динамическим распределением двумерных массивов.

```
#include <stdio.h>
typedef int type; // Определение типа элементов матрицы

// Процедура создания прямоугольной матрицы

type **matrixn(int m, int n) {
   if ((m<=0)||(n<0))
      return NULL;

// Создание массива указателей строк матрицы

   type **x=new type *[m];
   if (x!=NULL) {

// Создание строк матрицы

      for (int i=0; i<m; i++)
        if ((x[i]=new type[n])==NULL) {

// Контроль наличия памяти

            while (i) delete[] x[--i];
            delete[] x;
            x=NULL;
            break;
         }
   }
   return x;
}

// Процедура уничтожения прямоугольной матрицы

void matrixd(type **x, int m, int n) {
   if (x!=NULL) {
```

```
// Уничтожение строк матрицы

      for (int i=0; i < m; i++)
         delete[] x[i];

// Уничтожение массива указателей строк матрицы
      delete[] x;
   }
}

// Процедура построчной печати прямоугольной матрицы

void matrixp(type **x, int m, int n) {
   if (x!=NULL) {
      for (int i=0; i < m; i++) {
         printf("\n%3d) ",i);
         for (int j=0; j < n; j++)
            printf(" %6d",x[i][j]);
      }
   }
}

void main() {
   int m=4, // Количество строк матрицы
   n=5; // Количество столбцов матрицы
   type **x=matrixn(m,n); // Создание матрицы x[m][n]
   if (x!=NULL) {

// Заполнение матрицы номерами ее элементов
      for (int k=0, i=0; i<m; i++) {
         for (int j=0; j<n; j++)
            x[i][j]=k++;
      }
      matrixp(x,m,n); // Печать матрицы x[m][n]
      matrixd(x,m,n); // Уничтожение матрицы x[m][n]
   }
}
```

Результаты работы программы:

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

Оценка возможности размещения объектов в памяти может быть проведена библиотечной функцией coreleft().

Следует учитывать, что в процессе захвата и освобождения памяти «сборка мусора» не производится. В большинстве систем программирования память выделяется из единого стека свободной памяти. Во избежание фрагментации памяти, возникающей при случайном характере освобождения захваченных областей, рекомендуется в критических по памяти программах создавать

отдельные модули высокоуровневого управления памятью с учетом специфики конкретной задачи. Для сокращения расхода памяти на блоки управления памятью следует уменьшать количество захватываемых областей.

1.14. Пространства имен

Пространство имен определяет область видимости имен объектов, которые не входят ни в один блок. Пространство имен не может быть определено внутри блока.

Синтаксис:

```
namespace имя_пространства_имен {
 область описания и определения объектов
}
```

Пример определения:

```
namespace MyNames { //  файл test.cpp
  int    m;
  int    add(int a, int b) {
         return(a+b);
  }
}
```

Пространства имен открыты, т. е. одно и то же пространство имен можно расширять в разных исходных файлах:

```
namespace MyNames { //  файл  function.h
  int    n;
  int    sub(int a, int b);
}
```

Пространства имен могут быть вложены друг в друга, например:

```
namespace PublicNames {
  int    n;
  namespace MyNames {
   int add(int a, int b) {
    return (a+b);
  }
  }
}
```

По умолчанию существует анонимное пространство имен, которое доступно во всех исходных файлах и которое не нужно объявлять. Это пространство имен называется глобальным. Можно сказать, что все пространства имен вложены в глобальное. Все имена, которые не входят ни в одно пространство имен, принадлежат глобальному. Имена элементов стандартных библиотек C и

C++ , объявленные в заголовочных файлах (с расширением .h), принадлежат глобальному пространству имен.

В любом исходном файле можно объявить локальное анонимное пространство имен, которое видно только в этом файле:

Синтаксис:

```
namespace {
  ...
}
```

По умолчанию в языке C существует пространство имен std, которое называется стандартным. Имена элементов из новых стандартных библиотек C и C++ находятся в этом пространстве. Имена заголовочных файлов для новых стандартных библиотек не включают расширение, например:

```
<cstdlib>       // C
<iostream>      // C++
```

Для доступа к именам пространства имен используется операция разрешения области видимости или привязки имени «::», перед которой указывается имя нужного пространства имен (квалификатор). Операция привязки выполняется на этапе компиляции программы и не влияет на ее производительность:

```
#include        <iostream>
#include        "function.h"
namespace MyNames {
  ...
}

int      MyNames :: sub(int a, int b){
  return (a-b);
}
int      main() {
  int    sum, diff;
  MyNames :: m = 2;
  MyNames :: n = 3;
  sum = MyNames :: add(MyNames :: m, MyNames :: n);
  diff = MyNames :: sub(MyNames :: m, MyNames :: n);
  std :: cout<<sum<<std :: endl;
  std :: cout<<diff<<std :: endl;
  return (1);
}

namespace PublicNames {
  ...
}
void f() {
  int    n;
  PublicNames :: n = 10;
  n = PublicNames :: MyNames :: add(PublicNames :: n, 5);
  std :: cout<<n<<std :: endl;
}
```

Для доступа к именам из локального анонимного пространства имен оператор разрешения области видимости не используется. Доступ к именам переменных, находящихся в локальном анонимном пространстве имен, невозможен, если это имя определено в глобальном анонимном пространстве:

```
int      n = 10;
namespace {
  int    n = 1; // доступ невозможен
  int    add(int n) {
   return(n + :: n);
  }
}
void main() {
  int    n = add(5);
  std :: cout<<n<<std :: endl;
}
```

Для объявления имени из некоторого пространства имен используется оператор using, который может находиться как в глобальной области видимости, так и внутри блока, а также в области видимости любого пространства имен:

```
#include <iostream>
using    std :: cout;  // для одного идентификатора
void main() {
 cout<<”Hello!”;
}
```

Оператор using может быть использован и для вложенных пространств имен:

```
#include <iostream>

using std :: cout;
namespace PublicNames {
  namespace MyNames {
   int add(int a, int b) { return a+b; }
  }
}

void main() {
  using  PublicNames :: MyNames :: add;
  cout<<add(2,3);
}
```

Для объявления сразу всех имен из некоторого пространства имен используется **директива** using.

Синтаксис:

using namespace имя_пространства_имен

Пример:

```
#include <iostream>
using namespace std;
void main() {
  cout<<”Hello”<<endl;
}
```

Правило использования директивы using совпадает с правилами использования оператора using. В случае вложенных пространств имен директива using с именем вложенного пространства имен не объявляет имена из объемлющего пространства имен.

Чтобы избежать дублирования для пространства имен, часто выбирается длинное имя. В этом случае для удобства записи длинному имени может быть назначен короткий псевдоним.

Синтаксис:

using namespace псевдоним = имя_пространства_имен;

Пример:

```
namespace Data_Base_Interface { /* ... */ }
namespace DBI = Data_Base_Interface;
```

Псевдонимы можно определять и для вложенных пространств имен. Одно пространство имен может иметь несколько псевдонимов, которые могут определяться друг через друга. Псевдонимы пространств имен используются так же, как и имена этих пространств.

Основные недостатки простанств имен:

- отсутствие формального учета вложенных пространств;
- потенциальная опасность запутывания текста из-за использования несовместимых соглашений о наименовании;
- препроцессорные операции над группами идентификаторов, основанных на пространствах имен, могут становиться сложными или даже недопустимыми;
- надежное программирование требует осуществлять идентификацию объектов с полным именем пространств, при этом идентификаторы становятся чересчур длинными.

Пространства имен могут эмулироваться некоторым соглашением о правилах идентификации объектов программы. Например, библиотеки языка С часто используют фиксированный префикс для идентификации всех функций и переменных, являющийся частью их внешнего интерфейса. Тем самым нет возможности для коллизии имен идентификаторов.

2. КЛАССЫ ОБЪЕКТОВ

2.1. Понятие класса объектов

Термин «класс» в языке C++ употребляется для обозначения типов объектов, определяемых пользователем с ориентацией на технологию объектно-ориентированного программирования.

В отличие от определяемых в стиле языка C производных структурированных типов, таких, как структур или объединений, объявление класса позволяет наряду с описанием данных представления объектов указать и аспекты использования таких данных:

- список функций доступа к объектам;
- правила наследования объектов производными классами;
- различная степень защиты элементов объектов.

Функции доступа к объектам класса могут переопределять умалчиваемые операции создания и уничтожения объектов, а также позволить обозначать различные действия над объектами традиционными в языке C символами операций.

Основная цель введения класса – отделение частных деталей представления объектов класса (скрытых данных и вспомогательных функций) от элементов, присущих объекту при правильном его использовании (открытые данные и функции доступа к любым данным). Пользователь класса может оперировать в программе его высокоуровневым внешним представлением, а контроль за корректностью использования данных и функций возлагается на компилятор.

В языке C++ определение классов ведется посредством расширения понятий структуры (struct) и обьединения (union). Традиционная для языка C интерпретация таких понятий остается в силе, что позволяет использовать программы на языке C без изменения исходного текста.

Синтаксис определения класса на языке C++ имеет вид

```
вид_класса имя_класса {
   описание_элементов_класса
};
```

Определение класса представляет лишь описание некоторого типа объектов, имя которого «вид_класса имя_класса». Между символами «}» и «;» можно разместить список имен определяемых объектов класса с использованием при необходимости операции инициализации. Вне оператора определения класса объекты определяются операторами декларации вида

```
вид_класса имя_класса список_имен_объектов;
```

В большинстве случаев наиболее целесообразно определение класса и его элементов поместить в отдельный заголовочный файл.

Рассмотрим подробнее элементы оператора определения класса.

Элемент «вид_класса» может принимать значения struct, union или class. Напомним, что различие между структурами и объединениями заключается в способе размещения их элементов в памяти – элементы структур располагаются последовательно, а элементы объединений размещаются от начального адреса одной и той же области, достаточной для размещения наибольшего из них.

Необязательный для классов вида struct и union элемент «имя класса» задается идентификатором. Ввиду однозначности соответствия имени класса его виду при ссылке на тип класса элемент «вид класса» в языке C++ обычно опускается даже для структур и объединений. Например, пусть описана структура

```
struct subject {
   char fio[80];
   int dolg;
};
```

Примеры ссылок на такой тип данных в стиле языка C:

```
struct subject teacher, *student;
student=(struct subject *)malloc(sizeof(struct subject));
```

В стиле языка C++ последние выражения допускается записывать в виде

```
subject teacher, *student;
student=(subject *)malloc(sizeof(subject));
```

Описание элементов класса может в общем случае включать декларацию элементов данных представления объекта и/или функций доступа к данным с указанием атрибутов права доступа. Иногда элементами являются описания собственных типов класса (например, структурированных типов объектов и перечислений). Принципиально возможно использование вложенных классов.

Элементы класса могут иметь один из атрибутов права доступа:

- private (локальный) – доступ только из функций доступа внутри класса;
- protected (защищенный) – доступ из функций доступа внутри класса и производных классов;
- public (глобальный) – доступ из любых функций (элементы интерфейса пользователя класса).

Элементы класса вида union могут иметь только атрибут доступа public. Различие между классами видов struct и class есть лишь в значении по умолчанию атрибутов доступа: элементы класса вида struct имеют атрибут public, а класса вида class – private.

Явное назначение атрибута доступа к элементам класса оформляется записью перед описанием элементов ключевого слова private, protected или public

и символа «:» (для последующих элементов с совпадающим атрибутом доступа повторение описания этого атрибута не обязательно):

```
class date {

// Описание собственных типов класса

   enum date_month={ // по умолчанию атрибут доступа private
       January=1, February, March, April, May, June,
       July, August, September, October, November, December };

// Декларация элементов данных класса

   int day, month, year; // private
public: // изменение атрибута доступа

// Описание функций-элементов

   void set(int, int, int); // public
   void next(); // public
protected: // изменение атрибута доступа
   void print(); // protected
};
```

Очевидно, что декларация структуры

```
struct name { type xxx; .... }
```

эквивалентна декларации класса

```
class name {
public:
   type xxx;
   ....
}
```

Элементы данных класса описываются обычным образом, но описание функций доступа может потребовать отражения их дополнительных свойств по отношению к классу.

По отношению к праву доступа к защищенным элементам класса различают:

- функции-элементы;
- дружественные функции класса.

Среди функций, имеющих право доступа, можно выделить:

- функции-конструкторы – создание объектов;
- функции-деструкторы – уничтожение объектов;
- функции определения операций над объектами.

Функции-элементы и дружественные функции, перечисленные в описании класса, имеют право доступа ко всем элементам класса и всем глобальным элементам вне класса. Функции-элементы находятся в области действия класса и вызываются с помощью операции выбора элемента структуры.

Дружественные функции не находятся в области действия класса и вызываются по общим правилам. Внешние функции имеют право доступа только к открытым элементам класса.

Особенности построения и использования функций доступа, ссылок на элементы класса и размещения их в памяти, а также правила построения производных классов рассматриваются ниже.

Определение класса должно быть глобальным. Иногда требуется упреждающая ссылка на имя некоторого класса class_name до его фактического определения. В исходном тексте можно использовать предварительное описание, которое может отражать и некоторые особенности класса:

```
class [__single_inheritance] class_name;
class [__multiple_inheritance] class_name;
class [__virtual_inheritance] class_name;

class __single_inheritance S;
int S::*p;
```

2.2. Функции-элементы

Функции, описания либо определения которых помещены в тело оператора определения класса, ранее названы функциями-элементами такого класса. Принципиальное отличие функций-элементов от обычных функций, декларированных вне какого-либо класса, состоит в том, что функции-элементы могут вызываться только для работы с объектами соответствующего месту их описания типа.

Пусть понятие даты в стиле языка С задано структурой

```
struct date { // дата:
   int   day, //  день,
   month,  //  месяц,
   year;  //  год
};
```

Предположим, что для работы с объектами типа date используется множество функций:

```
void set_date(date *, int, int, int); // установка даты
void next_date(date *); // получение даты следующего дня
void print_date(date *); // печать даты
```

Очевидно, что явной связи между функциями и типом данных, контролирумой транслятором, здесь нет. Элементы структуры типа date доступны любой функции. Перепишем определение структуры date, учитывая понятие класса в языке С++:

```
struct date {
// описание элементов данных
   int day, month, year;              // public
```

```
// описание функций-элементов
   void set(int, int, int);          // public
   date& next();                     // public
   void print();                     // public
};
```

Здесь данные и функции по-прежнему доступны из любой внешней функции, но использование перечисленных в блоке определения класса функций транслятор может контролировать. Приведем пример программы по оценке влияния наличия функций-элементов на размер структуры:

```
#include <stdio.h>

struct c1 { // Структура без функций-элементов
   int x,y,z;
};

struct c2 { // Структура с функциями-элементами
   int x,y,z;
   void print();
   void set();
};

#define P(X) printf("\n Sizeof(%s)=%d",#X,sizeof(X))
void main() {
   P(c1);
   P(c2);
}
```

Результаты работы программы:

```
Sizeof(c1)=12
Sizeof(c2)=12
```

Можно заметить, что наличие функций-элементов в рассмотренном простейшем варианте класса не вызывает увеличения размера объектов. Независимо от количества объектов некоторого класса его функции-элементы в модуле программы присутствуют в единственном экземпляре (на механизм встраивания функций внимание можно не обращать). Перейдем к рассмотрению особенностей вызова и построения функций-элементов. Вызов функций-элементов реализуется использованием стандартного для языка С синтаксиса доступа к элементам структуры:

```
date today;
date next_day;
date *test_day=new date;

void f() { // непосредственный выбор функций-элементов
   today.set(1,9,2005);
   next_day=today.next();
   today.print(); // косвенный выбор функции-элемента
   test_day->print();
}
```

Обращения к функциям вида

```
set(19,01,2013);
```

без указания имени или косвенной ссылки на объект требуемого типа считаются относящимися к внешне определенным функциям.

Таким образом, использование функции-элемента возможно только после явного или косвенного определения объекта класса. Операция вызова функции-элемента оказывается привязанной к экземпляру объекта класса, но функция-элемент имеет доступ к любым открытым функциям и данным вне класса. Отсюда следует, что устанавливается ограничение доступа к элементам класса: функция-элемент может использовать любые данные и функции, но закрытые элементы класса доступны только функциям-элементам класса.

Чаще всего интерфейс пользователя класса удобно определить набором функций-элементов с атрибутом доступа public, а элементы-данные и вспомогательные функции скрыть.

Декларация функций-элементов допустима как в форме описания, так и в форме определения. Никаких ограничений на возможности декларации (переопределение, установка параметров по умолчанию и т. п.) нет. Определяемым в блоке любого класса функциям-элементам автоматически приписывается атрибут inline. Если некоторой функции-элементу атрибут inline не может быть назначен (см. подразд. 1.12), то для исключения выдачи диагностических сообщений транслятора определение такой функции приходится размещать вне блока класса.

Пример определения функции-элемента в описании объединения:

```
union word {
// декларация элементов-данных
   int i;                  // public
   char b[2];              // public
// декларация функций-элементов
   void print();           // public, описание
   void set(int x) {i=x};  // public, определение
};
```

Поскольку разные классы могут иметь функции-элементы с одинаковыми именами, то при определении функции-элемента вне описания класса необходимо указывать имя класса:

```
void date::print() {
   printf("%02d.%02d.%d",day,month,year);
}
void word::print() { printf("%04x<==>%02x%02x",i,b[0],b[1]); }
```

Символы «::» здесь обозначают бинарную операцию привязки имени к элементам класса или, как иногда говорят, разрешения его видимости в некотором классе. Синтаксис бинарной операции привязки:

```
<имя_класса>::<имя_члена_класса>
```

Отсутствие имени класса означает обращение к имени глобального объекта программы (см. подразд. 1.4). В теле функции-элемента имена элементов класса могут использоваться без явной ссылки на объект. В этом случае имя относится к элементу того объекта, для которого функция была вызвана (например, функции word::set, date::print, word::print). В случае же использования при определении функции-элемента имен локальных объектов, совпадающих с именами элементов класса необходимо явно применить операцию привязки «::». Пример использования операции привязки:

```
#include <stdio.h>

int n=-1; // глобальная переменная
class x {
   int m;
public:
   void set_m(int m=::n) { // использование глобального n
      x::m=m; // m – имя элемента класса и параметра функции
   };
   int readm() {
      return m; // m – имя элемента класса
   }
};

x a, b;

void main() {
   a.set_m(); // инициализация a.m
   b.set_m(0); // инициализация b.m
   int a = ::a.readm(); // a=::a.m;
   int b = ::b.readm(); // b=::b.m -?; // ...
   printf("\n a=%d, b=%d",a,b);
}
```

Результаты работы программы:

```
a=-1, b=0
```

Таким образом, в операторном блоке тела функции-элемента имена нелокальных объектов по умолчанию считаются дополненными справа именем соответствующего класса и парой двоеточий. Операция привязки может применяться для ссылок на определения любых объектов класса, например:

```
class abc {
public:
   enum { // множество кодов ошибок
   overflow, underflow, zerodivide
   };
   // ...
};
// ...
void f(int e) { // функция обработки ошибок
```

```
   switch (e) {
      case abc::overflow:   /* ... */
      case abc::underflow:  /* ... */
      case abc::zerodivide: /* ... */
   }
}
```

Рассмотренный способ привязки имен элементов класса основан на понятии типа объекта. Кроме этого, язык C++ предоставляет возможность привязки имен на основе понятия экземпляра объекта.

В каждую нестатическую функцию-элемент класса X неявно передается скрытый параметр X *const this, ссылающийся на объект, для которого выполнена операция вызова (переопределить идентификатор this нельзя, т. к. это ключевое слово языка C++).

Приведем другое определение класса x:

```
class x {
   int m;
public:
   void set_m(int m=::n) {  // использование глобального n
      this->m=m;  // m – имя параметра и элемента класса
   };
   int readm() { return this->m; }
};
```

Ссылка на элементы класса с использованием указателя this – другая альтернатива устранения коллизии имен. Реальная потребность обращения к this возникает лишь при необходимости работы непосредственно с указателями объектов. Пример использования указателя this при построении симметричного (двухсвязного) списка:

```
// Определение класса элементов симметричного списка
class double_list {
   // ...
   double_list *prec; // указатель предшествующего элемента
   double_list *next; // указатель следующего элемента
   // ...
public:
   void append(double_list *);
   // ...
};

// Определение функции включения элемента
// симметричного списка

void double_list::append(double_list *p) {
   p->next = next; // p->next = this->next
   p->prec = this;
   next->prec = p; // this->next->prec = p
   next = p; // this->next = p
}

// Определение указателя симметричного списка
```

```
double_list* head; // глобальная переменная

// Пример функции создания симметричного списка

void f() {
   double_list x1, x2;
   // ...
   head->append(&x1); // включить x1
   head->append(&x2); // включить x2
}
```

Здесь следует отметить, что единицей защиты в C++ является класс, а не отдельный объект класса. Объекты, адресуемые указателями this, prec и next, относятся к классу double_list, поэтому функции double_list::append() одновременно доступен любой из них. В последнем примере указатель списка head не может быть модифицирован, поэтому при пустом списке, когда (head==NULL), функция double_list::append будет работать неверно. Возникает потребность в использовании общих элементов данных, относящихся ко всем объектам класса. Такими элементами могут быть либо глобальные переменные либо статические элементы класса (см. подразд. 2.3). Изменение указателя this как указателя-константы функцией-элементом не допускается.

Если объект, адресуемый указателем this, также не должен изменяться при вызове некоторой функции-элемента, то такую функцию следует описать с ключевым словом const после списка параметров. Это приведет к построению описания указателя объекта класса X в виде

```
const X *const this;
```

Пример программы с запретом изменения объекта:

```
class X {
   int x;
public:
// функция с запретом изменения объекта
   int read() const {
      return x;
   }
// функция без запрета изменения объекта
   void write(int i) {
      x=i;
   }
};

X var; // декларация объекта-переменной
const X fix; // декларация объекта-константы

void main() {
   // ...
   var.read(); // разрешенное действие
   var.write(1); // разрешенное действие
   fix.read(); // разрешенное действие
```

```
    fix.write(1); // ошибка – попытка изменения объекта
    // ...
}
```

Если некоторая функция-элемент представлена в блоке класса описанием, то обсуждаемый здесь модификатор const должен использоваться как в определении, так и в описании функции. В противном случае декларируемые функции будут считаться переопределенными.

Модификатор mutable указывает, что элемент константного объекта может быть модифицирован.

Обсудим возможность использования указателей на функции-элементы. Указатели-переменные на элементы класса X должны иметь модификатор описания вида X::* вместо обычно используемого при декларации указателя символа «*». Значение указателю-переменной можно присвоить оператором определения адреса «&» по имени элемента класса с привязкой к имени класса. Установка связи указателя на элемент класса с объектом возможна посредством специфичных для языка C++ операций «.*» и «->*» (косвенная адресация элемента класса при прямой или косвенной адресации объекта).

Пример программы с использованием указателей на элементы класса:

```
#include <stdio.h>
class alpha {
   int x, y;
public:
   void set(int i) { y=x=i; }
   void print(char *s) { printf("\n%s: x=%d y=%d",s,x,y); }
};

void main() {
   alpha obj; // определение объекта
   obj.set(1);
   obj.print("Вариант 1:"); // Прямой вызов функции-элемента

// Декларация указателя-переменной func на функцию-элемент

   void (alpha::*func)(char *);

   func=& alpha::print; // func указывает на alpha::print
   obj.set(2);
   (obj.*func)("Вариант 2:");// Косвенный вызов функции-элемента

   alpha *p=&obj; // Декларация указателя на объект
   obj.set(3);
   (p->*func)("Вариант 3:"); // Косвенный вызов функции

// Декларация указателя dptr на элемент данных класса alpha

   typedef int alpha::* Int; // Введение имени сложного типа

   Int dptr=(Int)&obj.y; // Приведение типа требует компилятор?!

   (*dptr)++;
```

```
   (p->*func)("Вариант 4");
   printf(" (*dptr=%d)",*dptr);

// Декларация указателя dint на элемент данных типа int

   int *dint=(int *)&obj.y; // Приведение требует компилятор...
   (*dint)++;
   (p->*func)("Вариант 5");
   printf(" (*dint=%d)",*dint);
}
```

Результаты работы программы:

```
Вариант 1: x=1, y=1
Вариант 2: x=2, y=2
Вариант 3: x=3, y=3
Вариант 4: x=3, y=4 (*dptr=4)
Вариант 5: x=3, y=5 (*dint=5)
```

Элементы объекта класса оказались модифицированными, но после явного приведения типов.

2.3. Статические элементы класса

Элементы класса (функции и данные), объявляемые с дополнительным атрибутом static, называются статическими. Статический элемент данных, независимо от количества объектов, будет размещен в единственном экземпляре в статической памяти и инициализирован один раз (по умолчанию, поле статического элемента заполняется нулями).

Таким образом, предоставляется возможность использования общих переменных для всех объектов класса, а потребность в глобальных переменных сокращается. Статическая функция-элемент не привязана к объектам класса. Указатель this ей непередается, поэтому она имеет доступ только к статическим элементам класса. Такая функция может быть вызвана при отсутствии объектов класса.

В любых функциях-элементах класса именование статических элементов ведется обычным образом, но для обращения к открытым элементам такого вида извне их имена дополняются слева именем класса и символами операции привязки «::». Рассмотрим пример использования статических элементов:

```
#include <stdio.h>
#include <string.h>
class list { // список строк:
   static list *head; // указатель списка
   static int size; // размер буфера строк
   list *next; // указатель следующего элемента
   char *data; // указатель поля данных
public:
   void append(char *item) { // добавление строки
      int n=strlen(item)+1;
```

```
        next=head, head=this;
        data=new char[n];
        strcpy(data,item);
        if (size>n) size=n;
    }
    static void print() { // вывод списка строк
        int i=0;
        for (list *item=head;item!=NULL; item=item->next)
            printf("\n %4d) %*s",++i,size,item->data);
    }
    static int isempty() { // проверка наличия списка строк
        return (head==NULL);
    }
};
list *list::head = NULL;
int list::size = 0;

void main () {
    (new list)->append("КАФЕДРА");
    (new list)->append("ФАКУЛЬТЕТ");
    (new list)->append("ВУЗ");
    if (!list::isempty()) {
        printf("\nСПИСОК ПОДРАЗДЕЛЕНИЙ:\n");
        list::print();
    }
}
```

Результаты работы программы:

```
СПИСОК ПОДРАЗДЕЛЕНИЙ:
1) ВУЗ 2) ФАКУЛЬТЕТ 3) КАФЕДРА
```

Можно еще раз обратить внимание на использование функции-элемента list::append – указатель this получается в результате операции захвата памяти.

Статические элементы данных – средство контроля за количеством объектов.

2.4. Дружественные функции класса

Дружественной называется объявленная в классе функция с атрибутом friend, которая, не являясь элементом класса, может использовать его любые элементы.

Декларация дружественной функции может проводиться в любом месте описания класса без учета атрибутов доступа. Декларация может проводиться в форме описания или определения с использованием всех возможностей языка C++. При определении дружественной функции в классе ей автоматически приписывается атрибут inline. Дружественная функция не привязана к объектам класса. Указатель this ей, как и статическим функциям-элементам, не передается. Операция вызова дружественной функции не использует синтаксис выбора элемента структуры, а записывается в обычной форме.

Иллюстрация различия между функцией-элементом и дружественной функцией класса:

```
class example {
   int a;

// Описание дружественной функции (может быть в любом месте)
   friend void friend_set (example *, int);
public:
   void member_set (int);
};

// Определение дружественной функции
void friend_set(example *p,int i) { p->a = i; }

// Определение функции-элемента
void example::member_set(int i) {
   a = i; // example::a=i; <===> this->a=i;
}
example object; // декларация объекта
void f() {

// Эквивалентные по действию операторы

   friend_set(&object,2005); // вызов дружественной функции
   object.member_set(2005); // вызов функции-элемента
}
```

Функция-элемент одного класса может быть дружественной другому классу:

```
class x {
   // ...
   void f(); // любой атрибут доступа
};

class y {
   // ...
   friend void x::f(); // разрешение доступа к любым
                       // элементам класса y функции x::f()
};
```

Если все функции класса X дружественны классу Y, то можно использовать сокращение вида

```
class Y {
   friend X;
   // ...
};
```

Таким образом, дружественные функции могут использоваться для связи элементов разных классов. Принципиальное различие между функциями-элементами и дружественными функциями лишь в синтаксисе доступа к элементам класса. Предпочтение в использовании того или иного вида функций

чаще всего определяется контекстом их использования. Лишь некоторые виды функций доступа (например конструкторы и деструкторы) обязательно должны быть элементами класса.

2.5. Конструкторы и деструкторы объектов

Существование в программе объекта любого типа определяется по отношению к его наличию и доступности в памяти:

- автоматический объект – создается каждый раз при выполнении программы в точке объявления и уничтожается при выходе из блока, в котором он используется;
- статический объект – создается один раз при запуске программы и уничтожается один раз при ее завершении;
- объект в свободной памяти – создается с помощью операции new и уничтожается с помощью операции delete либо при завершении программы.

Фактически автоматический объект может быть создан в любом выражении при явном либо неявном применении операции приведения типа, а также в момент формирования элемента списка параметров и возвращаемого значения функции.

Создание объекта, по крайней мере, включает операцию выделения памяти и, при необходимости, операцию инициализации полей данных. Уничтожение объекта означает отказ от использования занимаемой им памяти.

В современном языке C++ интервал существования объекта класса в памяти ассоциируется с вызовом функций, называемых конструкторами и деструкторами. Вызов таких функций планируется компилятором в местах декларации объекта, конца блока или операции приведения типа, а также при выполнении операций new и delete. Говорят, что конструктор и деструктор создают функциональное замыкание интервала существования объекта класса в памяти.

В языке C++ конструкторы и деструкторы объектов определяемых пользователем классов **могут** объявляться явно в описании класса.

Конструктор – функция-элемент класса, имя которой совпадает с именем класса. Конструкторы могут быть переопределенными, иметь параметры по умолчанию, но не должны возвращать результат.

Деструктор – функция-элемент класса, имя которой совпадает с именем класса, но дополнено слева символом «~» (символ «~» не является допустимым для создания идентификатора объекта другого назначения). Деструктор не может быть переопределен, не должен иметь параметры и возвращать результат.

Пример использования конструктора и деструктора:

```
#include <stdio.h>
#include <string.h>
class simple {
   char *name;
public:
```

```
   void print() { // отображение объекта
      printf("\n***  объект \"%s\" существует",name);
   }
   simple(char *x) { // конструктор
      printf("\nСоздание объекта \"%s\"",x);
      name=new char[strlen(x)+1];
      strcpy(name,x);
   }
   ~simple() { // деструктор
      printf("\nУничтожение объекта \"%s\"",name);
      delete name;
   }
};

simple global("global");

void main () {
   printf("\n* Начало основного блока");
   global.print();
   simple local_1("local_1");
   local_1.print();
   simple *plocal_2=new simple("local_2");
   (*plocal_2).print();
  {
     printf("\n* Начало вложенного блока");
     simple local_3("local_3");
     local_3.print();
  }
  delete plocal_2;
  local_1.print();
}
```

Результаты работы программы:

```
Создание объекта "global"
* Начало основного блока
***  объект "global" существует
Создание объекта "local_1"
***  объект "local_1" существует
Создание объекта "local_2"
***  объект "local_2" существует
* Начало вложенного блока
Создание объекта "local_3"
***  объект "local_3" существует
Уничтожение объекта "local_3"
Уничтожение объекта "local_2"
***  объект "local_1" существует
Уничтожение объекта "local_1"
Уничтожение объекта "global"
```

Отметим, что вызов конструктором и деструктором в рассмотренном примере функций вывода использован лишь для пояснения их сущности. Наиболее часто на практике такие функции используются для управления нетривиальным распределением памяти или других вычислительных ресурсов, а также инициализации сложных объектов (см. элемент name). Другим полезным

их применением может быть программирование неявно определяемого контекста среды выполнения функций и блоков, где декларируются объекты класса.

Рассмотрим пример программы со слежением за процессом ее выполнения во времени на уровне выделяемых блоков.

Идея построения механизма слежения:

- ввести класс agent, объекты которого декларируются в выделяемых блоках;
- предписать конструктору и деструктору класса agent решение задачи формирования отчета о существовании объекта такого класса во времени в виде линейного списка объектов класса report;
- ввести класс shief, объект которого декларируется в управляющем блоке программы и существует во время выполнения выделенных блоков, а деструктор ~shief отображает список отчета.

```
// ПРИМЕР ИСПОЛЬЗОВАНИЯ КОНСТРУКТОРОВ И ДЕСТРУКТОРОВ

#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <time.h>
#include <windows.h>
class report { // отчет работы агента
   static report *head; // указатель списка
   char *name; // имя агента
   clock_t start, finish; // начало и конец существования агента
   report *next; // указатель следующего отчета
   friend class agent;
   friend class shief;
public:
   report(char *x) { // конструктор
      start=clock();
      name=new char[strlen(x)+1];
      strcpy(name,x);
      next=head, head=this;
   } // деструктор не использован
};
class agent { // описание агента
   report *link; // указатель протокола
public:
   agent(char *x) {  // конструктор
      link=new report(x);
   }
   ~agent() { //  деструктор
      link->finish=clock();
   }
};
class shief { // описание шефа
public:
   ~shief() { // деструктор
      for (report *x=report::head; x!=NULL; x=x->next)
         printf("\n %s: S=%d, F=%d, L=%d",x->name,
            (int)x->start,(int)x->finish,
            (int)(x->finish-x->start));
```

```
   }
} Shief;
report *report::head = 0;

void main () {
   void f1(), f2();
   agent  a0("A0");
   f1();
   f2();
}
void f1() {
   agent a1("A1");
   sleep(1);  //  задержка на 1 с (18 тиков)
}
void f2() {
   agent a2("A2");
   sleep(2); // задержка на 2 с (36  тиков)
}
```

Результаты работы программы:

```
A2: S=0, F=1492, L=1492
A1: S=0, F=0, L=0
A0: S=0, F=1492, L=1492
```

Примечательно, что прикладная программа дополняется лишь декларацией объектов требуемых классов. Вызов функций класса в таких случаях неявным образом планируется компилятором.

Варианты синтаксиса неявного вызова конструктора при декларации объекта:

```
имя_класса имя_объекта;
имя_класса имя_объекта(параметры_конструктора);
имя_класса имя_объекта=имя_объекта_того_же_класса;
```

(здесь «имя_класса» совпадает с именем конструктора).

Выбор подходящего конструктора среди имеющихся в классе производится по правилам использования переопределенных функций:

```
class X {
   // ...
   X(); // конструктор по умолчанию
   X(какие_нибудь_параметры); // конструктор общего вида
   X(X&); // конструктор копирования
};
```

Конструктор вида X() иногда называют конструктором по умолчанию, а конструктор X(X&) – конструктором копирования. При отсутствии таких конструкторов в определении класса автоматически создается их стандартный аналог.

Рассмотрим примеры использования конструкторов.

```
#include <stdio.h>
#include <string.h>
```

```
class complex {
   double re, im;
   void display(int i) {
      print();
      printf(" [%d]",i);
   }
public:
   complex() { // конструктор по умолчанию
      re=im=0;
      display(0);
   }
   complex(complex &x) { // конструктор копирования
      re=x.re, im=x.im;
      display(1);
   }
   complex(int rei, int imi) {
      re=rei, im=imi;
      display(2);
   }
   complex(int rei) {
      re=rei, im=0;
      display(3);
   } // ...
   void print() {
      printf("\n (%.0lf, %.0lf)",re,im);
   }
};

 void main () {
   complex a;  // complex()
   complex b=complex(1,2); // complex(int,int)
   complex c(b);  // complex(complex &)
   complex d=b;  // complex(complex &)
   complex e(5);  // complex(int)
   a.print();
   b.print();
   c.print();
   d.print();
   e.print();
   a=d; // операция присваивания
   d=complex(3,4);  // присваивание complex(int,int)
   a.print();
   d.print();
 }
```

Результаты работы программы:

```
(0, 0) [0]
(1, 2) [2]
(1, 2) [1]
(1, 2) [1]
(5, 0) [3]
(0, 0)
(1, 2)
(1, 2)
(1, 2)
```

```
(5, 0)
(3, 4) [2]
(1, 2)
(3, 4)
```

Явный вызов конструктора соответствует операции приведения типа в функциональной форме записи.

2.6. Конструкторы вложенных классов

Объект любого класса может быть элементом другого класса. Это вытекает из наследованного из языка С отсутствия ограничений на тип элементов структуры. В таких случаях иногда возникает необходимость задания конструктором включающего класса параметров конструкторов элементов включаемых классов.

Схема описания связи параметров конструкторов:

```
class X { // описание включаемого класса
   // ...
   X (par_X); // конструктор с параметром
};

class Y { // описание включающего класса
   // ...
   // описание элементов включаемых классов
   X elem_1; // параметры здесь указать нельзя,
   X elem_2; // т. к. это только описание класса, а
   // ... // параметры относятся к определяемым объектам
   // определение конструктора включающего класса
   Y (par_Y_1):elem_1(par_X_1), elem_2(par_X_2) { // ... }
};
```

В **определении** конструктора включающего класса после символа «:» записывается в произвольном порядке список элементов включаемых классов с указанием в круглых скобках параметров их конструкторов. Конструкторы включаемых элементов будут вызваны перед конструктором включающего класса. Порядок вызова конструкторов элементов не определен, поэтому список параметров конструкторов не должен содержать взаимосвязанные выражения присваивания.

Пример программы:

```
#include <stdio.h>
#include <string.h>
class string {
   char *buffer; int length;
public:
   string(int n=80) {
      buffer=new char[length=n];
      buffer[0]=0;
   }
```

```
   ~string() {
      delete buffer;
   }
   void status(char *x) {
      printf("\n%s: B=\"%s\", L=%d",x,buffer,length);
   }
};
class primer {
   string s1; string s2;
public:
   primer(int, int);
   void test() {
      s1.status("s1");
      s2.status("s2");
   }
};
primer::primer(int l1, int l2): s1(l1), s2(l2) {
   printf("\n* %d %d",l1,l2);
}

void main () {
   primer exam(10,20);
   exam.test();
}
```

Результаты работы программы:

```
* 10 20
s1: B="", L=10
s2: B="", L=20
```

В качестве представителей вложенных классов могут рассматриваться элементы базовых типов, поэтому определение конструктора в классе

```
class alpha {
   int i;
   clock_t s;
public:
   alpha():i(0),s(clock()) { printf("\n i=%d, s=%d",i,s); }
   ~alpha() { printf("\n l=%d",clock()-s); }
};
```

эквивалентно следующему варианту с операторами присваивания:

```
alpha() {
   i=0;
   s=clock();
   printf("\n i=%d, s=%d",i,s);
}
```

Этим обстоятельством можно воспользоваться для повышения лаконичности исходного текста программы.

2.7. Конструирование массивов объектов

Особенность создания массивов объектов любого класса – раздельный вызов конструктора для каждого элемента массива. При необходимости декларации массива объектов некоторого класса в таком классе должен быть предусмотрен конструктор с пустым списком параметров. Используемые в подобных ситуациях параметры приходится передавать через глобальные переменные либо статические элементы данных класса.

Деструктор также должен быть вызван для каждого элемента массива. Для массивов, размещенных без использования оператора new, это делается неявно. Однако для массивов в свободной памяти неявный вызов невозможен – компилятор не отличает указатель на один объект от указателя на первый элемент массива объектов. Уничтожение объектов массива в свободной памяти должно проводиться оператором delete[] с явным указанием признака массива – пары квадратных скобок.

Пример работы с массивами объектов класса:

```
#include <stdio.h>
#include <string.h>
class simple {
   char *name;
public:
   simple(); // описание конструктора по умолчанию
   simple(char *x) { // конструктор с параметром
      printf("\nСоздание объекта \"%s\"",x);
      name=new char[strlen(x)+1];
      strcpy(name,x);
   }
   ~simple() { // деструктор
      printf("\nУничтожение объекта \"%s\"",name);
      delete name;
   }
     void print() { // отображение объекта
      printf("\n***  объект \"%s\" существует",name);
   }
};

simple::simple() { // определение конструктора по умолчанию
   static int i=0;
   if ((name=new char[80])!=NULL) {
      sprintf(name,"Элемент %d",i++);
      printf("\n %s создан",name);
   }
}

simple global("global");

void main () {
   printf("\n* Начало блока программы");
 // Работа с массивом фиксированной размерности
   simple a[3]; // размерность массива должна быть
                // задана константным выражением
```

```
    for (int i=0; i<SIZEOF(A)/sizeof(*a); a[i++].print())
  // Работа с массивом переменной размерности
    int n=2; // значение размерности массива
    simple *b=new simple[n];
    for (i=0;  i<N; b[i++].print());
    delete[] b;
  }
```

Результаты работы программы:

```
Создание объекта "global"
* Начало блока программы
Элемент 0 создан
Элемент 1 создан
Элемент 2 создан
***  объект "Элемент 0" существует
***  объект "Элемент 1" существует
***  объект "Элемент 2" существует
Элемент 3 создан
Элемент 4 создан
***  объект "Элемент 3" существует
***  объект "Элемент 4" существует
Уничтожение объекта "Элемент 3"
Уничтожение объекта "Элемент 4"
Уничтожение объекта "Элемент 0"
Уничтожение объекта "Элемент 1"
Уничтожение объекта "Элемент 2"
Уничтожение объекта "global"
```

Можно заметить, что элементы массива обрабатываются **последовательно**.

Очевидно, что запрет создания массива объектов некоторого класса можно установить формальным объявлением конструктора с пустым списком параметров, пустым телом и атрибутом доступа private.

Пример запрета создания массива объектов класса:

```
// Назначение приоритета потока

class ThreadPriorityControl {
 HANDLE CurrentThread;
 int Priority;
 ThreadPriorityControl() {} // Конструктор по умолчанию
public:

 ThreadPriorityControl(int priority):
  CurrentThread(GetCurrentThread()) {
  Priority = GetThreadPriority(CurrentThread);
  SetThreadPriority(CurrentThread,priority);
 }

 ~ThreadPriorityControl() {
  SetThreadPriority(CurrentThread,Priority);
 }
};
```

2.8. Конструирование статических объектов

Конструкторы глобальных статических объектов вызываются в порядке определения объектов перед выполнением программы, а их деструкторы – в обратной последовательности при завершении программ. Конструкторы локальных статических объектов вызываются всегда после конструкторов глобальных объектов, но порядок их вызова зависит от порядка активизации блоков. При неоднократной активизации блока с локальным статическим объектом его конструктор вызывается только при первой активизации.

Пример использования статических объектов:

```
#include <stdio.h>
class x {
   int i;
public:
   x(int j=0) { printf("\nСоздание: %d",i=j);  }
   ~x() { printf("\nУничтожение: %d",i); }
};
void f1() {
   static x a(1);
   x aa(11); // ...
}
void f2(int n) {
   static x a(n>>1); // конструктор будет вызван один раз!
   x aa(22); // ...
}
x c;

void main () {
   x b(3);
   f2(10);
   f2(20);
}
```

Результаты работы программы:

```
Создание: 0
Создание: 3
Создание: 5
Создание: 22
Уничтожение: 22
Создание: 22
Уничтожение: 22
Уничтожение: 3
Уничтожение: 0
Уничтожение: 5
```

Следует учитывать, что аварийное завершение программы, а также вызов библиотечных функций вида abort() подавляют вызов деструкторов. Рекомендуется использовать для нормального завершения программы в любой точке функцию вида exit().

3. ОПРЕДЕЛЕНИЕ ОПЕРАЦИЙ НАД ОБЪЕКТАМИ КЛАССОВ

3.1. Схема определения операций над объектами

Декларация объектов класса означает появление в программе возможности отображения некоторых сущностей реального мира. Конструкторы и деструкторы позволяют управлять созданием и уничтожением таких объектов. Любой доступ к объектам класса требует записи выражений, составляемых из символических обозначений операндов и символов операций над ними. В языке C++ наряду с обычными возможностями конструирования выражений над операндами базовых типов имеется возможность определения операций для определенных пользователей типов – классов. В результате конструирование выражений может проводиться на уровне операций над объектами, а не на уровне операций над их элементами.

По умолчанию для объектов класса определены лишь операции присваивания «=» и определения адреса «&». Определение любой операции ⊗ для некоторого класса реализуется объявлением функции с именем operator⊗. Такая функция должна иметь соответствующие смыслу операции набор и типы параметров и, как любая функция, может быть переопределенной. Рассмотрим пример:

```
class complex {
   double re, im;
public:
   complex(double r, double i) { re=r; im=i; }
   friend complex operator+(complex, complex);
   friend complex operator*(complex, complex);
};
```

Здесь функции с именами operator+ и operator* определяют новый смысл операторов «+» и «*». Такие функции можно вызывать явно, например:

```
complex x,y,z;
...
z=operator+(x,y);
x=operator+(operator*(x,z),complex(1.34,2.52));
```

Важнейшие последствия декларации подобных функций – возможность сокращения записи операций в выражениях:

```
complex x,y,z;
...
z=x+y;
x=x*z+complex(1.34,2.52);
```

Приведенные выражения приближаются в рамках синтаксических возможностей языка C к общепринятой записи комплексных выражений. При этом

действуют стандартные в языке С понятия приоритета и порядка выполнения операций. Последнее выражение эквивалентно следующему:

```
x=(x*z)+complex(1.34,2.52);
```

Определение операций должно вестись в рамках их синтаксиса для базовых типов объектов с учетом следующих ограничений:

- не допускается переопределение некоторых операций (например, в большинстве систем программирования на языке С++ запрещенными являются операции «.», «::», «.*», «?:»);
- приоритет операций и синтаксис их использования в выражениях (например арность) изменить нельзя;
- определение всех операций, кроме операции «=», действительно для производных классов.

В конкретных реализациях системы программирования С++ приведенные ограничения могут быть модифицированы. Например, переопределение операций «=», «()», «[]» и «->» может быть реализовано только функцией-элементом.

Нехватка символических конструкций для обозначения операций легко преодолевается введением набора функций.

Очевидно, что определяемыми могут быть только унарные или бинарные операции. Операции вызова функции «()» и обращения по индексу «[]» являются бинарными:

```
fun_ptr(arg_list) <=> operator()(fun_ptr,arg_list);
array_ptr[index]  <=> operator[](array_ptr,index).
```

3.2. Особенности определения операций

Имя функции operator⊗ не может быть идентификатором обычной функции по правилам языка С++. Компилятор на этапе синтаксического анализа текста программы по значению символа ⊗ выделяет типы и имена операндов соответствующей операции. Организация вызова подходящей с учетом правил переопределения функции-оператора планируется только при использовании в качестве одного из параметров объекта некоторого класса – типа вида struct, union или class, определенного пользователем (ислючение – операции new и delete).

Декларация функций-операторов возможна как вне класса, так и в классе. Разница между этими вариантами:

- возможности доступа к элементам класса – функциям вне класса доступны только его открытые элементы;
- время создания текста функции-оператора и определения класса – функция вне класса может быть создана позже (естественно требование отсутствия конфликтов переопределения функций).

```
// Пример определения функции-оператора вне класса

#include <stdio.h>
class a { // Класс без функций-операторов
   int i;
public:
   a(int j):i(j) {}
   int value() { return i; }
};

class b { // Класс без функций-операторов
   int i;
public:
   b(int j):i(j) {}
   int value() { return i; }
};

// Функции-операторы вне класса a

void operator+=(a &x, b &y) { // Вариант 1
   printf("\n x=%d y=%d",x.value(),y.value());
}

void operator+=(a &x, int y) { // Вариант 2
   printf("\n x=%d int y=%d",x.value(),y);
}

void operator+=(int x, a &y) { // Вариант 3
   printf("\n int x=%d y=%d",x, y.value());
}

void main() {
   a x(2);
   b y(5);
   x+=y; // Вариант 1
   x+=3; // Вариант 2
   6+=x; // Вариант 3 - константа слева!
   4+=y; // Вариант ?
}
```

Результаты работы программы:

```
x=2 y=5
x=2 int y=3
int x=6 y=2
x=4 y=5
```

Вопросы согласования типов подробно обсуждаются в подразд. 3.3.

Функции-операторы вне класса – основа определения операций ввода – вывода в стандартные потоки C++ для классов пользователя.

Операция над объектами любых классов может быть **определена в классе** как функцией-элементом, так и дружественной функцией. Ранее отмечалось, что функция-элемент получает неявный параметр – указатель объекта this. При определении операций этот параметр считается указателем на первый операнд.

Отсюда следует, что унарная операция ⊗ может быть определена функцией-элементом без параметров или дружественной функцией с одним параметром. Постфиксное выражение a⊗ или префиксное выражение ⊗a может интерпретироваться одним из взаимоисключающих вариантов:

```
a.operator⊗() – operator⊗ – функция-элемент;
operator⊗(a) – operator⊗ – дружественная функция.
```

Одинаковая интерпретация префиксной и постфиксной форм выражений приводит к тому, что для операций «++» и «--» формально отсутствует различие между их префиксными и постфиксными формами записи (для объектов базовых типов такое различие существенно с точки зрения результата операции). Это обстоятельство может быть учтено простым приемом – переопределением функций-операторов:

```
class X {
public:
   X& operator++();       // Префиксный оператор инкремента
   X operator++(int);     // Постфиксный оператор инкремента
   X& operator--();       // Префиксный оператор декремента
   X operator--(int);     // Постфиксный оператор декремента
 //…
};
```

Бинарная операция ⊗ может быть **определена в классе** функцией-элементом с одним параметром или дружественной функцией с двумя параметрами. В результате выражение a⊗b может интерпретироваться одним из взаимоисключающих вариантов:

```
a.operator⊗(b) – operator⊗ – функция-элемент;
operator⊗(a,b) – operator⊗ – дружественная функция.
```

Некоторые символы операций «⊗» относятся как к унарным, так и бинарным операциям:

```
class X {
//...
// Описание дружественных функций
   friend X operator-(X); // унарный минус
   friend X operator-(X,X); // бинарный минус
   friend X operator-(); // ошибка: нет операндов
   friend X operator-(X,X,X); // ошибка: тернарная операция

// Описание функций-элементов (с неявным первым параметром this)
   X* operator&(); // унарный & (определение адреса)
   X operator&(X); // бинарный & (поразрядная коньюнкция)
   X operator&(X,X); // ошибка: тернарная операция
 };
```

Смысл определяемых операций, в том числе и результат, должен быть полностью задан пользователем. Например, при переопределении операции присваивания проверка того, что первый операнд является переменной, не выполняется. Разработчик класса должен учесть подобные особенности подбором подходящего варианта функции и типов ее параметров:

```
#include <stdio.h>
class x {
   int i;
public:
   x(int j) {
      printf("\nC %d",i=j);
   }
   ~x() {
      printf("\nY %d",i);
   }
// Ассоциативное присваивание объектов класса x

   x& operator=(x& b) {
      if (i==b.i) printf("\n %d==%d !!",i,i);
      else printf("\n %d <== %d ?",i,b.i), i=b.i;
      return *this;
   }
// Ассоциативное присваивание целых чисел объектам класса x
   x& operator=(int i) {
      if (this->i==i) printf("\n %d==%d !?",i,i);
      else printf("\n %d <<= %d ?",this->i,i), this->i=i;
      return *this;
   }
   // Неассоциативное присваивание чисел с плавающей точкой
   // объектам класса x
   void operator=(float f) {
      if (this->i==f) printf("\n %d==%f !?",i,f);
      else printf("\n %d <<= %f ?",this->i,f), this->i=f;
   }
};

void main () {
   x a(0);
   x b(1);
   x c(2);
   a=b=c=x(3)=4; // ассоциативное присваивание
   x(4)=a;
   x(5)=3;
   a=float(6.0); // неассоциативное присваивание
}
```

Результаты работы программы:

```
C 0
C 1
C 2
C 3
3 <<= 4 ?
2 <== 4 ?
```

```
1 <== 4 ?
0 <== 4 ?
C 4
4==4 !!
C 5
5 <<= 3 ?
4 <<= 6.000000 ?
Y 3
Y 4
Y 4
Y 4
Y 4
Y 6
```

Функция operator⊗ должна быть элементом класса или иметь хотя бы один параметр – объект класса (исключение – функции, переопределяющие операции new и delete). Дополнительное ограничение – нельзя определить функцию оператора, которая оперирует исключительно с указателями. Отсюда следует:

- защита любых выражений с символом операции «⊗» от неуместного применения функций вида operator⊗ гарантируется правилами переопределения функций;
- если первый операнд операции в классе не является объектом класса, то определить такую операцию можно только дружественной функцией.

Интерпретации операций над объектами класса не предполагают учет фундаментальных свойств базовых типов. Например, коммутативность операции «*» влечет эквивалентность выражений obj_1*obj_2 и obj_2*obj_1, но, как в приведенном ранее примере определения операции присваивания, к объектам определенного пользователем класса это не относится. Аналогичным образом не учитывается взаимозависимость некоторых операций языка С. Например, выражение obj_1*=obj_2 для объектов класса не эквивалентно выражению obj_1=obj_1*obj_2.

Пример программы с определением операций:

```
#include <stdio.h>
struct integer {
   int i;
   integer operator=(int j) {
      i=j;
      return *this;
   }
   friend  integer operator*(integer, int);
   friend integer operator*(int, integer);
   friend integer& operator*=(integer&, int);
};
integer operator*(integer x, int j) {
   integer y;
   y.i=x.i*j;
   return y;
}
integer operator*(int j, integer x) {
```

```
   return operator*(x,j);
}
integer& operator*=(integer& x, int j) {
   x.i*=j;
   return x;
}

void main() {
   integer a, b, c, d;
   a=1;
   b=2;
   c=b*2;
   d=3*c;
   printf("\n a=%d b=%d c=%d d=%d",a.i,b.i,c.i,d.i);
   d*=4;
   printf("\n d=%d",d.i); (a*=4)*=2;
   printf("\n a=%d",a.i);
}
```

Результаты работы программы:

```
a=1 b=2 c=4 d=12
d=48
a=8
```

Функции-операторы не могут иметь параметры по умолчанию.

Определение операций над объектами класса и объектами базовых типов может проводиться с учетом явных и неявных преобразований типа.

3.3. Способы согласования типов

Определение операций над объектами разных классов часто связано с необходимостью преобразования типов. Например, для коммутативных по характеру операций над арифметическими объектами прямолинейное определение функций-операторов для всех возможных комбинаций операндов не только обременительно, но и неэффективно – можно сократить объем программ за счет использования неявных преобразований. Неявными преобразованиями удобно пользоваться в операциях присваивания, операторах инициализации, а также для преобразования фактических параметров и возвращаемых значений функций.

Имеется два пути согласования типов объектов:

- создание из объектов других типов объекта класса явным либо неявным вызовом его конструктора (неявно может вызываться только конструктор с единственным параметром);
- определение в классе операции приведения к другому типу.

Например, определим фрагмент класса complex, предусматривающий операции и с данными типа double:

```
class complex {
```

```
   double re, im;
public:
   complex(double r, double i) { re=r; im=i; }

   friend complex operator+(complex, complex);
   friend complex operator+(complex, double);
   friend complex operator+(double, complex);

   friend complex operator-(complex, complex);
   friend complex operator-(complex, double);
   friend complex operator-(double, complex);

   friend complex operator*(complex, complex);
   friend complex operator*(complex, double);
   friend complex operator*(double, complex);

   // ...
};
```

Исключить потребность в определении нескольких функций можно определением конструктора, который по заданному объекту типа double создает объект типа complex. Здесь это можно осуществить двумя способами:

а) явное определение двух конструкторов

```
...
complex(double r, double i) { re=r; im=i; }
complex(double r) { re=r; im=0; }
```

б) определение конструктора с параметром по умолчанию

```
...
complex(double r, double i = 0) { re=r; im=i; }
...
```

Используя второй способ, можно представить ранее записанный фрагмент класса complex в виде:

```
class complex {
   double re, im;
public:
   complex(double r, double i=0) { re=r; im=i; }
   friend complex operator+(complex, complex);
   friend complex operator-(complex, complex);
   friend complex operator*(complex, complex);
   // ...
};
```

Конструктор вызывается при возможности построения объекта своего класса из определенных его параметрами объектов. Такая возможность определяется последовательным сопоставлением типов фактических параметров объектов и формальных параметров конструктора (правилами использования переопределенных функций):

1) точное совпадение типов;

2) совпадения после применения стандартных преобразований;

3) уникальное совпадение после применения явно определенных преобразований.

Неудача сопоставления влечет выдачу компилятором сообщения о необходимости явного вызова конструктора.

Отсюда следует, что последний вариант класса complex допускает конструирование объектов для любых сочетаний арифметических данных:

```
complex x1 = complex(123,456); // явный вызов конструктора
complex x2 = 234.6; // неявный вызов конструктора
void f(complex);
//...
f(10);   // f(complex(10));
x1=x2*5; // x1=operator*(x2,complex(double(5),double(0)))
```

Попытки подбора других конструкторов для преобразования параметров в промежуточные типы, приемлемые для некоторых имеющихся конструкторов, не предпринимаются:

```
class X { /* ... */ X(int); }
class Y { /* ... */ Y(X); }
// ...
Y z=23; // ошибка: Y(X(23)) не планируется
Y z=X(23); // правильный явный вызов конструктора
```

Пример нарушения уникальности совпадения типов параметров:

```
class x { /* ... */ x(int); x(char*); };
class y { /* ... */ y(int); };
class z { /* ... */ z(x); };
void f(x);
void f(y);

f(1); // неоднозначность: f(x(1)) или f(y(1)) ?
f(x(1)); // правильный явный вызов конструктора
f(y(1)); // правильный явный вызов конструктора
```

Преобразование типа конструктором характеризуется следующими очевидными ограничениями:

- невозможно преобразование объекта определенного пользователем класса в объект базового типа (такие типы не являются классами);
- описание преобразования из нового типа в старый требует изменения описания старого типа.

Указанные ограничения снимаются определением в классе X функции-элемента вида operator type(), выполняющей преобразование объекта класса X в объекты типа type. Здесь type – имя типа без модификаторов «[]» или «()». Схема введения функции преобразования:

```
class X { // …
 operator int();
};
```

```
X a,b; int i;

// Эквивалентные выражения

i=int(b); // явное функциональное приведение типа
i=(int)b; // явное традиционное приведение типа
i=b; // неявное приведение типа

// Другие примеры неявного приведения типа

i=(a)? a+1:0;
while (a||b) { /* ... */ }
```

Таким образом, подобно конструктору, функция преобразования X::operator type() может вызываться неявно. Явный ее вызов возможен посредством записи операции приведения типа в синтаксических формах type(e) или (type)e, где e – выражение типа X. Преобразования объектов в базовые типы данных позволяют воспользоваться управляющими операторами языка C++ в следующем стиле:

```
while (cin>>x) cout<<x;
```

Здесь cin – стандартный поток ввода – объект класса istream, cout – стандартный поток вывода – объект класса ostream, «>>» и «<<» – обозначения операций ввода и вывода. Операция ввода cin>>x возвращает объект istream&, который неявно преобразуется к целому значению, отражающему результат ввода.

Определение преобразования из одного типа в другой подобным образом может быть связано с потерей информации, но позволяет получать укрупненную оценку состояния объекта. Использование конструктора, наоборот, означает детальное представление состояния объекта по входным параметрам.

Рассмотрим пример программы с явным определением операций преобразования типов.

```
#include <stdio.h>
class cx {
   friend class cy;
   double re, im;
public:
   cx(double r=0, double i=0) { re=r, im=i; }
   void print() {
      printf("\n R=%lf, I=%lf",re,im);
   }
   int nonzero() { return(re||im); }
};

class cy {
   double re, im;
public:
   cy(double r, double i=0) { re=r, im=i; }

   cy(cx x) { this->re=x.re, this->im=x.im; }
   operator cx() {
```

```
      printf("\n***");
      cx x;
      x.re=re,
      x.im=im;
      return x;
   }

   operator int() {
      return(re||im);
   }

   void print() {
      printf("\n R=%lf, I=%lf",re,im);
   }
};

void main() {
   cx a=1;
   cx b=1.13;
   cx c(' ');
   cy d(a);
   a=d;
   a.print();
   if (a.nonzero()) printf("\n a!=0...");
   b.print();
   c.print();
   d.print();
   if (d) printf("\n d!=0...");
}
```

Результаты работы программы:

```
***
 R=1.000000, I=0.000000
 a!=0...
 R=1.130000, I=0.000000
 R=32.000000, I=0.000000
 R=1.000000, I=0.000000
 d!=0...
```

3.4. Особенности использования ссылочных типов

Опеределение бинарных операций для относительно больших объектов целесообразно выполнить с назначением операндам ссылочных типов. Это позволит избежать бесполезных пересылок полей операндов. Традиционные для языка С указатели использовать нельзя из-за запрета определения операций над указателями.

Рассмотрим пример класса:

```
class matrix {
   double m[10][256];
public:
   matrix();
```

```
    friend matrix operator+(matrix&, matrix&);
    friend matrix operator*(matrix&, matrix&);
    //...
};
```

Операцию сложения можно определить так:

```
matrix operator+(matrix& x, matrix& y) {
   matrix sum;
   for (int i=0; i<10; i++)
      for (int j=0; j<256; j++)
         sum.m[i][j]=y.m[i][j]+y.m[i][j];
   return sum;
}
```

Функция operator+ обращается к операндам через ссылки, но результат возвращает посредством копирования поля объекта. Принципиально возможна ссылочная схема передачи результата:

```
class matrix {
   // ...
   friend matrix& operator+(matrix&, matrix&);
   friend matrix& operator*(matrix&, matrix&);
};
```

Здесь возникают сложности с выделением памяти для результата:

• поле результата не может иметь атрибут auto, т. к. ссылка на него будет возвращена вызывающей функции;

• результат нежелательно размещать и в статической памяти, так операция сложения в выражении может использоваться неоднократно;

• размещение результата в свободной памяти потребует отдельной операции ее освобождения.

Отсюда следует, что в большинстве случаев копирование результата оказывается наиболее приемлемым решением по времени выполнения, размеру программы и простоте программирования. Тем не менее в некоторых ситуациях по смыслу операции можно возвратить ссылку на известный объект, например, один из операндов (см. пример определения операции ассоциативного присваивания).

3.5. Особенности операций присваивания и инициализации

Рассматриваемые операции практически всегда неизбежны в любой программе, где объявляются объекты класса. Особого внимания здесь требуют объекты с нетривиальным распределением памяти, для которых имеющееся по умолчанию побитовое копирование полей данных существенно искажает отображение «объект-память». Например, рассмотрим простейший класс строковых данных:

```
struct string {
   char* p;  // указатель строки
   int size; // размер поля строки
   string(int number) { p = new char[size=number]; }
   ~string() { delete p; }
};
```

Пример программы с последствиями:

```
void f1() {
   string s1(10);
   string s2(20);
   //...
   s1 = s2;
}
```

Здесь будет создано две строки, но присваивание s1=s2 приведет к дублированию указателя на буфер строки s2. При выходе из функции f1() для объектов s1 и s2 автоматически будет вызываться деструктор, который будет уничтожать один и тот же массив символов с непредсказуемо аварийными последствиями. Правильным решением здесь является явное определение операции присваивания объектов типа string:

```
struct string {
   char* p;  // указатель строки
   int size; // размер поля строки
   string(int number) { p = new char[size=number]; }
   ~string() { delete p; }

   // Определение операции неассоциативного присваивания
   void operator=(string& y) {
      if (this!=&y) { // защита от излишней операции
         delete p; // удаление буфера левого операнда
         p=new char[size=y.size];
         strcpy(p,y.p);
      }
   }
}
```

Однако возможные последствия использования класса строковых данных предвосхищены еще не полностью:

```
void f2() {
   string s1(10);
   string s2=s1;  // инициализация побитовым копированием
   //...
}
```

Операция присваивания применяется только для созданных объектов, а здесь требуется наличие конструктора копирования, учитывающего особенности распределения памяти:

```
struct string {
    char* p;  // указатель строки
    int size; // размер поля строки
    string(int number) { p = new char[size=number]; }
    ~string() { delete p; }

    void operator=(string&); // неассоциативное присваивание
    string(string& y) { // конструктор копирования
       p=new char[size=y.size];
       strcpy(p,y.p);
    }
 };
```

Таким образом, если класс X использует нетривиальное управление памятью, то в общем случае должен быть определен полный комплект функций для гарантированного исключения побитового копирования объектов:

```
class X {
    // ...
    X(something);  // конструктор – создание объекта
    X(X&);         // конструктор копирования

    operator=(X&); // переопределение операции присваивания

    ~X(); // деструктор
 };
```

Класс с подобным набором функций-элементов иногда называют **полным**. Кроме операции инициализации, конструктор копирования может использоваться в общем случае при передаче фактических параметров и возвращаемого результата функции.

3.6. Пример переопределения операции индексации

Смысл индексов для объектов класса можно переопределить функцией operator[]. Индекс, задаваемый вторым параметром функции operator[], может иметь любой тип. Операция индексации удобна для определения ассоциативных связей между объектами разных типов.

Перепишем пример программы для подсчета числа вхождений строк в ассоциативный массив (см. подразд. 1.6).

```
#include <string.h>
#include <stdio.h>
struct pair {
   char* name; // указатель строки
   int val; // количество повторений строки
};

class list {
   pair* vec; // указатель массива строк
   int max; // текущий размер массива
```

```
   int free; // индекс свободного элемента
public:
   list(int size=16) { // конструктор
      free=0;
      vec=new pair[max=(size<16)? 16:size];
   }
   int& operator[](char*);
   void print_all();
};

int& list::operator[](char* p) {
   pair* q;
   for (q=vec+free; vec!=q--; )
      if (!strcmp(p,q->name))
         return q->val;
   if (free==max) { // переполнение: вектор увеличивается
      pair* nvec = new pair[max<<1];
      for (int i=0; i<max; i++) *(nvec+i)=*(vec+i);
         delete vec;
      vec=nvec;
      max<<=1;
   }
   q=vec+free++;
   q->name = new char[strlen(p)+1];
   strcpy(q->name,p);
   q->val=0; // начальное значение: 0
   return q-> val;
}

void list::print_all() { // Вывод ассоц. массива
   for (int i=0; i < free; i++)
      printf("\n %s %d",vec[i].name,vec[i].val);
}

// Подсчет количества повторения вводимых строк
void main() {
   char buf[256]; // Буфер ввода строки
   list vector; // Список введенных строк

// Ввод списка строк
   while (printf("\n?? "), scanf("%s",buf)>0)
   vector[buf]++;

// Отображение списка строк

   vector.print_all();
}
```

В приведенной программе размер вектора увеличивается в 2 раза при обнаружении ситуации переполнения текущего буферного массива.

3.7. Пример переопределения операции вызова функции

Операция вызова функции в форме записи fun_ptr(arg_list) рассматривается как бинарная операция, которую можно переопределить функцией operator(). Список параметров этой функции формируется по общим правилам передачи параметров, а тип результата определяется смыслом операции. Например, пусть имеем класс

```
class X {
 void operator()() { /* .... */ }
 //...
};
```

Здесь операция вызова функции переопределена функцией без параметров и возвращаемого значения. В результате после определения множества объектов класса X, например, оператором

```
X x1, x2, x2;
```

имеется возможность записи выражений вида

```
x1(); x2(); x3();
```

Внешне эти выражения напоминают традиционный вызов трех функций, но транслятор интерпретирует их следующим образом:

```
x1.operator()(); x2.operator()(); x3.operator()();
```

Операции вызова функции могут быть переопределены:

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

class X {
public:
   X() {}
   ~X() {}
   void operator()(int x) { cout<<endl<<"x="<<x; }
   void operator()(char *x) { cout<<endl<<"x=\""<<x<<"\""; }
   void operator()(int x,char *y) {
    cout<<endl<<"x="<<x<<", y=\""<<y<<"\"";
   }
};

void main() {
   X x1,x2,x3;
   x1(1);
   x1("?");
   x1(1,"x=1 ?");
   x2(2);
   x2("??");
   x2(2,"x=2 ??");
```

```
   x3(3);
   x3("???");
   x3(3,"x=3 ???");
}
```

Результаты работы программы:

```
x=1
x="?"
x=1, y="x=1 ?"
x=2
x="??"
x=2, y="x=2 ??"
x=3
x="???"
x=3, y="x=3 ???"
```

Переопределение операции вызова функции часто полезно для классов с единственной операцией. Типичный пример таких классов – **итераторы** – вспомогательные классы для организации выборки в заданном порядке элементов некоторого множества.

Приведем пример построения итератора iterator для ранее рассмотренного класса list. Наличие итератора здесь позволит отказаться от функии-элемента print_all, что обеспечит большую гибкость в форме вывода. Очевидно, что для доступа итератора к данным класса list требуется модификация декларации этого класса:

```
class list {
   friend class iterator; // разрешение доступа
   pair* vec;
   int max, free;
public:
   list(int);
   int& operator[](char*);
};
```

Определение итератора:

```
class iterator {
   list *base;      // текущий массив list
   int   index;     // текущий индекс
public:
   iterator(list& owner) { // конструктор
      base=&owner, index=0;
   }

// Определение операции вызова функции
   pair* operator()() {
      return (index<base->free)? &base->vec[index++]: 0;
   }
};
```

Создание объекта класса iterator для некоторого объекта класса list позволит далее при помощи операции вызова функции получать указатели последовательно на все элементы типа pair. Конец последовательности индицируется возвратом пустого указателя. Перепишем программу из подразд. 3.6, используя класс iterator.

```
// Подсчет количества повторения вводимых строк

void main() {
   char buf[256]; // Буфер ввода строки
   list strings; // Список введенных строк
// Ввод списка строк
   while (printf("\n?? "), scanf("%s",buf)>0) strings[buf]++;
// Определение итератора для существующего списка строк
   iterator next(strings); // next – имя объекта
// Отображение списка строк
   pair* p; // указатель выбранного объекта
   while ((p=next())!=0) printf("\n %s %d",p->name,p->val);
}
```

Итераторы, реализованные подобным образом, предназначены для сохранения закрытых описаний состояния процесса выборки объектов. Это позволяет порождать без особых усилий требуемое количество независимых итерационных процессов. Такие процессы естественно связываются с блоками программы автоматическим вызовом конструкторов и деструкторов итераторов. В последней функции main итератор next создан после формирования списка строк и будет существовать до конца программы. Очевидно, что после завершения выборки объектов активный итератор должен до уничтожения возвращать соответствующий этому состоянию признак.

Объект, используемый как функция, называют **функтором**. Функторы позволяют заменить понятия функций обратного вызова (делегатов) в языках без переопределения операций:

```
/* функция обратного вызова */
    int compare(int A, int B) {  return A < B; }
/* объявление функции сортировки */
    void sort_ints(int* a, int n, int (*cmpfunc)(int, int));

    int items[] = {4, 3, 1, 2};
    sort_ints(items, sizeof(items)/sizeof(int), compare);

  struct compare_class {
    bool operator()(int A, int B) { return (A < B); }
  };

// объявление функции сортировки
   template <class ComparisonFunctor>
    void sort_ints(int* a, int n, ComparisonFunctor c);

   compare_class functor;
   sort_ints(items, sizeof(items)/sizeof(int), functor);
```

4. ПРОИЗВОДНЫЕ КЛАССЫ

4.1. Понятие производного класса

Производный класс – расширение существующих классов, называемых в этом случае **базовыми**. Любой класс, не являющийся объединением (union), может служить базовым. В отличие от вложенных классов производный класс определяется независимо от его базовых классов, включая и этап компиляции.

Вспомним синтаксис декларации класса в языке C++:

```
вид_класса имя_класса {
   декларация_элементов_класса
};
```

(между символами «}» и «;» может размещаться список имен определяемых объектов класса с использованием, по необходимости, операции инициализации; элемент «вид_класса» может принимать значения struct, union или class).

Синтаксис декларации производного класса:

```
вид_класса имя_производного_класса:список_базовых_классов {
   декларация_элементов_производного_класса
};
```

Отличие описания производного класса от обычного состоит в дополнительном элементе «список_базовых_классов», размещаемом после имени декларируемого производного класса и символа «:». Элемент «список_базовых_классов» содержит разделенные запятыми атрибуты наследования и имена базовых классов. Атрибут наследования базового класса задается ключевыми словами private, protected или public. Он может быть опущен при учете умалчиваемого атрибута доступа для соответствующего производному классу атрибута «вид_класса» (struct – public, class – private). Пример декларации иерархии производных классов:

```
class b1 { /* ... */ };

class b2 { /* ... */ };

class d0: public b1,b2 { // class d0: public b1, private b2
       //...
};

struct d1: d0 { // struct d1: public d0
       //...
};
```

Говорят, что производный класс наследует все элементы базовых классов, т. е. определение объекта такого класса означает создание и соответствующих

ему элементов базовых классов. Однако право функций доступа производного класса на использование элементов его базовых классов зависит от атрибута наследования базового класса (табл. 2).

Таблица 2

Схема изменения атрибутов доступа к элементам базового класса в производном классе

Доступ в базовом классе	Атрибут наследования базового класса	Доступ в производном классе
Public Protected Private	Public	Public Protected нет доступа
Public Protected Private	Protected	Protected Protected нет доступа
Public Protected Private	Private	Private Private нет доступа

Из приведенной схемы следует, что производный класс с атрибутом наследования базового класса private ограничивает свободу доступа к элементам базового класса. Комбинируя атрибуты элементов и базовых классов, можно получать разные степени защиты наследуемых элементов класса. Скрытые элементы базовых классов всегда остаются скрытыми. Однако для дружественных функций класса ограничений на доступ к скрытым элементам нет.

Пусть данные о студентах представляются в виде

```
struct student {  // описание студента
   char   *name;  // фамилия, имя, отчество
   char     sex;  // пол
   char   *address;  // адрес
   //...
};
```

Понятие учебной группы может быть определено линейным списком, элементы которого представлены классом

```
class group_member: student {
   student *next;   // указатель следующего студента
   //...
};
```

Здесь отражено только простейшее отношение вложенности раздельно определенных понятий student и group_member. Производный класс может отражать сложные неоднородные отношения между объектами разных классов.

В зависимости от количества базовых классов, использованных при определении производного класса, различают **простое** и **множественное наследование**. Простое наследование соответствует случаю единственного базового

класса. Множественное наследование формально в C++ разрешено, но в некоторых реальных системах классов, использующих механизм динамической идентификации типа, – не допускается.

4.2. Виртуальные базовые классы

Производный класс может быть базовым классом, что означает возможность рекурсивного определения иерархии классов. При этом любой базовый класс присутствует в производном классе один раз. Более общая сетевая структура не может быть задана прямолинейно. Например, следующее определение синтаксически неверно:

```
class BASE { /* ... */ };
class DERIVED: BASE, BASE { /* ... */ };
```

Однако имеется косвенная возможность неоднократного включения некоторого базового класса в производный:

```
class DERIVE_1: public BASE { /* ... */ };

class DERIVE_2: public BASE { /* ... */ };

class DERIVE12: public DERIVE_1,
                public DERIVE_2 { /* ... */ };
```

Объект класса DERIVE12 будет включать 2 экземпляра объектов класса BASE. Для обращения к отдельным экземплярам придется использовать операцию привязки (разрешения области видимости) «::».

Если по некоторым соображениям в объекте производного класса необходимо иметь всего 1 экземпляр объекта базового класса, то следует объявить такие базовые классы **виртуальными**:

```
class Derive_1: virtual public BASE { /* ... */ };
class Derive_2: virtual public BASE { /* ... */ };
```

Объект производного класса следующего уровня, например,

```
class Derive12: public Derive_1,
                public Derive_2 { /* ... */ };
```

будет включать по одному экземпляру объекта виртуального базового класса BASE. В результате открывается возможность отражения сложных сетевых отношений между объектами. Наследование виртуальных базовых классов реализуется включением в объект производного класса скрытого для пользователя указателя на объект виртуального базового класса.

Атрибут virtual не мешает независимому объявлению объектов классов Derive_1 и Derive_2 (для каждого из них объект базового класса BASE также

требуется в одном экземпляре). К элементам виртуального базового класса можно обращаться без указания имени его производного класса.

4.3. Конструкторы и деструкторы производных классов

Производные классы могут иметь конструкторы и деструкторы. Если базовые классы имеют конструкторы, то они должны в общем случае вызываться при конструировании объектов производного класса. Семантически это подобно ранее рассмотренной схеме взаимодействия конструкторов вложенных классов. Связь конструктора производного класса с конструкторами базовых классов выполняется по следующей схеме:

```
class B1 { // описание базового класса
   // ...
   B1 (par_B1); // конструктор с параметром
};
class B2 { // описание базового класса
   // ...
   B2 (par_B2); // конструктор с параметром
};
//...
class BN { // описание базового класса
   // ...
   BN (par_BN); // конструктор с параметром
};
// Описание производного класса
 class D_1_N: B1, B2,... BN {
  // Определение конструктора производного класса
    D_1_N (par_D_1_N):B1(arg_B1), B2(arg_B2),...BN(arg_BN) {
      // ...
   }
   // ...
};
```

В **определении** конструктора производного класса после символа «:» записывается в произвольном порядке список операций вызова конструкторов базовых классов (будем называть такой список списком конструктора). Элементы списка разделяются запятыми. Конструкторы объектов базовых классов будут вызваны перед конструктором производного класса. Порядок вызова таких конструкторов не определен, поэтому в списках их аргументов не рекомендуется использовать взаимосвязанные выражения присваивания. В общем случае списки аргументов конструкторов базовых классов могут включать константы, глобальные параметры и/или параметры из списка аргументов конструктора производного класса. Если производный класс включает другие классы, то список его конструктора дополняется обращением к конструкторам элементов включаемых классов (см. подразд. 2.6). Рассмотрим пример производного класса с включением класса:

```
#include <string.h>
#include <stdio.h>
class base_1 {
   int a;
protected:
   int b;
public:
   base_1(int x, int y) { // конструктор base_1
      a=x, b=y;
      printf("\nbase_1: %d%d",a,b);
   }
   ~base_1() { // деструктор base_1
      printf("\n~base_1 (%d,%d)",a,b);
   }

   void print_1() { // отображение объекта класса base_1
      printf("\nbase_1: a=%d b=%d !",a,b);
   }
};

class base_2 {
protected:
   int c;
public:
   base_2(int x) { // конструктор base_2
      c=x;
      printf("\nbase_2: %d",c);
   }
   ~base_2() { // деструктор base_2
      printf("\n~base_2 (%d)",c);
   }

   void print_2() { // отображение объекта класса base_2
      printf("\nbase_2: c=%d !",c);
   }
};

class derive: public base_1, private base_2 {
   int d;
   base_1 e; // элемент вложенного класса
public:
   derive(int x, int y, int z): // конструктор derive
   base_1(x,y), // связь с конструктором base_1
   base_2(z), // связь с конструктором base_2
   e(x+1,y+1) { // связь с конструктором элемента "e"
      d=x;
      printf("\nderive: %d %d %d",d,b,c);
   }

   ~derive() { // деструктор derive
      printf("\n~derive (%d)",d);
   }

   void print_d() { // отображение объекта класса derive
      e.print_1();
      printf(" - derive: d=%d !",d);
```

```
    }
};

void main() {
   derive D(1,2,3); // определение объекта класса derive
   D.print_1();
   D.print_2();
   D.print_d();
}
```

Результаты работы программы:

```
base_1: 1 2
base_2: 3
base_1: 2 3
derive: 1 2 3
base_1: a=1 b=2 !
base_2: c=3 !
base_1: a=2 b=3 ! - derive: d=1 !
~derive (1)
~base_1 (2,3)
~base_2 (3)
 ~base_1 (1,2)
```

Таким образом, объекты производного класса конструируются снизу вверх – сначала объекты базовых классов, затем элементы вложенных классов, и наконец собственно производный класс. Уничтожение объектов производится в обратном порядке.

Связь конструктора производного класса с конструкторами виртуальных базовых классов описывается таким же образом:

```
class BASE { /* ... */ };

class D__1: virtual public BASE { /* ... */ };

class D__2: virtual public BASE { /* ... */ };

class D_12: public D__1, public D__2 {
   // ...
   D12( /* ... */ ): D__1( /* ... */ ),
                     D__2( /* ... */ ),
                     BASE( /* ... */ ) { /* ... */ }
};
```

Список конструктора производного класса здесь дополняется обращением к конструктору виртуального базового класса. Особенность языка C++ – конструкторы виртуальных базовых классов вызываются перед остальными конструкторами.

Рассмотрим простой пример использования виртуальных базовых классов в задаче представления данных о стоимости товара в упаковке.

Пусть набор базовых классов отражает понятия:

price – стоимость (цена, единица измерения);
unit – товар (наименование, стоимость);
box – упаковка (наименование, стоимость).

Образуем производный класс unit_box – товар в упаковке (наименование, стоимость), где стоимость отражает общую цену, а ее разделение на составляющие не требуется по условию задачи. Обьявление класса price виртуальным для классов unit и box позволит объединить его экземпляры в классе unit_box.

Пример программы представления сведений о товарах:

```
#include <string.h>
#include <stdio.h>
struct price { // описание стоимости
   int val; // стоимость
   char type; // единица измерения
   price(int vx, char v) {
      val=vx, type=v;
   }
};

struct unit: virtual public price { // описание устройства
   char name[20]; // наименование
   unit(char *x, int vx=0, char v=' '):price(vx,v) {
      strcpy(name,x);
   }
   void print() {
      printf("\nUnit %s %d %c",name,val,type);
   }
};

struct box: virtual public price { // описание упаковки
   char name[20]; // наименование
   box(char *x, int vx=0, char v=' '):price(vx,v) {
      strcpy(name,x);
   }
   void print() {
      printf("\nBox %s %d %c",name,val,type);
   }
};

// Описание устройства в упаковке с полной стоимостью

struct unit_box: public unit, public box {
   char name[40]; // наименование товара в упаковке
   unit_box(char *x, int vx, // наименование и цена товара
            char *y, int vy, // наименование и цена упаковки
            char *z, // общее наименование товара в упаковке
            char v): // единица измерения цены
      unit(x), box(y), price(vx+vy,v)
   { strcpy(name,z); }
   void print() {
      printf("\nUNIT_BOX %s,",name);
      printf("\nBOX %s,\nUNIT %s,",box::name,unit::name);
      printf("\nPRICE %d %d %d %c",
box::val,unit::val,val,type);
```

```
    }
};

void main() {
    unit_box tv_set("телевизор",1000, "упаковка",
                    10, "телевизор в упаковке",'p');
    tv_set.print();
    box b("b-111",111,'$');
    unit u("u-222",222,'Y');
    b.print();
    u.print();
}
```

Результаты работы программы:

```
UNIT_BOX телевизор в упаковке,
BOX упаковка,
UNIT телевизор,
PRICE 1010 1010 1010 p
Box b-111 111 $
Unit u-222 222 Y
```

Если список конструктора производного класса не содержит операций вызова конструкторов некоторых базовых классов, то автоматически будут вызываться конструкторы таких классов по умолчанию (без параметров).

4.4. Взаимосвязь компонент производного и базовых классов

Пусть определен производный класс derived:

```
class base { /* ... */ };
class derived : public base { /* ... */ };
```

Объект класса derived включает на уровне хранения в памяти компоненту класса base. Непосредственный доступ к элементам класса по их именам не позволяет на этапе исполнения программы учесть взаимосвязь компонент на уровне класса. Если не использовать такую взаимосвязь, то производный класс может рассматриваться лишь как средство сокращения описания объектов.

Наиболее конструктивным применением производного класса является возможность решения во время выполнения программы двух задач:

- выделение объекта базового класса из объекта производного класса;
- выделение объекта производного класса по его объекту базового класса.

Перечисленные задачи решаются **косвенным** обращением к компонентам класса по указателям (ссылкам) с учетом следующих правил:

- указатель на объект класса derived можно присваивать указателю на объект класса base без явного преобразования типа;
- преобразование указателя на объект класса base в указатель на объект класса derived должно быть явным.

Пример применения этих правил:

```
derived D;
base* pb = &D;       // неявное преобразование указателя
derived* pd = pb;    // ошибка: base* не есть derived*
pd = (derived *)pb; // явное преобразование
base &rb = D;        // неявное преобразование ссылки
```

Рассмотрим результаты эксперимента над указателями в системе программирования C++:

```
#include <stdio.h>
class base_1 {
   int x;
public:
   int y;
};

class base_2 {
   char x[13];
public:
   long y;
};

class base_3 {
   long x;
public:
   long y;
};

class derive: public base_1, private base_2, base_3 {
   long z;
public:
   long d;
};

void main() { // Model SMALL
   printf("\nРазмеры объектов базовых и производного класса\n");

   printf("\n|base_1|=%d",sizeof(base_1));
   printf("\n|base_2|=%d",sizeof(base_2));
   printf("\n|base_3|=%d",sizeof(base_3));

   printf("\n|derive|=%d",sizeof(derive));

  derive objd, *p_d=&objd;

   base_1 *pb1=&objd; // Указателю на базовый класс
   base_2 *pb2=&objd; // можно присвоить значение указателя
   base_3 *pb3=&objd; // производного класса

   printf("\n\nВыделение объектов базовых классов\n");
   printf("\np_d: %u, %u",p_d,p_d+1);
   printf("\npb1: %u, %u",pb1,pb1+1);
   printf("\npb2: %u, %u",pb2,pb2+1);
```

```
    printf("\npb3: %u, %u",pb3,pb3+1);

    printf("\n\nПолучение адреса производного класса\n");

    p_d=(derive *)pb1;
    printf("\np_d==pb1? %u - %u",p_d,pb1);
    p_d=(derive *)pb2;
    printf("\np_d==pb2? %u - %u",p_d,pb2);
    p_d=(derive *)pb3;
    printf("\np_d==pb3? %u - %u",p_d,pb3);
}
```

Результаты работы программы:

```
Размеры объектов базовых и производного класса
base_1|=4 |base_2|=17 |base_3|=8 |derive|=37
Выделение объектов базовых классов
p_d: 65484, 65521 pb1: 65484, 65488 pb2: 65488, 65505
pb3: 65505, 65513
Получение адреса производного класса
p_d==pb1? 65484 - 65484 p_d==pb2? 65484 - 65488 p_d==pb3?
65484 - 65505
```

Таким образом, по адресу объекта производного класса можно непосредственно получить адрес любого его объекта базового класса. Обратная операция возможна, но требует явного использования операции приведения типа. Операции инкремента или декремента указателей на объекты базовых классов по отношению их вложенности в объект производного класса выполнять бессмысленно – значение указателя базового класса не будет адресовать объект такого класса в смежных объектах производного класса.

4.5. Виртуальные функции

Определение связанных совокупностей объектов необязательно одинаковых классов приходится проводить посредством использования указателей. Косвенная адресация объектов обеспечивает максимальную гибкость программы как по отношению к набору связываемых объектов, так и времени связывания.

Сочетание понятий указателя и производного класса в языке C++ позволяет обеспечить возможность связывания объектов классов как по данным, так и по функциям доступа вне зависимости от времени компиляции и/или выполнения. Ранее отмечалось, что между объектами базового и производного классов по указателям может быть установлено симметричное отношение. Отсюда следует, например, что функции базовых классов могут применяться к компонентам таких классов в объектах позднее определяемых производных классов. В результате, используя возможность переопределения функций, легко построить альтернативный интерфейс доступа к базовому классу.

Программирование на основе использования связей между классами в общем случае порождает проблему определения типа производного класса по известному указателю на объект базового класса.

Возможны 3 основных способа решения этой проблемы:

1) адресация объектов единственного типа;

2) включение в базовый класс поля признака типа;

3) использование **виртуальных** функций.

Обычно указатели на базовые классы используются при разработке так называемых контейнерных (вмещающих) классов. Примеры контейнерных классов: множества, векторы, списки, таблицы и т. п.

В этом случае первое решение дает однородные списки объектов одного типа. Второе и третье решения пригодны для построения неоднородных списков объектов различных типов. Третье решение – специальный вариант второго решения, когда классификационным признаком выступает контролируемый компилятором тип объектов.

Виртуальными называют функции-элементы базового класса, объявляемые с атрибутом virtual, которые переопределены в производных классах. Атрибут virtual предписывает при вызове подобных функций через указатель выбирать экземпляр функции, соответствующий **типу** адресуемого объекта. Обычные переопределенные функции базового и производного классов выбираются при вызове через указатель по **типу указателя.**

Пример использования виртуальных функций:

```
#include <stdio.h>

#define P(X) printf("\n\n* %s\n",X)

struct base {
   int n;
   base(int x=0) { n=x; }

   virtual void f() { // определение виртуальной функции
      printf("\nbase %d",n);
   }
};

struct de_1: base {
   de_1(int x=0):base(x) { } // пустое тело допустимо !
   void f() { // переопределение виртуальной функции
      printf("\nde_1 %d",n);
   }
};

struct de_2: base {
   de_2(int x=0):base(x) { } // пустое тело допустимо !
   void f() { // переопределение виртуальной функции
      printf("\nde_2 %d",n);
   }
};
```

```
void main() {
   de_1 D1(1);
   de_2 D2(2);
   base B1(0);

   P("Неявный выбор функции по типу объекта");
   B1.f();
   D1.f();
   D2.f();

   P("Явный выбор функции по типу объекта");
   B1.base::f();
   D1.de_1::f();
   D2.de_2::f();

   P("Явный выбор функции базового класса");
   B1.base::f();
   D1.base::f();
   D2.base::f();

   P("Косвенный выбор по указателю на базовый класс (1)");
   base *p; // указатель на базовый класс
   p=&B1; p->f();
   p=&D1; p->f();
   p=&D2; p->f();

   P("Косвенный выбор по указателю на базовый класс (2)");
   ((base *)&B1)->f();
   ((base *)&D1)->f();
   ((base *)&D2)->f();
}
```

Результаты работы программы:

```
* Неявный выбор функции по типу объекта
base 0 de_1 1 de_2 2
* Явный выбор функции по типу  объекта
base 0 de_1 1 de_2 2
* Явный выбор функции базового класса
base 0 base 1 base 2
* Косвенный выбор по указателю на базовый класс (1)
base 0 de_1 1 de_2 2
* Косвенный выбор по указателю на базовый класс (2)
base 0 de_1 1 de_2 2
```

Следующий пример программы демонстрирует применение виртуальных функций для классификации объектов, помещаемых в контейнер:

```
#include <stdio.h>
struct base {
   virtual void print() {
       printf("\n base");
   }
};
```

```
void print_set(base **x, int n) {
   for (int i=0; i<n; i++) {
      x[i]->print();
   }
}

class der_1: public base {
public:
   void print() {
      printf("\n der_1");
   }
};

class der_2: public base {
public:
   void print() {
      printf("\n der_2");
   }
};

class der_3: public base {
public:
   void print() {
      printf("\n der_3");
   }
};

void main() {
   base *x[6];
   x[0]=new base;
   x[1]=new der_1;
   x[2]=new der_2;
   x[3]=new der_3;
   x[4]=new der_1;
   x[5]=new der_2;
   print_set(x,sizeof(x)/sizeof(*x));
}
```

Результаты работы программы:;

```
base
der_1
der_2
der_3
der_1
der_2
```

Таким образом, виртуальные функции позволяют автоматизировать процесс классификации объектов в неоднородной иерархии классов. Существенно, что базовые классы и их виртуальные функции-элементы могут быть созданы и откомпилированы до определения производных классов. Предоставляемая тем самым возможность эволюционного развития программы в большинстве случаев оказывается важнейшей ее качественной характеристикой.

Пример использования виртуальных функций:

```
#include <stdio.h>
struct base {
   int i;
   base(int j):i(j) {};
   virtual void print(base &x) { printf("\nbase(%d)",x.i); }
   void operator!() { print(*this); }
};

struct deri: public base {
   deri(int j):base(j) {}
   void print(base &x) { printf("\nderi(%d)",x.i); }
};

void main() {
   deri x(5);
   x.print(x);
   !x;
}
```

Результаты работы программы:

```
deri(5)
deri(5)
```

Проведем эксперимент по оценке влияния атрибута virtual на размер объекта:

```
#include <stdio.h>
struct b1 { // базовый класс с виртуальной функцией
   int i;
   virtual void f() { i=0; }
};
class d1: b1 {
public:
   void f() { i=0; }
};

struct b2 { // базовый класс без виртуальной функции
   int i;
   void f() { i=0; }
};
class d2: b2 {
public:
   void f() { i=0; }
};

#define P(X,Y) printf("\n sizeof(%s)=%d",X,sizeof(Y));
void main() {
   P("b1",b1);
   P("d1",d1);
   P("b2",b2);
   P("d2",d2);
}
```

Результаты работы программы:

```
sizeof(b1)=8
sizeof(d1)=8
sizeof(b2)=4
sizeof(d2)=4
```

Недостатки использования виртуальных функций:

• увеличение размера объекта из-за необходимости хранения указателя функции;

• обращение к виртуальным функциям через указатель означает отказ от возможности встраивания функции, что снижает быстродействие программы.

Явное указание имени класса при обращении к функции посредством операции «::» подавляет действие атрибута virtual. Этим можно воспользоваться для программирования задач с вызовом одной виртуальной функции из другой. Явное указание имени класса позволяет отказаться от накладных расходов на динамическое определение типа объекта в ситуациях, когда между базовыми классами по сути задачи существует жесткая взаимосвязь. Последнее особенно важно в критических по быстродействию программах, когда желательно использовать механизм встраивания функции.

Обязательным требованием для работы механизма виртуальности является совпадение типа возвращаемого значения, числа и типов параметров виртуальной функции, а также атрибута const (после списка параметров) в базовом классе и во всех производных классах. В противном случае функция рассматривается как переопределенная.

Рассмотрим пример применения виртуальных функций в задаче формирования неоднородного списка студентов некоторой группы (рис. 1).

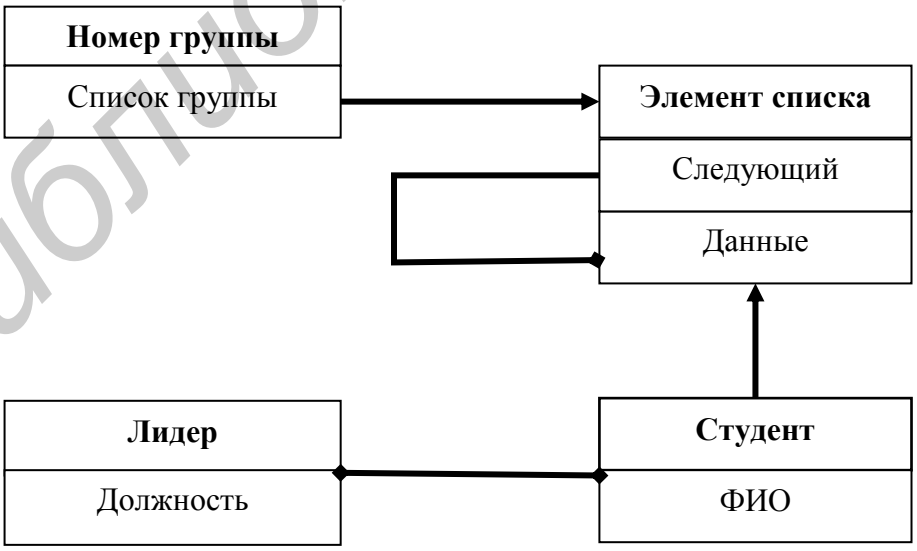

Рис. 1. Модель неоднородного списка студентов группы

Пример программы формирования списка студентов:

```
#include <stdio.h>
#include <string.h>
class student { // описание студента
   char *name; // имя студента
public:
   student(char *x) { // конструктор
      name=new char[strlen(x)+1];
      strcpy(name,x);
   }
   ~student() { delete name; } // деструктор

   virtual void f() { // отображение объекта
      printf("\n%s",name);
   }
};

class lider: public student { // описание активиста
   char titul[10]; // наименование должности
public:
   lider(char *x, char *y):student(x) { // конструктор
      strcpy(titul,y);
   }
   void f() { // отображение объекта
      student::f(); // Явное обращение к объекту по его типу
      printf(" - %s",titul);
   }
};

struct list; // вспомогательная декларация структуры
struct list { // описание спискового класса
   list *next; // указатель следующего элемента
   void *item; // указатель поля данных
   void include(list **head, void *x) { // расширение списка
      list *y=new list;
      y->next=*head, *head=y, y->item=x;
   }
};

class group { // описание группы студентов (контейнерный класс)
   char ident[7]; // идентификатор группы
   list *head; // указатель списка студентов
public:
   group(char *x) { // конструктор группы
      strcpy(ident,x);
      head=0;
   }

   void operator+=(void *x) { // операция включения в группу
      head->include(&head,x);
   }

   void operator()(); // описание операции вызова функции
   ~group(); // описание деструктора
};
```

```
void group::operator()() { // операция отображения группы
   printf("\nСписок группы \"%s\"\n",ident);
   for (list *x=head; x!=0; x=x->next) ((student *)x->item)->f();
    // Косвенное обращение к объекту
}

group::~group() { // деструктор
   list *temp; while (head!=0)
   { // уничтожение элементов списка
      head=(temp=head)->next;
      delete temp;
   }
}

void main() {
   group acs("Team-1"); // Определение группы

 // Включение в группу рядовых студентов

   acs+=new student("st1");
   acs+=new student("st2");
   acs+=new student("st3");
   acs+=new student("st4");

 // Включение в группу студентов-активистов

   acs+=new lider("sh1","shief");
   acs+=new lider("sh2","viceshief");
   acs+=new lider("sh3","profboss");

 // Отображение состава группы

   acs();
}
```

Результаты работы программы:

```
Список группы "Team-1"
sh3 - profboss
sh2 - viceshief
sh1 - shief
st4
st3
st2
st1
```

Виртуальные функции позволяют достичь гибкости привязки функций к объектам за счет незначительных потерь памяти и быстродействия. Виртуальной может объявляться любая функция-элемент класса, за исключением конструктора. Виртуальные деструкторы подробно рассмотрены в подразд. 6.7. Класс, где есть хотя бы одна виртуальная функция, называют **полиморфным.**

Для вызова виртуальной функции явное применение указателя либо ссылки не всегда требуется.

Пример программы с использованием полиморфизма:

```
#include <stdio.h>
#include <stdlib.h>

// Иерархия без виртуальных функций

class base {
public:
   int func(int x) { return x*x; }
   int test(int x) { return func(x)<<1; }
};

class derive: public base {
public:
   int func(int x) { return x*x*x; }
};

// Иерархия с виртуальными функциями

class Base {
public:
   virtual int func(int x) { return x*x; }
   int test(int x) {
    return func(x)<<1; // return (this->func(x))<<1;
   }
   int work(int x) { return Base::func(x)<<1; }
};

class Derive: public Base {
public:
   int func(int x) { return x*x*x; }
};

void main() {
   derive d;
   Derive D;
   printf("\n %d",d.test(3));
   printf("\n %d",D.test(3));
   printf("\n %d",D.work(3));
}
```

Результаты работы программы:

```
18
54
18
```

Запрет использования виртуального конструктора легко преодолевается посредством имитации конверта, в котором лежит письмо. Класс конверта представляет собой базовый класс, содержащий указатель на объект того же базового типа. Этот указатель будет указывать на письмо, т. е. объект некоторого производного класса. Если требуется создать объект некоторого типа, то необходимо передать идентификатор этого типа в конструктор класса конверта. На основании этого идентификатора конструктор создает в динамической памяти

объект соответствующего производного класса, адрес которого становится значением указателя на письмо:

```
// Идентификаторы всех производных классов
enum VBase_ID { Der_ID1, Der_ID2, Der_ID3 };

struct VBase { // Класс конверта
    VBase(): p(0) { }
    VBase( VBase_ID id );
    virtual void info() { p->info(); }
    virtual ~VBase() { delete p;  }
  private:
    VBase* p;
};
```

4.6. Абстрактные классы

Производный класс может не иметь функции, определенной в базовом классе как виртуальная. Версия функции базового класса доступна по определению производного класса с учетом правил защиты. В некоторых случаях возникают ситуации, когда при отсутствии виртуальной функции в производном классе использование ее версии из базового класса недопустимо. В таких случаях в базовом классе достаточно объявить строго или чисто (pure) виртуальную функцию, используя синтаксис:

```
virtual тип_результата имя_функции(список_параметров)=0;
```

Версия функции с подобным прототипом должна обязательно определяться в производном классе. Любой класс, где объявлена хотя бы одна строго виртуальная функция, называется **абстрактным.**

Определение объектов абстрактного класса невозможно. Абстрактный класс можно использовать только в качестве базового для наследования невиртуальных элементов в производных классах. Однако указатель или ссылка на абстрактный класс может определяться в списках параметров или возвращаемых значений функции для адресации объектов производного класса.

Иллюстрация использования абстрактного класса:

```
#include <stdio.h>
class base { // абстрактный класс
public:
   virtual void f()=0; // строго виртуальная функция
};

class derive_1: public base {
public:
   void f() { printf("\nderive_1"); }
};
```

```
class derive_2: public base {
public:
   void f() { printf("\nderive_2"); }
};

class derive12: public derive_1 {
public:
   void f() { printf("\nderive12"); }
};

void main() {
   derive_1 D_1;
   derive_2 D_2;
   derive12 D12;
   base *pb;
   pb=&D_1;
   pb->f(); // Выбор функции класса derive_1
   pb=&D_2;
   pb->f(); // Выбор функции класса derive_2
   pb=&D12;
   pb->f(); // Выбор функции класса derive12
}
```

Результаты работы программы:

```
derive_1
derive_2
derive12
```

Следующий пример демонстрирует возможность эволюционного процесса создания программы:

```
#include <stdio.h>
class base {
protected:
   int width;
public:
   base(int n=0):width(n) {}
   virtual void print()=0;
};

class line {
   int i;
public:
   line(int j=0):i(j) {}
   void operator=(int j) { i=j; }
   void operator<<(base &x) {
      printf("\n%d) ",++i);
      x.print();
   }
};

class dw1: public base {
   int value;
public:
```

```
      virtual void print() { printf("%*d",width,value); }
      dw1(int x, int w=0):base(w),value(x) {}
};

class dw2: public base {
   int value;
public:
   virtual void print();
   dw2(int x, int w=0):base(w),value(x) {}
};

void dw2::print() {
   printf("%d ",value);
   for (int i=0; i<width; i++) printf("?");
}

void main() {
   line out;
   for (int i=0; i<5; i++) { out<<dw1(i,i); }
   out=0;
   for (i=0; i<5; i++) { out<<dw2(i,i+1); }
}
```

Результаты работы программы:

```
1) 0
2) 1
3)  2
4)   3
5)    4
1) 0 ?
2) 1 ??
3) 2 ???
4) 3 ????
5) 4 ?????
```

Понятие абстрактного класса оказывается весьма полезным средством отражения общих свойств некоторых сущностей реального мира. Создаваемая на основе абстрактного класса иерархия производных классов может оперировать с объектами, которые определяются независимо от состава операций. Например, в библиотечных классах систем C++ определена сложная иерархия классов абстрактных структур данных, отражающая понятия стека, очереди, дека, массива и т. п.

Функции манипуляции соответствующими этим понятиям объектами определены в производных классах, откомпилированы и помещены в библиотеку объектных модулей. Пользователь может определить производный класс конкретных объектов на основе базового абстрактного класса таких структур данных, после чего открывается возможность работы с любыми объектами иерархии.

5. ОБЪЕКТНО-ОРИЕНТИРОВАННЫЙ ВВОД – ВЫВОД

5.1. Классы и потоки ввода – вывода

В языках C и C++ нет собственных лингвистических конструкций ввода – вывода. Система ввода – вывода создается средствами языка в конкретной вычислительной среде. В языке C для этих целей используется процедурный подход, реализуемый библиотеками функций ввода – вывода. В языке C++ наряду с этим обычно реализуется объектно-ориентированный подход на основе библиотек классов ввода – вывода. Очевидно, что оба подхода оказываются системно-зависимыми, хотя наблюдается тенденция к стандартизации имен функций и/или классов ввода – вывода. Один из характерных принципов построения подобных систем – преемственность для совместимости старых программ с новыми системами программирования. Рассмотрим пример объектно-ориентированного подхода к построению средств ввода – вывода в системах C++ фирмы BORLAND. Схема иерархии классов ввода – вывода в Borland C++ соответствует следующему фрагменту описания:

```
// Классы потокового ввода-вывода
//
class ios { /* ... */ };
class istream: virtual public ios { /* ... */ };
class ostream: virtual public ios { /* ... */ };
class iostream: public istream, public ostream { /* ... */ };
//
// Классы предопределенных стандартных потоков ввода-вывода
//
class istream_withassign: public istream { /* ... */ };
class ostream_withassign: public ostream { /* ... */ };
class iostream_withassign: public iostream { /* ... */ };
//
// Классы файлового ввода-вывода
//
class fstreambase : virtual public ios { /* ... */ };
class ifstream: public fstreambase,public istream {/* ... */ };
class ofstream: public fstreambase,public ostream { /* ... */ };
class fstream: public fstreambase,public iostream { /* ... */ };
//
// Классы строкоориентированного ввода-вывода
//
class strstreambase: public virtual ios { /* ... */ };
class istrstream:public strstreambase,public istream {/*...*/ };
class ostrstream:public strstreambase,public ostream {/*...*/ };
class strstream:public strstreambase,public iostream {/*...*/ };
//
// Классы буферов ввода-вывода
//
class streambuf { /* ... */ };
class filebuf: public streambuf { /* ... */ };
class strstreambuf: public streambuf { /* ... */ };
```

Исходный базовый класс ios, а также большинство производных классов определены в файле iostream.h. Класс ios предназначен для отображения состояния процесса ввода – вывода. Производные классы отражают:

- направление передачи данных (i – ввод, o – вывод, io – ввод – вывод);
- вид источника данных (f – файл, str – строка символов);
- способ организации процесса обмена данными (буферизация, форматирование) и др.

Операции ввода – вывода рассматриваются на двух уровнях:

- нижний или файловый уровень – файл как именованная последовательность байтов, имеющая начало и конец;
- верхний уровень – последовательный **поток** объектов разных типов, отображаемый на файл.

Функции ввода – вывода выполняют прямое и/или обратное преобразование между уровнями. Понятие потока соответствует объекту класса с именем ...stream. Манипулирование потоками требует определения объекта соответствующего класса.

В системе C++ имеется 4 определенных по умолчанию потока:

Имя объекта	Имя класса	Аналог потока в языке C
cin	istream_withassign	stdin
cout	ostream_withassign	stdout
cerr	ostream_withassign	stderr (без буфера)
clog	ostream_withassign	stderr (с буфером)

При небходимости работы с файлами требуется явное определение объектов соответствующих классов.

5.2. Ввод – вывод данных базовых типов

Для любых потоков определены две операции:

«>>» – извлечение (extraction) данных из потока;

«<<» – запись (insertion) данных в поток.

Функции-элементы, определяющие перечисленные операции, многократно переопределены для приема всех основных типов данных.

Пример декларации операций вывода:

```
class ostream {
   // ...
public:
   ostream& operator<< (  signed char);
   ostream& operator<< (unsigned char);
   ostream& operator<< (short);
   ostream& operator<< (unsigned short);
   ostream& operator<< (int);
```

```
    ostream& operator<< (unsigned int);
    ostream& operator<< (long);
    ostream& operator<< (unsigned long);
    ostream& operator<< (float);
    ostream& operator<< (double);
    ostream& operator<< (long double);
    ostream& operator<< (const   signed char*);
    ostream& operator<< (const unsigned char*);
    ostream& operator<< (void*);
    // ...
};
```

Пример декларации операций ввода:

```
class istream {
    // ...
public:
    istream& operator>> (  signed char*);
    istream& operator>> (unsigned char*);
    istream& operator>> (unsigned char&);
    istream& operator>> (  signed char&);
    istream& operator>> (short&);
    istream& operator>> (int&);
    istream& operator>> (long&);
    istream& operator>> (unsigned short&);
    istream& operator>> (unsigned int&);
    istream& operator>> (unsigned long&);
    istream& operator>> (float&);
    istream& operator>> (double&);
    istream& operator>> (long double&);
    // ...
};
```

Функции определения операторов ввода – вывода здесь имеют первый неявный операнд – указатель на объект класса ostream или istream и возвращают ссылку на объект того же класса. Это позволяет использовать ассоциативные операции ввода – вывода:

```
#include <iostream.h>
void main() {
    int x,y,z;
    cout<<"\n\a* x,y,z-?";
    cin>>x>>y>>z; // последовательный ввод x,y,z
    cout<<"\n data is"<<x<<y<<z; // последовательный вывод
}
```

Результаты работы программы:

```
* x,y,z-? 1 2 3
data is 123
```

Здесь возникают вопросы управления форматом ввода – вывода и контроля исключительных ситуаций.

5.3. Ввод – вывод объектов определенных пользователем классов

Не обсуждая подробности управления вводом – выводом (форматирование, исключительные ситуации и т. п.), рассмотрим стандартные схемы объектно-ориентированного программирования ввода – вывода.

Определение операций ввода – вывода для определенных пользователем классов рекомендуется проводить путем дополнительного переопределения операторов «>>» и «<<» для подходящих библиотечных классов.

Пример определения операторов ввода – вывода:

```
#include <iostream.h>
class our_class {
public:
   int data;
};

ostream& operator<<(ostream& output, our_class& object) {
   output<<"{{"<<object.data<<"}}";
   return output;
}

istream& operator>>(istream& input, our_class& object) {
   input>>object.data;
   return input;
}

void main() {
   our_class x;
   while (cout<<"\n\a* data-? ", cin>>x)
      cout<<"\n data is "<<x;
}
```

Результаты работы программы:

```
* data-? 1
data is {{1}}
* data-? 2
data is {{2}}
data-? e
```

Приведенная схема не требует вмешательства в определения библиотечных классов ввода – вывода и не зависит от версии таких классов.

5.4. Контроль исключительных ситуаций ввода – вывода

Ключевая информация о состоянии обмена любого потока предоставляется элементами базового класса ios. Функция получения информации о состоянии потока:

```
int rdstate();  // чтение битов состояния
```

Возможные состояния потока перечислены в классе ios:

```
enum io_state { // биты состояния потока
   goodbit  = 0x00,  // все в порядке – ошибок нет
   eofbit   = 0x01,  // конец файла
   failbit  = 0x02,  // ошибка без потери символов
   badbit   = 0x04,  // ошибка с потерей данных
   hardfail = 0x80   // грубая ошибка
};
```

(обращение к элементам такого перечисления возможно посредством использования операции «::», например ios::goodbit).

Функции проверки конкретных исключительных ситуаций:

```
int eof();  // конец файла
int fail(); // неудачная операция
int bad();  // наличие ошибок
int good(); // отсутствие ошибок
```

Смысл приведенных функций можно выявить путем анализа исходного текста определения класса ios в файле iostream.h:

```
class ios {
   // ...
public:
   int  state;       // биты состояния потока
   long x_flags;     // биты флагов форматирования
   int  x_precision; // точность вывода данных типа float
   int  x_width;     // ширина поля вывода
   int  x_fill;      // символ заполнения потока
   // ...
};
inline int ios::rdstate() { return state; }
inline int ios::eof() { return state & eofbit; }
inline int ios::fail() {
 return state & (failbit | badbit | hardfail);
}
inline int ios::bad() { return state & (badbit | hardfail); }
inline int ios::good() { return state == 0; }
inline long ios::flags() { return x_flags; }
```

Функция установки текущего состояния потока:

```
void  clear(int =0); // установка битов состояния
```

Обращение к функции clear позволяет сбросить биты ошибок или принудительно установить некоторые ситуации для последующего анализа. Нулевое значение параметра означает запрос сброса битов ошибок. Следует учитывать, что после любой ошибки операции над потоком игнорируются до сброса битов ошибок. Пример программы с контролем исключительных ситуаций:

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

```
void main() {
   int i;
   double x;

   for (;; ) {
      cout<<"\n?? ";
      cin>>x;
      switch(i=cin.rdstate()) {
           case ios::goodbit:cout<<"sin("<<x<<")="<<sin(x);
           break;
           case ios::eofbit: exit(0);
           default:cout<<"\n\a* Error "<<i;
           cin.seekg(0,ios::end); // Пропуск введенных символов
           cin.clear(); // Сброс битов ошибок
      }
   }
}
```

Результаты работы программы:

```
?? 5
sin(5)=-0.958924
?? e
* Error 2 ??^C
```

Проверка наличия ошибок возможна и посредством операторов:

```
operator void* (); // отсутствие ошибок
int operator! ();  // наличие ошибок
```

Их существование позволяет транслятору воспользоваться неявным преобразованием типа.

Пример программы с контролем глобального состояния потоков:

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

void main() {
   double x;
   cout<<"\nКосинус-калькулятор\a\n";
   while(cout<<"\n\a* x-? ") {
    if (!(cin>>x)) break;
    // аналог: if (cin>>x, cin.bad()) break;
     cout<<" cos("<<x<<")="<<cos(x);
   }
}
```

Результаты работы программы:

```
Косинус-калькулятор
* x-? 4 cos(4)=-0.653644 * x-? r
```

5.5. Форматный ввод – вывод

Базовый класс ios предоставляет возможность управления преобразованием форм представления данных с помощью флагов и параметров форматирования. Функции чтения/установки/сброса флагов форматирования:

```
long flags(); // чтение поля флагов
long flags(long); // чтение и установка поля флагов
```

Возможные флаги форматирования определены в классе ios элементами перечисления

```
enum {
 skipws    = 0x0001, // пропуск пустого пространства при вводе
 left      = 0x0002, // "левое" выравнивание при выводе
 right     = 0x0004, // "правое" выравнивание при выводе
 internal  = 0x0008, // знак или признак основания системы
                     // счисления не отделять от данных
 dec       = 0x0010, // десятичное преобразование
 oct       = 0x0020, // восьмеричное преобразование
 hex       = 0x0040, // шестнадцатеричное преобразование
 showbase  = 0x0080, // вывод признака основания
 showpoint = 0x0100, // вывод десятичной точки
 uppercase = 0x0200, // верхний регистр при выводе hex-данных
 showpos   = 0x0400, // вывод знака '+' для положительных чисел
 scientific= 0x0800, // формат вида 1.2345E2 вывода float-данных
 fixed     = 0x1000, // формат вида 123.45 вывода float-данных
 unitbuf   = 0x2000, // очистка буфера потоков после вывода
 stdio     = 0x4000  // очистка буфера потоков stdout и stderr
                     // после вывода
};
```

Для работы с флагами форматирования могут быть использованы следующие функции-элементы: long setf(long setbits, long field) – сброс флагов, соответствующим единицам в поле field, установка их по значению поля setbits и возврат первоначального поля флагов; long setf(long) – чтение и установка флагов; long unsetf(long) – чтение и сброс помеченных единицами флагов. С целью удобного доступа к некоторым связанным флагам в классе ios определены константы для второго параметра функции setf:

```
static const long basefield;   // dec | oct | hex
static const long adjustfield; // left | right | internal
static const long floatfield;  // scientific | fixed
```

Таким образом, доступ к объектам класса ios можно программировать без записи числовых констант:

```
cout.setf(ios::left);
cout.setf(ios::right | ios::fixed);
cout.setf(ios::right + ios::hex + ios::showbase);
cin.setf(ios::oct,ios::basefield);
```

Кроме вызова функции setf, существует другой способ установки основания системы счисления. Классы istream и ostream включают дополнительное переопределение операторов ввода и вывода:

```
istream& istream::operator>>(istream& (*f)(istream&)){ /*...*/ }
ostream& ostream::operator<<(ostream& (*f)(ostream&)){ /*...*/ }
```

В файле iostream.h описаны так называемые функции-манипуляторы:

```
ios& dec(ios&); // установка десятичного преобразования
ios& hex(ios&); // установка шестнадцатеричного преобразования
ios& oct(ios&); // установка восьмеричного преобразования
```

Отсюда следует, что основание системы исчисления может быть установлено непосредственно операторами ввода и вывода:

```
int data;
cin>>hex>>data; // ввод шестнадцатеричного числа
cin>>dec; // переключение на ввод десятичных чисел
// ...
cout<<dec<<data; // вывод десятичного числа
cout<<hex<<data; // вывод шестнадцатеричного числа
```

Последний способ отличается лаконичностью записи выражений.

Примеры других функций-манипуляторов:

```
ostream& endl(ostream&);  // переход на новую строку
ostream& ends(ostream&);  // добавление символа конца строки
ostream& flush(ostream&); // вывод содержимого буфера потока
istream& ws(istream&);    // пропуск "пробелов" при вводе
```

Рассмотрим набор функций для чтения и/или установки количественных параметров формата – ширины поля, символа заполнения неиспользованных элементов поля и точности вывода данных с плавающей точкой.

Функции чтения/установки ширины поля вывода:

```
int width() – чтение текущей ширины;
int width(int) – чтение текущей и установка новой ширины.
```

Установленное значение ширины действительно только для очередной операции вывода. Если ширина поля недостаточна, то она расширяется для предотвращения потери данных. По умолчанию ширина поля нулевая.

Функции чтения/установки символа заполнения потока:

```
char fill() – чтение символа заполнения;
char fill(char) – чтение и установка символа заполнения.
```

Функции чтения/установки точности вывода данных с плавающей точкой:

```
int precision() – чтение точности вывода;
```

```
int precision(int) – чтение и установка точности вывода.
```

Пример программы с форматным вводом – выводом:

```
#include <iostream.h>

// Процедура отображения целого числа

void intout(int x) {
 long old_flg=cout.flags(); // Сохранение флагов форматирования
 cout<<endl<<"Dec: "<<dec<<x;
 cout.setf(ios::showbase);
 cout<<endl<<"Oct: "<<oct<<x;
 cout<<endl<<"Hex: "<<hex<<x;
 cout.setf(old_flg); // Восстановление флагов форматирования
}

// Программа демонстрации форматного ввода-вывода

void main() {
 int i;
 double d;
 cout<<endl<<"Dec-? ";
 cin>>dec>>i;
 intout(i);
 cout<<endl<<"Oct-? ";
 cin>>oct>>i;
 intout(i);
 cout<<endl<<"Hex-? ";
 cin>>hex>>i;
 intout(i);

 cout<<endl<<"Double-? ";
 cin>>d;
 cout<<d;

 cout.width(20), cout.setf(ios::left), cout.fill('_');
 cout<<endl<<d;
 cout.width(20), cout.precision(3);
 cout<<endl<<d;
 cout.width(20), cout.setf(ios::right), cout.unsetf(ios::left);
 cout<<endl<<"End";
}
```

Результаты работы программы:

```
Dec-? 10
 Dec: 10
 Oct: 012
 Hex: 0xa
 Oct-? 10
 Dec: 8
 Oct: 010
 Hex: 0x8
 Hex-? 10
 Dec: 16
```

```
Oct: 020
Hex: 0x10
Double-? 123.456789
123.456789
123.456789__________
123.457_____________
 ___________________End
```

Дополнительные возможности по управлению форматированием предоставляют функции-манипуляторы, определенные в файле iomanip.h:

- setbase(int base) – установка основания системы исчисления base = 0, 8, 10 или 16 (0 – десятичное основание при выводе, а при вводе основание определяется автоматически);
- resetiosflags(long flags) – сброс указываемых флагов форматирования;
- setiosflags(long flags) – установка указываемых флагов форматирования;
- setfill(int fill) – установка указанного символа заполнения потока;
- setprecision(int prec) – установка точности вывода чисел с плавающей точкой;
- setw(int width) – установка ширины поля ввода или вывода.

Пример использования манипуляторов потока:

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

void main(void) {
   int i=123;
   float f=123.456789;
   cout<<endl<<setbase(10)<<setw(10)<<i;
   cout<<endl<<setbase(8)<<i;
   cout<<endl<<setfill('.')<<setw(10)<<i;
   cout<<endl<<setprecision(3)<<setw(10)<<f;
   cout<<endl<<setw(10)<<f<<setw(8)<<i;
}
```

Результаты работы программы:

```
123
173
.......173
...123.457
...123.457.....173
```

Технология программирования собственных манипуляторов:

```
#include <iostream.h>

ostream& func(ostream & x) {
   cout<<endl<<"func";
   return x;
}

class my_manip {
```

```
   int n;
public:
   my_manip(int i):n(i) {}
   friend ostream& operator<<(ostream & x, my_manip y) {
      x<<endl<<y.n<<" my_manip ?";
      return x;
   }
};

void main() {
   cout<<func<<my_manip(1999)<<func;
}
```

Результаты работы программы:

```
func
1999 my_manip
func
```

5.6. Бесформатный ввод – вывод

Представленные здесь функции предназначены для обмена между потоками и полями данных, рассматриваемыми как последовательности байтов.

Ввод строки символов с указанием максимальной длины и символа-разделителя (символ-разделитель в строку не включается, строка завершается нулевым символом):

```
istream& get(  signed char*, int, char = '\n');
istream& get(unsigned char*, int, char = '\n');
```

Ввод заданного количества символов:

```
istream& read(  signed char*, int);
istream& read(unsigned char*, int);
```

Ввод строки символов с указанием максимальной длины и символа-разделителя включением символа-разделителя:

```
istream& getline(  signed char*, int, char = '\n');
istream& getline(unsigned char*, int, char = '\n');
```

Ввод одиночных символов:

```
int get();
istream& get(unsigned char&);
istream& get(  signed char&);
```

Последний вариант функций ввода символа допускает использование результата операции ее вызова в сложных выражениях ассоциативного типа.

Пример программы:

```
#include <iostream.h>

void main() {
   char i;
   int j;
   (cin>>ws).get(i)>>j; cout<<i<<" "<<j;
}
```

Результаты работы программы:

```
r   123
r 123
```

Чтение очередного символа без извлечения из потока:

```
int peek();
```

Количество символов, полученных последней операцией ввода:

```
int gcount();
```

Возврат символа в поток:

```
istream& putback(char);
```

Пропуск заданного количества символов с остановкой в позиции разделителя:

```
istream& ignore(int =1, int = EOF);
```

Вывод одиночного символа:

```
ostream& put(char);
```

Вывод массива символов указанной длины:

```
ostream& write(const   signed char*, int);
ostream& write(const unsigned char*, int);
```

Ввод символов в заданный буфер потока, исключая символ-разделитель:

```
istream& get(streambuf&, char = '\n');
```

Требуемый здесь указатель на буфер потока можно получить обращением к функции

```
streambuf *ios::rdbuf();
```

5.7. Управление позиционированием потока

Если рассматривать поток как последовательно заполняемый или читаемый массив символов, то текущая **позиция** соответствует индексу очередного символа, участвующего в операции обмена данными. Манипулирование позициями позволяет часто упростить и повысить эффективность операций ввода – вывода за счет отказа от последовательного сканирования потока.

Рассмотрим набор функций-элементов, оперирующих с текущей позицией потока. Файл iostream.h содержит определение двух типов данных:

```
typedef long streampos; // абсолютная позиция потока
typedef long streamoff; // относительная позиция потока
```

Чтение значения указателя текущей позиции потока:

```
streampos tellg(); // поток ввода istream
streampos tellp(); // поток вывода ostream
```

Установка указателя текущей позиции потока:

```
istream& seekg(streampos); // поток ввода istream
ostream& seekp(streampos); // поток вывода ostream
```

Относительное смещение указателя текущей позиции потока:

```
istream& seekg(streamoff,seek_dir); // поток ввода istream
ostream& seekp(streamoff,seek_dir); // поток вывода ostream
```

Определение символических констант направления смещения:

```
enum seek_dir { // Точка отсчета направления смещения
   beg=0, // начало потока
   cur=1, // текущая позиция
   end=2  // конец потока
 };
```

Пример программы с позиционированием потока cin рассматривался в подразд. 5.4:

```
//...
cin.seekg(0,ios::end); // Пропуск введенных символов
//...
```

Здесь запрашивается переход в конечную позицию потока введенных символов для игнорирования ошибочных символов. Относительное смещение позиции допускается не для всех типов потоков.

5.8. Связанные потоки

С любым из потоков ввода может ассоциироваться так называемый **связанный** выходной поток (tied stream), который позволяет выполнять логически корректные двунаправленные операции ввода – вывода: если начинается ввод данных, то содержимое буфера вывода будет обязательно выдано на носитель данных. По умолчанию с потоком cin связан поток cout. В любой момент времени с потоком ввода может быть связан только один поток вывода. Управление связыванием реализуется функциями: ostream* tie() – чтение указателя связанного потока; ostream* tie(ostream*) – чтение указателя связанного потока и установка новой связи по указателю потока вывода.

Вызов функции tie с нулевым параметром приводит к разрыву связи потоков. Например, оператор

```
ostream *old_flw=cin.tie(0);
```

устанавливает асинхронный режим обмена через потоки cin и cout. Восстановление связи можно выполнить оператором

```
cin.tie(old_flw);
```

Упрощенный вариант приведенных действий:

```
cin.tie(0);
cin.tie(&cout);
```

5.9. Создание и организация взаимодействия потоков

До сих пор в примерах программ использовались предопределенные по умолчанию потоки. Практика программирования часто требует использования потоков, ассоциированных не только со стандартными файлами ввода – вывода, но и с другими файлами либо областями памяти. Важнейший элемент любого потока – буфер потока, представляемый объектами производных классов на основе базового класса streambuf (см. подразд. 5.1). Именно на элементах такого класса определены функции стратегии обмена данными. Буфер потока среди элементов данных включает, кроме буфера ввода – вывода, набор переменных состояния обмена. Конструирование потока означает его ассоциацию с буфером подходящего типа. В системах программирования C++ фирмы Borland поддерживается 2 вида специализированных высокоуровневых потоков обмена:

– файловые потоки (классы fstream, ifsream, ostream – определены в файле fstream.h);

– строкоориентированные потоки (классы strstream, istrstream, ostrstream – определены в файле strstrea.h).

Если специализация потокового объекта нежелательна, то можно использовать универсальные потоки обмена (классы iostream, istream и ostream опре-

делены в файле iostrstream.h). Однако в этом случае требуется явное установление ассоциации со специализированным буфером. Такую ассоциацию можно установить следующими способами:

– конструирование потока со ссылкой на буфер требуемого типа;

– выполнение операций обмена данными некоторого потока с данными во внешнем для этого потока буфере требуемого типа.

Реализация первого способа требует знакомства с конструкторами используемых классов (приходится обращаться к исходному тексту определения классов). Например, класс iostream имеет конструктор

```
iostream(streambuf *);
```

Если требуется связь такого потока с файлом, то необходима ссылка на буфер потока – объект класса filebuf.

Набор конструкторов класса filebuf:

```
// Буфер потока без связи с файлом
filebuf();
// Буфер потока со связью с открытым файлом
filebuf(int fd);
// Буфер потока со связью с открытым
// файлом и назначенным буфером обмена
filebuf(int fd, char *buf_addr, int buffer_len);
```

(здесь fd – дескриптор файла, а buf_addr и buf_len) – указатель и размер буфера обмена).

Идентификатором файла на внешнем носителе как набора данных является его имя, но после открытия файла достаточна более лаконичная схема идентификации дескриптором. Система управления вводом – выводом строит взаимно-однозначное соответствие между именем файла и его дескриптором. Для любого файла, открытого обычным образом, например,

```
FILE *pf=fopen("a:user.txt","wt");
```

его дескриптор определяется макросом fileno:

```
int fd=fileno(pf);
```

Приведенный выше набор конструкторов позволяет создать буфер потока с различной исходной привязкой к файлу. Рассмотрим набор функций-элементов класса filebuf, которые позволяют управлять такой привязкой во времени. Открытие потока и связываемого с ним файла с именем fname, режимом открытия mode и признаком защиты файла prot:

```
filebuf* open(const char *fname, int mode,
              int prot=filebuf::openprot);
```

(признак защиты имет смысл в многозадачных системах). Возможные режимы открытия потоков и связанных с ними файлов определены в классе ios:

```
enum open_mode { // Виды открытия потоков
   in        = 0x01, // открытие для чтения
   out       = 0x02, // открытие для записи
   ate       = 0x04, // переход к концу файла при открытии
   app       = 0x08, // открытие для добавления
   trunc     = 0x10, // открытие для усечения файла
   nocreate  = 0x20, // ошибка, если при открытии файла нет
   noreplace = 0x40, // ошибка, если при открытии файл есть
   binary    = 0x80  // открытие двоичного файла
};
```

Очевидно, что можно объединять несколько признаков операцией дизъюнкции или суммирования. Например, выражение

```
ios::nocreate | ios::out | ios::binary
```

запрашивает открытие нового двоичного файла для вывода. Операция открытия обязательна, если предполагается использование буфера потока, созданного конструктором класса filebuf без параметров. Другие функции для управления связью «поток – файл» посредством переназначения буфера потока:

```
// Очистка буфера и закрытие файла
filebuf* close();
// Проверка открытия файла
int is_open();
// Получение дескриптора, связанного с буфером файла
int fd();
// Подключение буфера потока к открытому файлу
filebuf* attach(int fd);
```

Операции обмена с потоком возможны лишь в состоянии, когда его буфер связан с открытым любым способом файлом. Для контроля исключительных ситуаций, управления позиционированием и выполнения операций ввода – вывода могут использоваться все ранее рассмотренные функции-элементы потоковых классов. Пример программы со связыванием потока с буфером:

```
#include <fstream.h>

void main() {
   filebuf fb; // Создание файлового буфера
   if (fb.open("con",ios::in|ios::out)) { // Файл открыт-?
      iostream cio(&fb); // Создание потока ввода-вывода
      int x;
      cio<<endl<<"\ax-? ";
      cio> >x; cio<<"x="<<x;
   }
}
```

Результаты работы программы:

```
x-? 2005
x=2005
```

Пояснения к программе:

1) операторы «>>» и «<<» определены для двух разных классов (istream и ostream) поэтому запись следующего выражения ошибочна:

```
cio<<endl<<"\ax-? ">>x<<"x="<<x;
```

2) операции закрытия файла автоматически выполняются деструкторами.

Рассмотрим второй способ образования ассоциации потоков – операции ввода – вывода посредством связи буферов. Производные классы istream, ostream и iostream включают функцию-элемент для получения указателя буфера потока:

```
filebuf* rdbuf();
```

Кроме этого, имеется дополнительное переопределение операторов:

```
istream& operator>> (streambuf*);
ostream& operator<< (streambuf*);
```

Указатель типа streambuf* неявно преобразуется к указателю типа filebuf* по правилам связи указателей базовых и производных классов. Таким образом, можно динамически связать один поток с другим без дополнительной пересылки данных в памяти.

5.10. Файловый ввод – вывод

Для работы с файлами в соответствии с направлением передачи данных можно создавать объекты одного из трех определенных в файле fstream.h специализированных файловых классов:

- fstream – ввод – вывод;
- ifstream – ввод;
- ofstream – вывод.

Файл fstream.h включает директиву #include <iostream.h> для включения декларации иерархии базовых классов (см. подразд. 5.1). Рассмотрим многообразие конструкторов объектов файлового ввода – вывода, обозначая именем fstream_t любой элемент множества {fstream, ifstream, ofstream}:

– создание потока без соединения с файлом:

```
fstream_t();
```

– создание потока, соединенного с открытым файлом указанием дескриптора файла (fd):

```
fstream_t(int fd);
```

– создание потока, соединенного с открытым файлом указанием дескриптора файла (fd), указателя (buf_addr) и размера (buf_len) буфера обмена:

```
fstream_t(int fd, char *buf_addr, int buffer_len);
```

– создание потока, соединяемого с открываемым файлом указанием имени файла (fname), режима открытия (mode) и признака защиты файла (prot):

```
fstream_t(const char *fname, int mode,
              int prot=filebuf::openprot);
```

Конструктор последнего вида в файловых классах описан с разными умалчиваемыми значениями параметра mode:

```
ofstream(const char *fname,
         int mode=ios::out, int prot=filebuf::openprot);
ifstream(const char *fname,
         int mode=ios::in, int prot=filebuf::openprot);
fstream(const char *fname,
         int mode, int prot=filebuf::openprot);
```

Отсюда следует, что для конструирования потоков классов ifstream или ofstream можно указать лишь имя файла. Состав и смысл параметров конструкторов файловых классов соответствует ранее рассмотренным параметрам конструктора класса filebuf. Использование файловых классов избавляет от необходимости явного конструирования буфера потока, но схема связи потока и файла остается прежней.

Функции-элементы файловых классов выполняют:

– открытие потока и связываемого с ним файла с именем fname, режимом открытия mode и признаком защиты файла prot:

```
void open(const char *fname,int mode,int prot=filebuf::openprot);
```

– получение указателя буфера потока:

```
filebuf* rdbuf();
```

Класс filebuf не является базовым для файловых классов, поэтому его функции-элементы, пригодные для управления связью «поток-файл», доступны косвенно по указателю буфера потока.

```
#include <fstream.h>

void main() { // Пример программы с управлением буфером
   ifstream fp;
   fp.open("tst77.c"); // Открытие файла tst77.c
   if (fp.rdbuf()->is_open()) { // Файл открыт-?
    char x;
```

```
    while ((x=fp.get())!=EOF) // Копирование файла в поток cout
     cout<<x;
   }
}
```

Здесь для определения состояния открытия файла использовано косвенное обращение к буферу файла, но чтение символов реализуется наследуемой из класса istream функцией get (см. подразд. 5.1).

Пример программы без явного обращения к буферу потока:

```
#include <fstream.h>
#include <stdlib.h>

void main() {
   ofstream R("f:random");
   if (R) {
      R.setf(ios::right);
      for (int i=0; (i<100)&& R.good(); i++) {
         R.width(8);
         R<<random(32767);
      }
   }
}
```

Здесь в файл f:random будут выведены 100 псевдослучайных чисел в десятичной системе исчисления в поля шириной 8 позиций.

5.11. Строкоориентированный ввод – вывод

Поток можно ассоциировать со строкой символов некоторой длины. Это позволяет воспользоваться средствами оперирования потоками для подготовки или обработки данных в памяти. Подобные действия на процедурном уровне выполняют, например, библиотечные функции sprintf и sscanf. Для работы со строками в соответствии с направлением передачи данных можно создавать объекты одного из трех определенных в файле strstrea.h специализированных строковых классов: strstream – ввод – вывод; istrstream – ввод; ostrstream – вывод.

Файл strstrea.h включает директиву #include <iostream.h> для включения декларации иерархии базовых классов (см. подразд. 5.1). Рассмотрим многообразие конструкторов и интерфейсных функций-элементов классов строкового ввода – вывода:

```
// Класс strstream:
strstream(char *buf, int size, int mode);
char *str();
// Класс istrstream:
istrstream(char *buf);
istrstream(char *buf, int size);

// Класс ostrstream:
```

```
ostrstream(char *buf, int size, int mode=ios::out);
char *str();
int pcount();
```

Идентификация элементов определения перечисленных классов:

- buf и str – указатели буферов обмена;
- size – размер буфера;
- mode – режим открытия;
- pcount – количество символов, выведенных последней операцией.

Функции-элементы строкоориентированных классов обеспечивают контроль исключительных ситуаций, связанных с переполнением буфера. При необходимости функция str позволяет рассмотреть сформированный поток как строку или массив символов.

Приведем пример программы буферизованного вывода строк псевдослучайных чисел заданного размера:

```
#include <strstrea.h>
#include <stdlib.h>
void main() {
   char buf[65]; // буфер вывода
   ostrstream R(buf,sizeof(buf)); // поток подготовки буфера
   if (R) {
      long j;
      R.setf(ios::right);
// Цикл вывода строк
      for (int k=0; k<5; k++) {
         R.seekp(0);
// Цикл заполнения строки
         for (int i=0; R.good(); i++) {
            j=R.tellp();
            R.width(8);
            R<<rand();
         }
         R.clear();
         R.seekp(j);
         R<<ends;
         cout<<endl<<R.str();
      }
   }
}
```

Результаты работы программы:

```
   41  18467   6334    26500   19169    15724   11478    29358
24464   5705  28145    23281   22879    18492    1360     5412
22463  25047  27119    31441    7190    13985   31214    27509
26571  14779  19816    21681   19651    17995   23593     3734
 3979  21995  15561    16092   18489    11288   28466     8664
```

Здесь каждая строка будет выведена только после ее формирования в памяти. При этом в отличие от буферизации данных файловой системой ввода – вывода существует возможность вмешательства в содержимое буфера.

6. ОСОБЕННОСТИ ПРОГРАММИРОВАНИЯ НА ЯЗЫКЕ С++

6.1. Обзор операционных особенностей объектов класса

Одной из причин сложного восприятия языка С++ по сравнению с языком С даже опытными программистами является потребность учета достаточно большого количества деталей при объявлении либо использовании классов. В языке С после объявления любого объекта поведение программы полностью можно характеризовать только в терминах операций языка. В языке С++ многие действия планируются компилятором неявно, а для производных классов могут даже только обозначаться. Здесь приводятся исходные тексты заготовок определений классов, которыми можно воспользоваться при подготовке рабочего варианта определения конкретного класса с учетом рекомендаций надежного [4–5] программирования на языке С++.

```
//-------------------------------------------------------------
// Прототип определения базового класса
//-------------------------------------------------------------
class base {
   any_class obj;
public:
   virtual ~base(void);
   base(void);
   base(const base &r);
   const base &operator=(const base &r);
private:
};

//-------------------------------------------------------------
base::~base(void) {}
//-------------------------------------------------------------
inline base::base(void):obj(any_value) {}
//-------------------------------------------------------------
inline base::base(const base &r):obj(r.obj) {}
//-------------------------------------------------------------
inline const base &base::operator=(const base &r) {
   if (this!=&r) {
      obj=r.obj;
   }
   return *this;
}
//-------------------------------------------------------------
```

Обратите внимание, что определение даже почти всегда требующихся функций-элементов вынесено за пределы определения класса. Это облегчает обзор архитектуры класса и позволяет не забывать о недостатках встраивания функций.

```
//-------------------------------------------------------------
```

```
// Прототип определения производного класса
//-------------------------------------------------------------
class derived: public base {
   any_class obj;
public:
   virtual ~derived(void);
   derived(void);
   derived(const derived &r);
   const derived &operator=(const derived &r);
private:
};
//-------------------------------------------------------------
derived::~derived(void) {}
//-------------------------------------------------------------
inline derived::derived(void):
base(any_value), obj(any_value) {}
//-------------------------------------------------------------
inline derived::derived(const derived &r):
base(r), obj(r.obj) {}
//-------------------------------------------------------------
inline const derived &derived::operator=(const derived &r) {
   if (this!=&r) {
      *((base *)this)=r;
      obj=r.obj;
   }
   return *this;
}
//-------------------------------------------------------------
```

Наличие конструктора и деструктора объектов класса существенно расширяет процедурные возможности языка С. Объявление любого объекта в языке С и интервал его существования определяются лишь классом памяти, а изменение состояния объекта требует явного программирования преобразования содержимого памяти. В языке С++ организуемый компилятором неявный вызов конструктора и деструктора позволяет запрограммировать некоторые действия, выполняемые на границах интервала существования объекта. Естественно, скрываемые таким образом переходные процессы будут гарантированно реализованы для каждого объекта. Это позволяет случайно не забыть для отдельного объекта, например, освободить захваченную память, закрыть открытый файл, восстановить вектор прерываний или обработчик ошибок и т. д.

Здесь объект is_true в блоке функции main используется для управления доступом к клавиатуре.

Можно ли конструктор и деструктор вызвать повторно для существующего объекта? Если из содержательных соображений такие действия желательны, то в конкретной системе программирования С++ потребуется лишь подбор приема вызова конструктора. Дело в том, что деструктор всегда вызывается после создания объекта, поэтому фрагмент текста программы, подобный

```
class X;
//...
X object;
```

```
//...
object.X::~X();
```

не имеет логических противоречий и будет синтаксически правильным. Вызов конструктора тесно увязан с процессом выделения памяти для конструируемого объекта. Лишь некоторые компиляторы C++ разрешают явный вызов конструктора для существующего объекта:

```
//... class X;
//... X object;
//...
object.X::X();  // Borland C++ 3.1 не разрешает...
```

Однако современный язык C++ позволяет использовать так называемую помещающую форму операции new, когда оператор вида

```
new(&object) X;
```

предписывает вызов конструктора класса X применительно к существующему объекту object. Фактически есть и другой способ разрешения последнего вопроса – введение вспомогательных функций

```
class X {
    //...
 public:
    //...
    void create()  { /*...*/}
    void destroy() { /*...*/}
    //...
    X() { create(); }
    ~X() { destroy(); }
    //...
 };
 //...
 X object;
 //...
 object.create();  // object.X::X();
 object.destroy(); // object.X::~X();
```

Особую роль, как отмечалось ранее, играют конструктор копирования и функция-элемент operator=(). Их определения практически безусловно необходимы при наличии в классе указателей либо ссылок на некоторые объекты. Рекомендуемый стиль оформления таких функций отражен выше в прототипах определений классов. В частности, не следует забывать о возможности самокопирования объектов, а также необходимости инициализации компонент базового класса при конструировании объектов производного класса.

Далее в настоящем разделе обсуждаются другие особенности использования классов, в том числе и потребность использования виртуальных деструкторов.

6.2. Особенности динамического управления памятью

Возможность динамического управления памятью в C++ реализуется разрешенными для переопределения операциями new и delete, классический синтаксис которых имеет вид:

```
// Размещение объектов типа type
type *ptr=new type;
type *ptr=new type(параметры_конструктора/инициализации);
type *ptr=new type[размерность_массива];
// Удаление объектов типа type
delete ptr;   // Удаление одиночного объекта
delete[] ptr; // Удаление массива
```

Ниже будет рассмотрен расширенный вариант синтаксиса оператора new в современных версиях языка C++.

Рекомендуется пользоваться операциями new и delete вместо традиционных для языка C обращений к функциям вида malloc/free по причинам:

– результат операции new – указатель на определенный тип, а возвращаемый функцией malloc пустой указатель (void *) в языке C++ потребуется явно преобразовывать;

– операции new и delete можно переопределить как глобально для всех типов, так и локально внутри классов;

– имеется возможность централизованной обработки ошибок распределения памяти.

Операции управления памятью относятся к глобально контролируемым действиям программы. Переопределяющие такие операции функции-операторы в любом классе подразумеваются **статическими** (для других разрешенных для переопределения операций языка C++ функции-операторы статическими быть не могут).

Определенной по умолчанию операции new соответствует функция

```
void *operator new(size_t);
```

(параметр типа size_t представляет объем запрашиваемой памяти для размещения объекта).

Переопределяя глобально подобную функцию вне класса, следует учитывать, что она будет вызываться для размещения:

– одиночных объектов типов, не являющихся классами с локальными функциями operator new();

– массивов объектов любых типов.

Помещенная в определении некоторого класса локальная функция operator new() используется только для размещения одиночных объектов этого класса. Обращение из любых функций к умалчиваемой или переопределенной версии оператора new возможно посредством операции привязки:

```
class type {
```

```
      //...
      void *operator new(size_t) { };
   };
   type *ptr1=::new type;       // Объект без инициализации
   type *ptr2=::new type(...);  // Инициализируемый объект
   type *ptr3=::new type[size]; // Привязка не обязательна – мас-
сив...
```

Функция operator new() всегда должна возвращать указатель на тип void и получать первый аргумент типа size_t. В современных версиях языка C++ разрешается расширять список параметров. Тогда функция

```
void *operator new(size_t,<дополнительные_параметры>);
```

вызывается для интерпретации операций new вида

```
new(<дополнительные_фактические_параметры>) <тип>;
new(<дополнительные_фактические_параметры>) <тип>(...);
new(<дополнительные_фактические_параметры>) <тип>[...];
```

(здесь угловые скобки синтаксическими элементами не являются).

Очевидно, что тем самым открывается возможность многократного переопределения функций вида operator new().

Определенной по умолчанию операции delete может соответствовать одна из функций:

```
void operator delete(void *);
void operator delete(void *, size_t);
```

(параметр типа void * представляет указатель на удаляемый объект, а size_t – размер объекта в байтах). В системе программирования Borland C++ 3.1 локальное переопределение операции delete в классе возможно любой из приведенных версий функций-операторов, но для глобального переопределения разрешено использование только функции с единственным параметром.

Подобно операции new глобально определенная вне какого-либо класса функция operator delete() будет вызываться для удаления:

– одиночных объектов типов, не являющихся классами с локальными функциями operator delete();

– массивов объектов любых типов.

Отсюда следует, что определенная в классе функция operator delete() может удалять только одиночные объекты такого класса. Если для такого удаления желателен вызов глобального варианта функции-оператора, то необходимо использовать операцию привязки.

Следующая программа демонстрирует различные работоспособные варианты управления размещением объектов с переопределением операторов new и delete:

```
#include <stdio.h>
```

```
#include <stdlib.h>
class X {
   int item;
public:
   X():item(0) { printf("\n X() ?"); }
   X(X& x):item(x.item) { printf("\n X(X&) ??"); }
   void *operator new(size_t size) {
      void *area=malloc(size);
      printf("\nЛок. запрос %d байт. Адрес %p",size,area);
      return area;
   }
   void operator delete(void *p, size_t i) {
      printf("\nЛок. возврат по адресу %p, размер %d",p,i);
      free(p);
   }
   X(int i):item(i) { printf("\n X(%d) ???",item); }
   ~X() { printf("\n~X(%d) ??",item); }
};

void *operator new(size_t size) { // Вариант 1
   void *area=malloc(size);
   printf("\nГлоб. запрос %d байт. Адрес %p",size,area);
   return area;
}

void *operator new(size_t size,char *s,int n) { // Вариант 2
   void *area=malloc(size);
   printf("\nГлоб. запрос %d байт. Адрес %p. %s %d",
size,area,s,n);
   return area;
}

void operator delete(void *p) { // Разрешен только 1 вариант...
   printf("\nГлоб. возврат по адресу %p",p);
   free(p);
}

void TEST(int i) {
   printf("\n\nТест %d:",i);
}

void main() {
   { TEST(1);           // Вызываются ли глобальные функции
      int *p=new int;    // operator new() и
      delete p;          // operator delete() ?
   }
   { TEST(2);           // Инициализация базовых типов
      int *p=new int(1); // доступна ?
      delete p;
   }
   { TEST(3);           // Какие версии функций определяет
      int *p=::new int;  // операция глобальной привязки ?
      ::delete p;
   }
   { TEST(4);           // Что происходит при работе с
      int *p=new int[3]; // массивами базовых типов ?
```

```
      delete[] p;         // Устаревшая версия: delete[3] p;
   }
   { TEST(5);           // Некорректное управление размещением
      int *p=new int[3]; // массивов базовых типов... работает ?
      delete p;          // Правильно: delete[] p;
   }
   { TEST(6);           // Как организуется размещение одиночных
      X *p=new X;         // объектов класса с переопределением
      delete p;          // операций new и delete ?
   }
   { TEST(7);           // Как организуется размещение массивов
      X *p=new X[3];      // объектов класса с переопределением
      delete[] p;         // операций new и delete ?
   }
   { TEST(8);                  // Допустимо ли комбинирование
      X *p=::new("Тест",8) X;  // глобальных и локальных версий
      delete p;                // операторов new и delete ?
   }
   { TEST(9);           // Другой вариант предыдущего теста
      X *p=new X(9);
      ::delete p;
   }
   printf("\n\nOK ? ");
}
```

Результаты работы программы:

```
Тест 1:
Глоб. запрос 4 байт. Адрес 00430070
Глоб. возврат по адресу 00430070

Тест 2:
Глоб. запрос 4 байт. Адрес 00430070
Глоб. возврат по адресу 00430070

Тест 3:
Глоб. запрос 4 байт. Адрес 00430070
Глоб. возврат по адресу 00430070

Тест 4:
Глоб. запрос 12 байт. Адрес 00430070
Глоб. возврат по адресу 00430070

Тест 5:
Глоб. запрос 12 байт. Адрес 00430070
Глоб. возврат по адресу 00430070

Тест 6:
Лок. запрос 4 байт. Адрес 00430070
X() ?
~X(0) ??
Лок. возврат по адресу 00430070, размер 4

Тест 7:
Глоб. запрос 16 байт. Адрес 00430070
X() ?
```

```
X() ?
X() ?
~X(0) ??
~X(0) ??
~X(0) ??
Глоб. возврат по адресу 00430070

Тест 8:
Глоб. запрос 4 байт. Адрес 00430070. Тест 8
X() ?
~X(0) ??
Лок. возврат по адресу 00430070, размер 4

Тест 9:
Лок. запрос 4 байт. Адрес 00430070
X(9) ???
~X(9) ??
Глоб. возврат по адресу 00430070

ОК ?
```

Можно экспериментально проверить, что если для базовых типов фрагмент программы вида

```
{ TEST(5);           // Некорректное управление размещением
   int *p=new int[3]; // массивов базовых типов... работает ?
   delete p; }        // Правильно: delete[] p;
```

не привел к аварийному завершению программы, то синтаксически правильный фрагмент

```
{ X *p=new X[3];   // Глобальная операция new
   delete p; }     // Локальная операция delete
```

не является логически корректным и завершается в общем случае аварийно.

Реализуемая функцией вида

```
  void *operator new(size_t, void *pointer)
```

вышеупомянутая помещающая форма операции new позволяет предписать место размещения в памяти создаваемого объекта:

```
#include <new.h>
#include <stdio.h>
inline void * operator new(size_t, void *pointer) {
   return pointer;
}

class AnyClass {
public:
   AnyClass() { printf("\nИнициализация -> %p",this); }
   ~AnyClass() { printf("\nРазрушение -> %p",this); }
};
```

```
AnyClass global_object;

void main() {
   printf("\nНачало программы");
   new (&global_object) AnyClass;
   printf("\nКонец программы");
}
```

Результаты работы программы:

```
Инициализация -> 0386
Начало программы
Инициализация -> 0386
Конец программы
Разрушение -> 0386
```

Здесь операция new используется с обращением по адресу к существующему объекту global_object только для повторного вызова конструктора. Конструктор в подобных ситуациях может учесть дополнительные обстоятельства, касающиеся уточнения глобального окружения объекта. Некоторые компиляторы (например Borland C++ 4.0 и более поздние версии) разрешают переопределение операций захвата и освобождения памяти массивов [1]:

```
void *operator new[](size_t); // Размещение массивов
void operator delete[](void *); // Удаление массивов
```

В процессе выполнения стандартной операции new возможны аварийные ситуации из-за недостатка памяти. Признаком такой ситуации является возврат нулевого указателя. Так как нормальное продолжение программы при нехватке памяти чаще всего невозможно, то вместо тестирования результата каждой операции new можно воспользоваться централизованной обработкой ситуаций недостатка памяти. В современных системах программирования на языке C++ определенная в файле new.h библиотечная функция

```
void *set_new_handler(void (* my_handler)())();
```

позволяет указать функцию my_handler, которая будет вызвана, если любой последующий оператор new не сможет получить требуемую память.

Функция my_handler может:

• завершить программу вызовом функций, подобных exit() или abort() с выдачей диагностического сообщения;

• выполнить перекомпоновку или освобождение памяти и, завершившись явным либо неявным оператором return, вынудить повторное исполнение оператора new (очевидно, что если памяти снова окажется недостаточно, то налицо условие для бесконечного цикла).

Стандартная реакция на ошибки в операторах new восстанавливается оператором

```
set_new_handler(0);
```

(нулевой указатель устанавливается по умолчанию). Функция set_new_handler возвращает в качестве результата указатель старого обработчика ошибок, что может быть использовано для восстановления реакции:

```
#include <iostream.h>
#include <new.h>
#include <stdlib.h>
void new_error_handler() {
   cerr<<endl<<"Дефицит памяти !"<<endl;
   exit(1);
}

void main() {
   //...
   void (* old_handler)(void) =
    set_new_handler(new_error_handler);

   char *ptr_1=new char[1024];
   cout<<"\nПервое размещение: "<&lthex<&ltlong(ptr_1);

   char *ptr_2=new char[64000U];
   cout<<"\nПоследнее размещение: "<&lthex<&ltlong(ptr_2);

   set_new_handler(old_handler); // Восстановление реакции
   //...
   set_new_handler(0); // Стандартная реакция
   //...
   delete[] ptr_1;
   delete[] ptr_2;
   //...
}
```

Результаты работы программы (потоки cout и cerr):

```
Первое размещение: 28970f28
Дефицит памяти !
```

В новых версиях реализации оператора new при нехватке памяти вместо возврата нулевого указателя может порождаться исключение xalloc [2, 7–9].

6.3. Использование статических элементов класса

Статические элементы класса, по существу, можно считать глобальными объектами программы, доступ к которым разрешен в пределах видимости класса. Независимо от количества фактически существующих объектов класса в программе такие элементы всегда присутствуют в единственном экземпляре. Статические элементы данных можно применять для нумерации и построения механизмов синхронизации процессов использования объектов.

Пример организации управления устройством коллективного использования:

```
class some_process {
   //...
   static int number_process; // Длина очереди процессов
   //...
public:
   some_process();
   virtual ~some_process();
   //...
};

// Статические элементы данных класса должны быть определены
// как глобальные и могут инициализироваться любым выражением,
// включая вызов функций...

int some_process::number_process=0;

some_process::some_process() {
   if (++number_process==1)   // Заявка при пустой очереди
      set_device_online_mode(); // требует включения устройства...
   //...
}
some_process::~some_process() {
   if (--number_process==0)    // При исчерпании очереди заявок
      set_device_offline_mode(); // устройство выключается...
   //...
}
```

Одно из типичных применений статических функций-элементов – обработка прерываний. Обычные функции-элементы не могут быть обработчиками прерываний, т. к. для их вызова требуется формирование в стеке параметров первого неявного параметра – указателя объекта this.

Пример обработки прерываний в классе:

```
#include <iostream.h>
#include <dos.h>
#include <stdlib.h>
#include <time.h>
class wakeup {
   enum { timer_int_no = 0x1C };
   static long count;
   static void interrupt (*old_handler)(...);
   static void interrupt new_handler(...) {
      if (count) count--;
      old_handler();
   }
public:
   wakeup(int delay=10) {
      count=long((delay<0)? 0:delay)*CLK_TCK;
      old_handler=getvect(timer_int_no);
      setvect(timer_int_no, new_handler);
   }
```

```
   operator int() { return count/CLK_TCK; }
   ~wakeup() { setvect(timer_int_no, old_handler); }
};

void interrupt (*wakeup::old_handler)(...)=0;
long wakeup::count=0;

void main() {
   cout<<endl<<"Отсчет 5 сек...";
   wakeup timer_set(5);
   while (timer_set) {
      cout<<endl<<"Еще не вечер...";
      sleep(1);
   }
}
```

Статические функции-элементы класса удобно использовать в качестве аргументов библиотечных функций на языке С. Например, в представленной ниже программе библиотечная функция qsort вызывается с адресом статической функции-элемента rndset::compare():

```
#include <iostream.h>
#include <iomanip.h>
#include <stdlib.h>
class rndset {
   int *x;
   int n;
   static int compare(const void *a, const void *b) {
      return(*((int *)a)-*((int *)b));
   }
public:
   rndset(int m);
   ~rndset() { delete[] x; }
   void sort() { qsort(x,n,sizeof(*x),compare); }
   friend ostream &operator<<(ostream &out, rndset &obj);
};

rndset::rndset(int m=10):n(m) {
   x=new int[n];
   if (x) for (int i=0; i<n; x[i++]=random(n));
}
ostream &operator<<(ostream &out, rndset &obj) {
   for (int i=0; i<obj.n; i++) {
      if (!(i%10)) out<<endl;
      out<&ltsetw(3)<&ltobj.x[i]<<' ';
   }
   return out<<endl;
}

void main() {
   rndset a(30);
   cout<<endl<<"   Исходный массив:"<<endl<<a;
   a.sort();
   cout<<endl<<"   Упорядоченный массив:"<<endl<<a;
}
```

Результаты работы программы:

```
Исходный массив:
 0   0  10   0  10   6  16   5  21  28
 8  13   3  20  16   1   4  24  20  22
24  28   6  12  28  25  27  24  13  18

Упорядоченный массив:
 0   0   0   1   3   4   5   6   6   8
10  10  12  13  13  16  16  18  20  20
21  22  24  24  24  25  27  28  28  28
```

6.4. Статическое и динамическое связывание

Термин «связывание» в языке C++ используют в контексте момента времени привязки функций к обрабатываемым данным. Обычные функции (без атрибута virtual) оказываются связанными **статически**, т. к. эта связь устанавливается на этапе компиляции. Виртуальные функции, вызываемые через указатель или ссылку на базовый класс, могут относиться к разным производным классам. Фактическая принадлежность к конкретному производному классу устанавливается во время исполнения программы, поэтому в таких случаях говорят о **динамическом**, или позднем, связывании. Можно выделить 2 назначения виртуальных функций:

– определение общих для всей иерархии производных классов возможностей с целью единообразия использования;

– определение специфичных возможностей для отдельного уровня наследования, которые не могут быть представлены элементами базового класса.

Рассмотрим действия компилятора при трансляции производных классов. Механизм виртуальных функций реализуется посредством таблиц указателей

```
typedef void (*_vtab[])(...);
```

по следующей схеме:

```
// Пример класса с типичным набором функций-элементов
class any_class {
   int item;
public:
   any_class();
   virtual void print();
   virtual void action();
   virtual int compar(const any_class &)=0;
   int nonvirtual();
};

any_class::any_class() {
// Конструктор по умолчанию в лучшем случае
// очищает элементы данных
```

```
   item=0;
};

void any_class::print() {
// Печать в какой-то форме
};

void any_class::action() {
// Какие-то неспецифические действия с объектом класса
};

int any_class::nonvirtual() {
// Какие-то специфические действия с объектом класса
};
```

Определение класса с учетом некоторого правила декорирования имен его элементов можно представить так:

```
// 1. Имена элементов класса расширяются (декорируются)
//    информацией о принадлежности к классу...
// 2. Каждая нестатическая функция-элемент имеет неявный
//    параметр - указатель this...

void _any_class__print(any_class *this) { /*... */ };
void _any_class__action(any_class *this) { /*... */ };
int _any_class__nonvirtual(any_class *this) { /*... */ };

_vtab _any_class__vtab={ // Таблица виртуальных функций
   _any_class__print,
   _any_class__action,
   NULL // Место для указателя чисто виртуальной функции
};

typedef struct {
   _vtab _vtable; // Добавляемый указатель
   int _any_class_item;
} any_class;

_any_class__any_class(any_class *this) { // Конструктор
   this->_vtable=_any_class__vtab;
   this->_any_class_item=0;
}
```

(здесь предполагается, что имя функции-элемента class::function представлено после расширения как _class__function). Обращения к невиртуальным функциям, например, в виде

```
any_class x, *p;
x.nonvirtual();
p->nonvirtual();
```

компилятор представляет в виде

```
_any_class__nonvirtual(&x);
```

```
_any_class__nonvirtual(p);
```

Для виртуальных функций операция вызова организуется через косвенное обращение к элементам таблицы виртуальных функций класса. Выражения вида

```
x.print();
p->print();
```

на самом деле представляются операциями

```
(x.vtable[_print_index])(&x);
(p->vtable[_print_index])(p);
```

(здесь _print_index – индекс функции print в таблице виртуальных функций, который во всех таблицах иерархии имеет одинаковое значение). Вид последних операций объясняет причины потерь времени на косвенный вызов виртуальных функций. Пусть определен производный класс

```
class derived: public any_class {
    int derived_item;
 public:
    virtual void print();
    virtual int compar(const any_class &);
 };
 void derived::print() {
 // Печать в какой-то форме
 }
 int derived::compar(const any_class &r) {
    return this->item<r.item;
  };
```

Определение такого класса с учетом тех же правил декорирования имен его элементов можно представить так:

```
 void _derived__print(derived *this) { /*... */ };
 void _derived__action(derived *this) { /*... */ };
 int _derived__nonvirtual(derived *this) { /*... */ };

 _vtab _derived__vtab={ // Таблица виртуальных функций
    _derived__print,
    _any_class__action,
    _derived_corpar
};

 typedef struct {
    _vtab _vtable; // Добавляемый указатель
 // Элементы данных базового класса
    int _any_class_item;
 // Элементы данных производного класса
    int derived_item;
} derived;
```

```
_derived__derived(derived *this) {
   _any_class__any_class(this); // Конструктор базового
                                // класса вызван вначале...
   this->_vtable=_derived__vtab;
   this->derived_item=0;
}
```

Интерпретация операций вызова виртуальных функций остается прежней. Для объекта

```
derived d;
```

операция

```
d->print();
```

компилятором будет представлена в виде

```
(d.vtable[_print_index])(&d);
```

После конструирования объекта d в таблице vtable элемент с нулевым индексом указывает на версию функции print производного класса. Говорят, что таблицы виртуальных функций представляют так называемое **динамическое связывание** функций-элементов и данных классов. Такая схема характеризуется тем, что конструктор базового класса не может вызвать виртуальную функцию, переопределенную в производном классе. Этим, в частности, объясняется запрет объявления конструктора виртуальным.

Деструктор класса с виртуальными функциями строится так:

```
_derived__destructor(derived *this) { // Деструктор
// 1. Фиксация указателя таблицы виртуальных функций
   this->_vtable=_derived__vtab;
// 2. Операторы тела деструктора
   // ...
// 3. Вызов деструкторов базовых классов
   // ...
}
```

Отсюда следует, что деструктор производного класса переопределенные виртуальные функции использует правильно.

6.5. Виртуальные деструкторы

По определению деструктор класса может быть виртуальным. Так как любой класс может стать базовым, то деструктор для надежности программирования рекомендуется всегда объявлять виртуальным [4].

Рассмотрим простейшую иерархию классов:

```
class base {
   char *bp;
```

```
public:
   base() { bp=new char[100]; }
   ~base() { delete bp; }
};

class derived: public base {
   char *dp;
public:
   derived() { dp=new char[100]; }
   ~derived() { delete dp; }
};
```

Неприятности использования таких классов проявляются в случае наличия в каких-либо местах программы пары операторов:

```
base *bx=new derived;
//...
delete bx;
//Оператор delete bx компилятор интерпретирует так:
_base__destructor(bx);
free(bx);
```

В результате деструктор фактически существующего объекта производного класса derived не будет вызван, хотя потребность в этом для рассматриваемой версии класса derived несомненна. Если же объявить деструктор базового класса виртуальным, то при интерпретации оператора delete bx компилятор использует косвенный вызов деструктора из таблицы виртуальных функций:

```
(bx->_vtable[destructor_index])(bx);
```

(здесь destructor_index – индекс деструктора в таблице виртуальных функций). Так как bx указывает на объект производного класса, то будет вызван именно деструктор производного класса, который при удалении объекта выполнит вызов и деструктора базового класса.

```
// Использование невиртуального деструктора
class base_1 { // Базовый класс без особенностей
public:
   base_1() { printf("\nbase_1"); }
   ~base_1() { printf("\n~base_1"); }
};
class derived_1: public base_1 {
   char *dp;
public:
   derived_1() {
      dp=new char[100];
      printf("\nderived_1");
   }
   ~derived_1() {
      delete dp;
      printf("\n~derived_1");
   }
};
```

```
// Использование виртуального деструктора
class base_2 { // Базовый класс без особенностей
public:
   base_2() { printf("\nbase_2"); }
   virtual ~base_2() { printf("\n~base_2"); }
};
class derived_2: public base_2 {
   char *dp;
public:
   derived_2() {
      dp=new char[100];
      printf("\nderived_2");
   }
   ~derived_2() {
      delete dp;
      printf("\n~derived_2");
   }
};

void main() {
   printf("\n\n* Использование невиртуального деструктора\n");
   base_1 *pb1=new derived_1;
   delete pb1;

   printf("\n\n* Использование виртуального деструктора\n");
   base_2 *pb2=new derived_2;
   delete pb2;
}
```

Результаты работы программы:

```
* Использование невиртуального деструктора
base_1
derived_1
~base_1

* Использование виртуального деструктора
base_2
derived_2
~derived_2
~base_2
```

Действительная потребность вызова деструктора производного класса в общем случае определяется только особенностями такого класса, но предвосхитить игнорирование вызова деструктора можно только объявлением деструктора базового класса виртуальным.

```
s.String::~String();     // Nonvirtual call
ps->String::~String();   // Nonvirtual call
s.~String();       //
ps->~String();     // Virtual call
```

6.6. Динамические особенности операторов присваивания

Рассмотрим функционирование механизма виртуальных функций при переопределении операторов присваивания функциями operator=(). В следующей программе на первый взгляд учтены все рекомендации по надежному программированию:

```
inline void * operator new(size_t, void *pointer) {
 return pointer;
}
class base { // Класс с функцией operator=()
   int item;
public:
   virtual ~base() {}
   virtual print() { printf("\nКласс base ?"); }
   base():item(0) {}
   base(const base &r):item(r.item) {
      printf("\nКопирование base ");
   };
   const base &operator=(const base &r);
};
const base &base::operator=(const base &r) {
   if (this!=&r) {
      this->base::~base(); // Разрушение текущего объекта
      new(this) base(r);   // и построение на его месте нового
   }
   printf("\nПрисваивание base");
   return *this;
}
class derived:public base { // Класс без функции operator=()
   int d_item;
public:
   virtual print() { printf("\nКласс derived ?"); }
   derived():d_item(0) {}
   ~derived() {}
};
derived x, y; // Глобальные объекты производного класса
void f() { // Тест указателя таблицы виртуальных функций
   base &rx=x;
   rx.print();
   base &ry=y;
   ry.print();
}
void main() {
   printf("\n* Начало...");
   x.print();
   y.print();
   printf("\n* Тест до присваивания");
   f();
   printf("\n* Присваивание");
   y=x;
   printf("\n* Тест после присваивания");
   f();
   printf("\nOK...");
}
```

Результаты работы программы:

```
* Начало...
Класс derived ?
Класс derived ?
* Тест до присваивания
Класс derived ?
Класс derived ?
* Присваивание
Копирование base
Присваивание base
* Тест после присваивания
Класс derived ?
Класс base ?
OK...
```

Однако результаты работы программы не располагают к оптимизму: операция присваивания разрушила указатель виртуальной таблицы объекта.

Причина ошибки здесь в том, что для класса derived компилятор предполагает использование умалчиваемой операции присваивания методом побитового копирования, но при этом дополняет ее явно определенной в базовом классе base функцией const base &operator=(const base &) (см. отображение хода присваивания). Последняя же функция, использующая включающую форму операции new, неявно вызывает конструктор базового класса, который и выполняет замену указателя виртуальной таблицы.

Таким образом, при переопределении операции присваивания для классов с виртуальными функциями вызов конструкторов из функции operator=() должен быть исключен. Если разработчик производного класса не имеет представления о базовом классе, то можно рекомендовать другой радикальный прием защиты от ошибок рассмотренного типа – переопределение операции присваивания в каждом производном классе (легко убедиться, что дополнение класса derived функцией derived &derived::operator=(derived &) приводит к правильной работе программы).

6.7. Точки следования и неопределенное поведение

Точка следования – точка среди выражений программы, в которой завершены вычисления всех предшествующих выражений. Между точками следования порядок вычисления выражений и подвыражений не фиксирован. Причина такой неопределенности – эффективность кода. Стандартами ряда языков, таких как и C/C++, предоставлена свобода планирования вычисления выражений в порядке, наиболее эффективном для целевой аппаратно-программной платформы. Это означает, что проблемы последовательности являются, как правило, проблемами переносимости. Проблема неопределенности порядка вычислений не возникает до тех пор, пока используется один и тот же компилятор для одной и той же платформы.

Список основных видов точек следования в C++:

- в конце оператора-выражения (выражения, заканчивающегося символом «;»);
- в конце выражений условных операторов (if), операторов циклов (while, for, do – while), переключателя (switch);
- после вычисления всех элементов списка параметров функции перед выполнением любых выражений в ее теле;
- после копирования возвращаемого значения и до выполнения любого выражения вне функции;
- после вычисления выражения очередного операнда логических операций (a&&b или a||b), операции условного вычисления (a? b:c) или операции последовательного вычисления (a,b);
- после инициализации каждого базового класса и элемента в списке инициализации конструктора.

Негативные последствия игнорирования точек следования достаточно просто прогнозировать и предвосхищать в выражениях с очевидным множественным доступом к переменной:

```
x[i]=i++ + 1; // x[i]=i + 1; i++;
f(i++); // i++; f(i);
```

Учет точек следования требуется для правильного отображения скрытых зависимостей:

```
x = f() + g() + h();
```

Проблема здесь возникнет при использовании функциями общих модифицируемых переменных. Особый вид скрытой зависимости – исключения, например, из-за нехватки памяти. Возможные причины скрытой зависимости приходится отражать явным назначением точек следования. Например, точки следования при инициализации каждого базового класса и члена в списке инициализации конструктора гарантируют строгий порядок. Инициализации происходят одна за другой в том порядке, в котором базовые классы и члены перечисляются в определении класса:

```
class Array {      // Правильная последовательность строк:
 int* data;        //     unsigned size;
 unsigned size;    //    int* data;
public:
 Array(int Size): size(Size>0? Size: 1), data(new int[size]) {}
};
```

Проблемы точек следования трудно отследить на практике. Отсюда очевидна рекомендация – формируйте исходный код без сложных выражений и особенно тех, которые содержат множественный доступ к одной и той же переменной с ее модификацией. Разложение сложных выражений означает явное введение дополнительных промежуточных точек следования, определяющих последовательность вычисления выражения.

7. ШАБЛОНЫ В ЯЗЫКЕ С++

7.1. Назначение и виды шаблонов

Естественное желание сокращения объема программы часто противоречит строгости контроля типов компилятором и требованиям надежного программирования. Например, задача определения максимального значения из пары значений разных типов может породить в программе наличие функций

```
int    max(int x, int y)          { return (x>y)? x:y; }
long   max(long x, long y)        { return (x>y)? x:y; }
char   max(char x, char y)        { return (x>y)? x:y; }
double max(double x, double y)    { return (x>y)? x:y; }
```

Тело функций во всех вариантах программы одинаково, но из-за различия типов параметров и возвращаемых значений результат компиляции будет существенно различным. Идея использования препроцессора здесь может не отвергаться лишь по причине малости текста функций, в других же ситуациях она несостоятельна по надежности. Приведенный пример иллюстрирует целесообразность параметризации определений функций в отношении **типов параметров** функций. Это оформляют в интуитивно понятной форме

```
// Параметризация функций в C++
template <class Type> // Шаблон функции максимума
Type max(Type x, Type y) { return (x>y)? x:y; }
```

(здесь Type – параметр, представляющий тип).

В языке С++ в современных системах программирования параметризация типов поддерживается понятиями **шаблонов** (template) **функций** и **классов**. Шаблоны класса, в отличие от шаблонов функций, позволяют параметризовать не только признак типа элементов класса, но и **константы** разных типов. Например, определения классов

```
class char_array { // Массив символов фиксированного размера
protected:
   char *area;
public:
   char_array() {
      area=new char[81]; // Константа характеризует тип!
   }
   ~char_array() {
      delete[] area;
   }
};

class int_array { // Массив слов фиксированного размера
protected:
   char *area;
public:
```

```
   int_array() {
      area=new int[150]; // Константа характеризует тип!
   }
   ~int_array() {
      delete[] area;
   }
};
```

из практических соображений можно было бы параметризовать относительно типа элементов и размера массива. Предварительно отметим, что это можно было бы выразить в форме

```
// Параметризация класса в C++
template <class Type, int Size> // Шаблон класса
class any_type_array { // Массив фиксированного размера
protected:
   Type *area;
public:
   any_type_array() {
      area=new Type[Size]; // Константа характеризует тип!
   }

   ~any_type_array() {
      delete[] area;
   }
};
```

(здесь Type – параметр, представляющий тип, а Size – константу типа int).

```
template-declaration :
 template < template-argument-list > declaration

template-argument-list :
 template-argument
 template-argument-list , template-argument

 template-argument :
  type-argument
  argument-declaration

 type-argument :
  class identifier
typename identifier
```

Механизм шаблонов в языке C++ – средство построения обобщенных определений функций и классов, независимых от используемых типов. Использование шаблонов избавляет от необходимости переписывания исходного текста функций и/или классов для различных типов объектов. Компилятор на основе определения шаблона по заданному в качестве параметра типу автоматически порождает соответствующие экземпляры или так называемые **представители** функции или класса.

7.2. Определение и использование шаблонов функций

Синтаксис определения шаблона функции:

```
template <список_параметров_шаблона> декларация_функции
```

(здесь угловые скобки обязательны).

Элемент «список_параметров_шаблона» определяет набор типов, параметризующих текст декларации функции. Каждый тип представляется ключевым словом class и локальным в рамках элемента «декларация_функции» идентификатором типа. Элемент «список_параметров_шаблона» не может быть пустым, а его элементы разделяются запятыми. Элемент «декларация_функции» подобен обычному определению или описанию функции, но в списке параметров функции должны быть хотя бы раз упомянуты типы, перечисленные в списке параметров шаблона.

Примеры отношений между списками параметров шаблона и функции:

а) шаблон с единственным параметром

```
template <class T>
void f1 (T par) {
   //
   // Тело функции f1
   //
}
```

(тип T можно использовать для спецификации возвращаемого значения и параметров функции, а также любых объектов в теле функции; обязательное требование упоминания T в списке параметров функции выполнено);

б) шаблон функции с частично параметризованными параметрами

```
template <class T>
void f2 (T par, int x, float y) {
   //
   // Тело функции f2
   //
}
```

(типы параметров шаблона должны обязательно использоваться в списке параметров функции, но функция может иметь параметры любых типов);

в) шаблон с несколькими параметрами

```
template <class T1, class T2>
void f3 (T1 par_1, T2 par_2) {
   //
   // Тело функции f3
   //
}
```

(здесь типы T1 и T2 необязательно должны быть различными, а порядок их перечисления в списке параметров шаблона значения не имеет, поэтому определение шаблона в виде

```
template <class T2, class T1>
void f3 (T1 par_1, T2 par_2) {
   //
   // Тело функции f3
   //
}
```

эквивалентно исходному варианту).

Использование шаблона функции не требует каких-либо действий от программиста – компилятор автоматически формирует требуемый операцией ее вызова экземпляр кода по набору типов аргументов. Параметризуемая функция может иметь атрибут inline, т. е. быть встраиваемой. Пример определения и использования шаблона функций:

```
#include <iostream.h>
#include <string.h>
template <class T> // Шаблон функции максимума
T max(T x, T y) { return (x>y)? x:y; }
class string { // Класс строк
   char *buffer;
public:
   string(char *x) {
      buffer=new char[strlen(x)+1];
      if (buffer) strcpy(buffer,x);
   }
   ~string() {
      delete buffer;
   }
   int operator>(string &item) {
      return strcmp(this->buffer,item.buffer)>0;
   }

   friend ostream &operator<<(ostream& out, string &obj) {
      out<&ltobj.buffer;
      return out;
   }
};

void main(void) {
   int ix=2, iy=3;
   cout<<endl<&ltix<<' '<&ltiy<<' '<&ltmax(ix,iy);

   float fx=1.12, fy=1.13;
   cout<<endl<&ltfx<<' '<&ltfy<<' '<&ltmax(fx,fy);

   long lx=123, ly=77;
   cout<<endl<&ltlx<<' '<&ltly<<' '<&ltmax(lx,ly);
}
```

Результаты работы программы:

```
 2 3 3
1.12 1.13 1.13
123 77 123
```

Шаблон функции определения максимума здесь избавляет от необходимости копирования ее определения для **любого типа параметров**. Нетрудно заметить, что тело параметризуемой функции не изменится при наличии переопределения операций для классов пользователя. Отсюда следует, что определения шаблонов целесообразно помещать в заголовочные файлы.

Технология подготовки шаблонов функций на первый взгляд весьма проста. Предположим, имеется текст программы сортировки массива целых чисел:

```
// Пример процедуры сортировки методом Шелла

void shells(int x[], int n) {
   if (n<=1) return;
   for (int k=1; k<=n; k<<=1);
      for (int m=(k>&gt1)-1; m; m>>=1)
         for (int k=n-m, j=0; j<k; j++)
            for (int i=j; (i>=0)&&(x[i]>x[i+m]); i-=m) {
               int tmp=x[i+m];
               x[i+m]=x[i], x[i]=tmp;
            }
}
```

Параметризованный относительно типа элементов массива шаблон такой функции, помещаемый в файл shells.h, может иметь вид

```
//  Шаблон процедуры сортировки методом Шелла

#ifndef SHELLS_H
#define SHELLS_H

template <class T>
void shells(T x[], int n) {
   if (n<=1) return;
   for (int k=1; k<n; k<<=1);
      for (int m=k-1; m; m>>=1)
         for (int k=n-m, j=0; j<k; j++)
            for (int i=j; (i>=0)&&(x[i]>x[i+m]); i-=m) {
               T tmp=x[i+m]; // Вызов конструктора копирования!
               x[i+m]=x[i], x[i]=tmp;
            }
}

#endif // SHELLS_H
```

Здесь тип элементов массива обозначен идентификатором T. Такой идентификатор использован в списках параметров шаблона и функции, а также в операторе декларации буферной переменной tmp. Так как текст функции других изменений не имеет, то в классе T должны быть определены все виды конструкторов (конструктор по умолчанию, конструктор копирования). Для элементов класса T, кроме этого, требуется определение функций operator>() и operator=().

Программа тестирования шаблона функции shells:

```
 #include <stdio.h>
#include <string.h>
#include "shells.h"
class string {
   char *buffer;
public:
   string():buffer(0) {} // Конструктор по умолчанию
   string(char *x) {
      buffer=new char[strlen(x)+1];
      if (buffer) strcpy(buffer,x);
   }
   string(string &x) { // Конструктор копирования
      buffer=new char[strlen(x.buffer)+1];
      if (buffer) strcpy(buffer,x.buffer);
   }

   int operator>(string &item) {
      return strcmp(this->buffer,item.buffer)>0;
   }

   string &operator=(string &item) { // Операция присваивания
      if (buffer) delete buffer;
      buffer=new char[strlen(item.buffer)+1];
      if (buffer) strcpy(buffer,item.buffer);
      return *this;
   }

   void operator=(char *item) { // Операция присваивания
      if (buffer) delete buffer;
      buffer=new char[strlen(item)+1];
      if (buffer) strcpy(buffer,item);
   }

   void print() { printf(" %s",buffer); }

   ~string() {
      delete buffer;
   }
};

void main(void) {
   int y[]={2,7,1,3,8};
   int n=sizeof(y)/sizeof(*y);
   string *x=new string[n];
   for (int i=0; i<n; i++) {
      char temp[10];
      sprintf(temp,"\"%03d\"\0",y[i]);
      x[i]=temp;
   }

   printf("\n\n Вход: ");
   for (i=0; i<n; i++) printf(" %5d",y[i]);
   shells(y,n);
   printf("\nВыход: ");
   for (i=0; i<n; i++) printf(" %5d",y[i]);
```

```
   printf("\n\n Вход: ");
   for (i=0; i<n; i++) x[i].print();
   shells(x,n);
   printf("\nВыход: ");
   for (i=0; i<n; i++) x[i].print();
}
```

Результаты работы программы:

```
 Вход:      2     7     1     3     8
Выход:      1     2     3     7     8

 Вход:  "002" "007" "001" "003" "008"
Выход:  "001" "002" "003" "007" "008"
```

Нетрудно заметить, что шаблон функции shells разрешает переопределение и операции индексации. Это позволяет выполнять сортировку на структурах хранения данных, отличных от линейного массива. Таким образом, механизм шаблонов существенно расширяет возможности формализации прикладных задач, предоставляемые языком C++, ранее рассмотренными понятиями классов и полиморфизма. Шаблоны могут применяться для взаимосвязанных функций:

```
// Решение системы линейных уравнений ax=b методом исключения

template <class Type> // Модуль числа произвольного типа
inline Type abs(const Type &value) {
   return (value<0)? -value : value;
}

template <class Type> // Реализация метода исключения
int simq(Type **a, Type *b, int n) {
   for (int j=0; j<n; j++) { // Цикл по всем переменным
// Поиск максимального элемента столбца
      Type amax=0, temp;
      for (int i=j, m; i<n; i++)
         if ((temp=abs(a[i][j]))>amax) {
            amax=temp, m=i;
         }
      if (!amax) return 0; // Случай вырожденной матрицы
// Деление текущего уравнения с заменой строк
      Type *ai=a[m], *aj=a[j];
      amax=ai[j];
      for (int k=j; k<n; k++) {
         temp=aj[k], aj[k]=ai[k], ai[k]=temp;
         aj[k]=amax;
      }
     Type &bj=b[j];
     temp=b[m], b[m]=bj, bj=temp/amax;
// Исключение текущей переменной
     for (i=m=j+1; i<n; i++) {
        ai=a[i];
        Type &aij=ai[j];
```

```
            for (k=m; k<n; k++)
               ai[k]-=aij*aj[k];
            b[i]-=bj*aij;
         }
      }

   // Формирование результата

      for (j=--n-1; j>=0; j--) {
         Type *aj=a[j], &bj=b[j];
         for (int k=n; k>j; k--)
            bj-=aj[k]*b[k];
      }
      return 1; // Случай невырожденной матрицы
   }
```

Здесь основной модуль – функция simq(), вызывающая функцию abs() для получения значения модуля элемента матрицы. Обе функции параметризованы раздельно, но компилятор предусмотрит их правильное совместное использование.

7.3. Переопределение шаблонов и специализация функций

Шаблоны функций могут быть переопределены подобно переопределению функций, т. к. их использование базируется на распознавании компилятором различий в списках параметров функций. Пример переопределения шаблонов функций:

```
// ШАБЛОНЫ ФУНКЦИЙ ПОИСКА МИНИМАЛЬНОГО ЗНАЧЕНИЯ
template <class T> // Выбор минимума из пары объектов
T min(T x, T y) { return (x<y)? x:y; }
template <class T> // Выбор минимума из массива объектов
T min(T *x, int n) {
   T temp=x[0];
   for (int i=1; i<n; i++)
      if (x[i]<temp) temp=x[i];
      return temp;
   }
```

(для ссылок далее текст шаблонов поместим в файл min_tmpl.h).

Более того, в программе могут присутствовать и обычные функции, переопределяющие специфическим составом списка параметров функции шаблона (такие функции иногда называют **специализированными**). Например, при намерении выбора минимума из пары строк в лексикографическом порядке текст программы можно просто дополнить функцией

```
// СПЕЦИАЛИЗИРОВАННАЯ ФУНКЦИЯ ПОИСКА МИНИМАЛЬНОГО ЗНАЧЕНИЯ
char *min(char *x, char *y) { return strcmp(x,y)<0 ? x:y; }
```

(для ссылок далее текст этой функции поместим в файл min_char.h).

Возможные конфликты разрешаются следующей жесткой последовательностью этапов выбора экземпляра переопределенных функций:

1) поиск специализированной (не представленной шаблонами) функции с совпадающим списком параметров;

2) поиск шаблона функции с точным соответствием списка параметров;

3) поиск специализированной функции по условию совпадения списка параметров после возможных преобразований типов.

Успех поиска на любом этапе завершает выбор функции. Отметим, что анализ компилятором шаблонов функций не предполагает преобразования типов.

Пример использований переопределенных шаблонов и специализированных функций:

```
#include <stdio.h>
#include <string.h>
#include "min_tmpl.h" // ШАБЛОНЫ ФУНКЦИЙ ПОИСКА
#include "min_char.h" // СПЕЦИАЛИЗИРОВАННАЯ ФУНКЦИЯ ПОИСКА
void main(void) {
   int y[]={2,7,1,3,8};
   int n=sizeof(y)/sizeof(*y);
   printf("\n\n Минимум из {%d",y[0]);
   for (int i=1; i<n; i++) printf(", %d",y[i]);
   printf("} есть %d",min(y,n));
   printf("\n Минимум из {%d, %d} есть %d",
   y[0],y[1],min(y[0],y[1]));
   char s1[]="one", s2[]="two";
   printf("\n Минимум из {\"%s\", \"%s\"} есть \"%s\"",
         s1,s2,min(s1,s2));
}
```

Результаты работы программы:

```
Минимум из {2, 7, 1, 3, 8} есть 1
Минимум из {2, 7} есть 2
Минимум из {"one", "two"} есть "one"
```

7.4. Определение шаблонов классов

Синтаксис определения шаблона класса:

```
template <список_параметров_шаблона> определение_класса
```

(здесь угловые скобки обязательны, как и для шаблонов функций).

Элемент «список_параметров_шаблона» не может быть пустым, а его элементы разделяются запятыми.

Виды параметров шаблона:

- параметр типа – ключевое слово class и идентификатор;
- нетиповой параметр – имя типа и идентификатор, определяющие константу.

Элемент «список_параметров_шаблона» определяет набор типов и констант, параметризующих текст определения класса. Каждый параметр является локальным в рамках элемента «определение_класса».

Элемент «определение_класса» – обычное определение класса, где хотя бы раз упомянуты идентификаторы типовых и нетиповых параметров из списка параметров шаблона.

Если функции-элементы определяются вне определения класса, то синтаксис их определения должен иметь вид

```
template <список_параметров_шаблона>
тип_результата
имя_класса<параметры_шаблона>::
имя_функции(параметры_функции) {
   // Тело функции
}
```

Пример параметризации класса с внешним определением функций-элементов:

```
#ifndef ANY_TYPE_ARRAY_H
#define ANY_TYPE_ARRAY_H
// Шаблон класса с описанием функций-элементов
template <class Type, int Size>
class any_type_array {
protected:
   Type *area;
public:
   any_type_array();
   ~any_type_array();
};

// Внешние определения функций-элементов шаблона класса

template <class Type, int Size> // Шаблон конструктора
any_type_array<Type,Size>::any_type_array() {
   area=new Type[Size];
   printf("\n+ Size=%d %d",Size,sizeof(Type));
}
template <class Type, int Size> // Шаблон деструктора
any_type_array<Type,Size>::~any_type_array() {
   printf("\n- Size=%d %d",Size,sizeof(Type));
   delete[] area;
}
#endif // ANY_TYPE_ARRAY_H
```

(для дальнейших ссылок предполагается наличие этого текста в файле tarray.h).

Вложенность определений шаблонов классов не допускается, однако внутри элемента «определение_класса» можно объявить вложенный класс с параметрами определяемого шаблона класса. Ограничений на содержание определения параметризуемого класса нет. Допускается использование статических элементов данных и функций, дружественных функций и вложенных классов.

7.5. Использование шаблонов классов

Шаблон класса используется для создания его представителей в местах употребления имени класса как имени типа (в языке C++ понятие класса – синоним типа, определяемого пользователем). Например, имя типа необходимо в операторах декларации объектов, приведения типа, переопределения типа (typedef) и т. п. Имя представителя образуется из имени шаблона класса и заключенного в угловые скобки списка аргументов. В списке аргументов каждому типовому параметру шаблона должно соответствовать известное в этой точке имя типа, а нетиповому – константное выражение ожидаемого типа.

Пример использования шаблона класса any_type_array:

```
#include <stdio.h>
#include "tarray.h"
void main() {
   any_type_array<int, 50>  x;
   any_type_array<long,25>  y;
}
```

Результаты работы программы:

```
+ Size=50 2
+ Size=25 4
-  Size=25 4
-  Size=50 2
```

Программирование шаблонов может проводиться с учетом возможностей автоматического образования представителей производных классов:

```
#include <iostream.h>
template <class Type>
class array {
protected:
   Type *area;
   int number;
public:
   class invalid_index {}; // Исключение при нарушении границ
   virtual ~array();
   array() {};
   array(int size);
   Type &operator[](int index) {
      if ((index>=0)&&(index<number)) return array::area[index];
      throw invalid_index; // Порождение исключения (см. 3.1)
   }
   operator int() { return area && number; }
};

template <class Type>
array<Type>::array(int size):
  number(size), area(new Type[size]) {}

template <class Type>
array<Type>::~array() {
```

```
   delete[] area;
}

template <class Type, int size>
class any_array: public array<Type> {
public:
   any_array():array<Type>(size) {}
};

void main(void) {
   array<int> X(10);
   if (X) cout<<endl<<"X: OK...";

   any_array<any_array<int,10>,15> Y;
   if (Y) cout<<endl<<"Y: OK...";
}
```

Результаты работы программы:

```
X: OK...
Y: OK...
```

Обратите внимание на оператор декларации объекта Y: в качестве параметра представителя шаблона использован также представитель шаблона.

7.6. Специализация параметризованных классов

Подобно специализации шаблонов функций, возможна специализация и шаблонов классов. Специализация класса может быть частичной либо полной. Частичная специализация предполагает явное определение представителей некоторых функций-элементов. Полная специализация соответствует переопределению самого класса с изменением состава его элементов.

Особенности синтаксиса специализации шаблонов отражены в следующем примере программы:

```
// Параметризация класса
template <class Type, int Size> // Шаблон класса
class any_type_array {
protected:
   Type *area;
public:
   any_type_array();
   ~any_type_array();
};

template <class Type, int Size> // Шаблон класса
any_type_array<Type,Size>::any_type_array() {
   area=new Type[Size];
   printf("\n+ Size=%d * %d",Size,sizeof(Type));
};
template <class Type, int Size> // Шаблон класса
any_type_array<Type,Size>:: ~any_type_array() {
```

```
   printf("\n- Size=%d * %d",Size,sizeof(Type));
   delete[] area; }

// Частичная специализация класса

any_type_array<int,11>::any_type_array() {
   area=new Type[Size];
   printf("\n+ Size=%d * %d ???",Size,sizeof(Type)); }
any_type_array<long,25>:: ~any_type_array() {
   printf("\n- Long Size=%d * %d",Size,sizeof(Type));
   delete[] area;  }

// Полная специализация класса

class any_type_array<char,100> {
   char buffer[100];  // Новый элемент данных
protected:
   char *area;
public:
   any_type_array<char,100>() {
      area=new char[100];
      area[0]=0;
      printf("\n+ Char Size=100 * %d",sizeof(char)); }
   any_type_array<char,100>(char *s) {
      area=new char[100];
      strcpy(area,s);
      printf("\n+ Char Size=100 * %d",sizeof(char)); }
   void print() { // Новая функция
      printf("\n Area=\"%s\"",area);
   }

   ~any_type_array() {
      printf("\n- Char Size=%d %d ???",100,sizeof(char));
      delete[] area;
   }
};

void main() {
   any_type_array<int,50>  x1;
   any_type_array<int,11>  x2; // Специализация
   any_type_array<long,25> y0;
   any_type_array<char,100> z1("z1"), z2;  // Специализация
   z1.print();
   z2.print();
   any_type_array<char,30> z3;
}
```

Результаты работы программы:

```
+ Size=50 * 2
+ Size=11 * 2 ???
+ Size=25 * 4
+ Char Size=100 * 1
+ Char Size=100 * 1
Area="z1"
Area=""
```

```
+ Size=30 * 1
-  Size=30 * 1
-  Char Size=100 1 ???
-  Char Size=100 1 ???
-  Long Size=25 * 4
-  Size=11 * 2
-  Size=50 * 2
```

Отметим, что специализированные функции шаблона класса могут быть переопределены многократно, но шаблон класса переопределять нельзя. Например, одновременное наличие в тексте программы определений

```
template <class Type, int Size>
class any_type_array { /* ... */ };
template <int Size, class Type>
class any_type_array { /* ... */ };
template <class Type>
class any_type_array { /* ... */ };
template <int Size>
class any_type_array { /* ... */ };
```

считается ошибочным.

Последнее отмеченное ограничение не является обременительным, т. к. можно просто использовать новое имя класса. Более того, даже непараметризованный класс – достаточно сложный элемент любой программы. Многократное переопределение класса чаще всего свидетельствует о недостаточно продуманном проектировании иерархии классов.

Ключевое слово typename используется только в определении шаблона

```
template <class T> class X  {
     typename T::Y m_y;   // treat Y as a type
};
template <class T1, class T2> ...
template <typename T1, typename T2> ...
```

7.7. Библиотека стандартных шаблонов

Библиотека стандартных шаблонов (Standard Template Library, STL) – набор согласованных обобщенных алгоритмов, контейнеров, средств доступа к их содержимому и различных вспомогательных функций в C++ [15–16].

STL содержит пять основных видов компонентов:

- алгоритм (algorithm) – определяет вычислительную процедуру;
- контейнер (container) – управляет набором объектов в памяти;
- итератор (iterator) – средство доступа алгоритма к содержимому контейнера;
- функциональный объект (function object) – инкапсулирует функцию в объекте для использования другими компонентами;

• адаптер (adaptor) – адаптирует компонент для обеспечения различного интерфейса.

Компоненты библиотеки объявлены в отдельных заголовочных файлах, интуитивно понятные имена которых часто совпадают с именами шаблонных классов:

```
<algorithm> <limits>     <cstdarg>       <sstream>   <fstream>
<cctype>    <map>        <cstdlib>       <streambuf> <hash_set>
<ciso646>   <numeric>    <cwchar>        <utility>   <iosfwd>
<cmath>     <set>        <exception>     <cassert>   <iterator>
<csignal>   <stdexcept>  <hash_map>      <cfloat>    <locale>
<cstdio>    <strstream>  <ios>           <clocale>   <new>
<ctime>     <vector>     <istream>       <csetjmp>   <queue>
<deque>     <bitset>     <list>          <cstddef>   <stack>
<functional><cerrno>     <memory>        <cstring>   <string>
<iomanip>   <climits>    <ostream>       <cwctype>   <valarray>
<iostream>  <complex>
```

Пример использования шаблона vector:

```
#include <iostream>
#include <vector>
using namespace std; // Добавляем нужное пространство имен
void main() {
    vector <int> k; // Объявляем вектор из целых.
    // Добавляем элементы в конец вектора.
    k.push_back(22);
    k.push_back(11);
    k.push_back(4);
    // Показываем все элементы вектора.
    for (int i = 0; i<k.size(); i++) {
        cout<<k[i]<<"\n";
    }
    cout<<"***\n";

    k.pop_back(); // Удаляем элемент с конца вектора.
    // Показываем все элементы вектора.
    for (i = 0; i<k.size(); i++){
        cout<<k[i]<<"\n";
    }
    cout<<"***\n";
    k.clear();    // Удаляем все элементы вектора
    if(k.empty) {// Проверяем, что вектор пуст.
        cout<<"Vector is empty\n";
    }
}
```

Пример использования шаблона string:

```
#include <iostream>
#include <string>
using namespace std;
void main(){
    string s0 = "abcde", s1 = "123 fg";
```

```
    string s = s0 + s1; // Конкатенация строк.
    cout<<s<<"\n";

    char ch0 = s0.at(1); // Получаем символ на определенном месте.
    cout<<ch0<<"\n";
    char ch1 = s0[3];
    cout<<ch1<<"\n";

    if (s0.empty()){ // Выясняем, не пустая ли строка.
        cout << "String is empty"<<"\n";
    } else {
        cout << "String isn't  empty"<<"\n";
    }
    swap(s0, s1); // Обмен значения двух строк.

    s1 = s0; // Присваиваем и сравниваем 2 строки.
    if(s1 == s0){
        cout << "Strings are equal"<<"\n";
    }
    else{
        cout << "Strings are not equal"<<"\n";
    }
    getline(cin, s1); // Чтение введенной с клавиатуры строки.
    cout<<s1;
    cout<<s1.length();// Получение длины строки.
}
```

Основные методы, которые есть почти во всех коллекциях STL:

- empty – определяет, является ли коллекция пустой;
- size – определяет размер коллекции;
- begin – возвращает итератор, указывающий на начало коллекции;.
- end – возвращает итератор, указывающий на конец коллекции (реально он не указывает на ее последний элемент, а указывает на воображаемый несуществующий элемент, следующий за последним);
- rbegin – возвращает обратный итератор, указывающий на начало коллекции;
- rend – возвращает обратный итератор, указывающий на конец коллекции;
- clear – удаляет все элементы коллекции (если в коллекции сохранены указатели, то необходимо удалить все элементы посредством вызова delete для каждого указателя);
- erase – удаляет элемент или несколько элементов из коллекции;
- capacity – вместимость коллекции определяет реальный размер буфера коллекции, а не то, сколько в нем хранится элементов (в некоторых реализациях STL первое выделение памяти происходит не в конструкторе, а при добавлении первого элемента коллекции.

STL расширяет основные средства C++ так, что программисту легко начать ею пользоваться. Например, STL содержит шаблонную функцию merge (слияние). Массивы a и b объединить в c можно так:

```
int a[1000], b[2000], c[3000];
merge(a, a+1000, b, b+2000, c);
```

Объединить вектор и список (классы vector и list) и поместить результат в заново распределенную неинициализированную память можно так:

```
vector<Employee> a;
list<Employee> b;
Employee* c = allocate(a.size() + b.size(), (Employee*) 0);
merge(a.begin(), a.end(), b.begin(), b.end(),
      raw_storage_iterator <Employee*, Employee> (c));
```

Здесь *begin()* и *end()* – функции-элементы контейнеров, позволяющие функции *merge* выполнить слияние, а *raw_storage_iterator* – адаптер, который помещает результат слияния в неинициализированную память, вызывая соответствующий конструктор копирования.

STL позволяет перемещаться в потоках ввода – вывода таким же образом, как и в обычных структурах данных. Например, если необходимо объединить две структуры данных и затем сохранить их в файле, то будет хорошо поместить результат непосредственно в файл без промежуточных структур. STL включает шаблонные классы *istream_iterator* и *ostream_iterator* для представления потоков ввода – вывода однородными блоками данных.

Далее приведем программу чтения файла целых чисел из стандартного потока ввода, удаляя все числа, делящиеся на параметр команды запуска программы, и записи результата в стандартный поток вывода:

```
void main((int argc, char** argv) {
 if (argc != 2) throw("usage: remove_if_divides integer\n ");
 remove_copy_if(
  istream_iterator<int>(cin),
  istream_iterator<int>(),
  ostream_iterator<int>(cout, "\n"),
  not1(bind2nd(modulus<int>(),atoi(argv[1])))
 );
}
```

Работа выполняется алгоритмом *remove_copy_if*, который читает целые числа одно за другим, пока итератор ввода не становится равным итератору end-of-stream (конец потока), который создается конструктором без параметров. Затем *remove_copy_if* записывает прошедшие проверку целые числа в выходной поток через итератор вывода, который связан с *cout*. В качестве предиката условия проверки *remove_copy_if* использует функциональный объект, созданный из функционального объекта modulus<int> как бинарный предикат, и превращает в унарный предикат, используя bind2nd для связи второго параметра с параметром командной строки atoi(argv[1]). Отрицание этого унарного предиката получается с помощью адаптера not1.

До этого были представлены основные приемы использования STL на примере использования некоторых коллекций. Это основа STL, но для того,

чтобы использовать всю мощь этой библиотеки, придется самостоятельно расширить практические знания.

STL в файле <algorithm> представляют набор готовых шаблонов функций, которые могут быть применены к коллекциям и могут быть подразделены на три основных группы: функции перебора всех членов коллекции, сортировки коллекции и функции выполнения арифметических действий над коллекцией.

Функции перебора всех членов коллекции и выполнения определенных действий над каждым из них:

```
count                   iter_swap               remove_copy
count_if                swap_ranges             remove_copy_if
find                    fill                    unique
find_if                 fill_n                  unique_copy
adjacent_find           generate                reverse
for_each                generate_n              reverse_copy
mismatch                replace                 rotate
count                   iter_swap               remove_copy
count_if                swap_ranges             remove_copy_if
find                    fill                    unique
find_if                 fill_n                  unique_copy
adjacent_find           generate                reverse
for_each                generate_n              reverse_copy
mismatch                replace                 rotate
equal                   replace_if              rotate_copy
search copy             transform               random_shuffle
copy _backward          remove                  partition
swap                    remove_if               stable_partition
```

Функции сортировки членов коллекции:

```
sort                    merge                   pop_heap
stable_sort             inplace_merge           sort_heap
partial_sort            includes                min
partial_sort_copy       set_union               max
nth_element             set_intersection        min_element
binary_search           set_difference          max_element
lower_bound             set_symmetric_difference lexographical_compare
upper_bound             make_heap               next_permutation
equal_range             push_heap               prev_permutation
```

Функции выполнения определенных арифметических действий над членами коллекции:

```
acumulate
partial_sum
inner_product
adjacent_difference
```

С использованием алгоритмов STL возможно создание эффективных программ. По компактности такой код превосходит код, написанный на таких со-

временных языках, как Java и C#, и в значительной степени эффективнее последних.

Для того чтобы использовать все это разнообразие, под рукой должна быть соответствующая документация. Microsoft предлагает достаточно подробную документацию, как часть MSDN для своей реализации STL.

7.8. Характеристика шаблонов

Шаблонное программирование продолжает эволюционирование и позволяет реализовать такие методики, как обобщенное программирование, вычисления во время компиляции, библиотеки шаблонных выражений, шаблонное метапрограммирование, порождающее программирование [11, 15–16]. Например, в шаблонах разрешено использование параметров по умолчанию:

```
template<typename Type>
 class allocator {};

template<typename Type,
 typename Allocator=allocator<Type> > // Пробел обязателен!
 class stack {};

stack<int> MyStack;
```

Подобно использованию идентификаторов в обычных операторах декларации объектов программы языка C++ допускается использование идентификаторов параметра шаблона в продолжении списка параметров:

```
class Y {};

template<class T, T *pT> class X1 {};
template<class T, T **rT = 0> class X2 {};

Y aY;
X1<Y, &aY> x1;
X2<int> x2;
```

В среде Microsoft Visual C++ .NET можно использовать параметризованные параметры шаблонов:

```
template <class T>
 struct S1 { T t; };

template <template<class A> class T>
 struct S2 { T<int> t; };

void main() {
 S2<S1> myS2;
 myS2.t.t = 2005;
 printf("%d\n", myS2.t.t);
}
```

Эффективное применение шаблонов – вычисления во время компиляции. В ряде случаев это позволяет заменить рекурсивный вызов функции рекурсивной конкретизацией шаблона. Идею подобной замены демонстрирует программа вычисления факториала:

```
int factorial (int n)  { return (n==0)? 1: n*factorial(n-1); }

template <int n>
 struct Factorial { enum { ret = Factorial<n-1>::ret * n };  };

template <>
 struct Factorial<0> { enum { ret = 1 }; };

void main() {
 cout << "f=" << factorial(4) << endl;
 cout << "F=" << Factorial<4>::ret << endl;
}
```

Достоинства шаблонов:

– высокая эффективность работы с различными типами объектов класса по сравнению с полиморфизмом;

– безопасное использование типов.

Недостатки шаблонов:

– увеличение размера исполняемого модуля программы из-за наличия представителей шаблона для каждого порожденного типа;

– реализация шаблона может оказаться оптимальной только для некоторого набора типов.

Очевидно, что отмеченные черты не могут быть причиной отказа от использования шаблонов, т. к. возможность их специализации позволяет добиться требуемого качества программы.

Например, шаблоны функций минимума и максимума в форме

```
template <class T>
inline const T& min(const T &x, const T &y) {
 return x>y ? y:x;
}

template <class T>
inline const T& max(const T &x, const T &y) {
 return x>y ? x:y;
}
```

(подобное определение имеется в файле stdlib.h) применимы ко всем типам, но для базовых типов данных недостаточно эффективны: ссылки на параметры и результат порождают дополнительные машинные команды косвенных обращений.

8. ОБРАБОТКА ИСКЛЮЧИТЕЛЬНЫХ СИТУАЦИЙ

8.1. Схема обработки исключений в C++

Термин «исключение» (exception) в C++ принято использовать для обозначения особых ситуаций, когда продолжение выполнения естественной последовательности операторов требуется безвозвратно переключить на блок завершающей части такой последовательности. Выявление особых ситуаций выполняется программным способом путем проверки некоторых условий нормального хода программы (например, наличие выделенной памяти по ненулевому значению указателя). Порождение (выброс) исключения программируется специальным оператором throw, синтаксис которого имеет вид

```
throw выражение;
```

(такой оператор подобен оператору возврата из функции и может быть использован в любом месте программы).

Местоположение оператора throw иногда называют точкой выброса исключения:

```
// Фрагмент программы с порождением исключений

int *int_array=new int[1000];
if (!int_array)
   throw 1000; /////////////// Выброс исключения типа int
ofstream output("e:\temp\data");
if (!output)
   throw "e:\temp\data"; ///// Выброс исключения типа char *
```

Очевидно, что при нормальном ходе программы точки выброса исключений обходятся. Управление исключениями согласно стандарту ANSI C++ строится по схеме завершающего выхода из контролируемого блока программы. Контролируемый блок программы оформляется посредством синтаксической конструкции вида

```
try {
   //
   // Операторы контролируемого блока, включая throw
   //
}
// Определение ловушек исключений – операторы catch
```

Среди операторов контролируемого блока может быть любое количество операторов throw. Контролируемые блоки могут быть вложенными. Тело любой функции, вызываемой прямо либо косвенно из контролируемого блока, считается принадлежащим этому блоку. Непосредственно за контролируемым

блоком размещаются в любом количестве блоки обработки исключительных ситуаций, синтаксис которых может иметь одну из трех форм:

1) параметризованный специализированный блок

```
catch (имя_типа имя_параметра) {
    //
    // Операторы блока обработки исключения
    //
}
```

(специализации блока соответствует «имя_типа», а элемент «имя_параметра» обозначает локальный в теле этого блока объект);

2) непараметризованный специализированный блок

```
catch (имя_типа) {
    //
    // Операторы блока обработки исключения
    //
}
```

(здесь специализации блока соответствует «имя_типа», но параметр этого типа в блок не передается);

3) неспециализированный блок

```
catch (...) {
    //
    // Операторы блока обработки исключения
    //
}
```

(многоточие здесь – синтаксический элемент, обозначающий способность блока обрабатывать не обслуженные предшествующими блоками исключения, поэтому такой блок должен размещаться последним).

Операторы catch – ловушки исключений, всегда следуют за оператором try, образуя единую синтаксическую конструкцию обработки исключений, порождаемых операторами throw в контролируемом блоке.

Набор блоков операторов catch в процессе выполнения операторов программы обходится, если в контролируемом блоке исключение не порождено.

Порождение исключения означает:

– создание копии объекта, представляющего выражение оператора trow (ограничений на тип выражения нет, но конструктор копирования должен быть доступен);

– вызов деструкторов объектов, порожденных от начала контролируемого блока до точки выброса исключения;

– поиск подходящего блока обработки исключения, выбираемого по **типу выражения** в операторе throw.

Если такой выбор успешно **однозначно** выполнен среди специализированных или наличного неспециализированного блока, то управление передается выбранному блоку подобно оператору вызова функции с одним параметром.

После исполнения операторов блока исключение считается обработанным и управление передается оператору, непосредственно расположенному за последним блоком catch активного контролируемого блока.

В случае неудачи выбора блока обработки исключения в некотором контролируемом блоке поиск последовательно продолжается в блоках предшествующих уровней вложенности (деструкторы порожденных локальных объектов продолжают также вызываться). Переход на предшествующий уровень вложенности может программироваться явно: оператор throw без выражения перебрасывает текущее исключение на такой же уровень с тем же значением выражения.

При неудаче поиска подходящего блока обработки исключения программа завершается аварийно – функция среды эпилога программы terminate по умолчанию организует вызов функции abort(). Можно назначить собственную функцию, которая будет вызвана перед завершением программы. Такое назначение реализуется определенной в файле eh.h функцией

```
terminate_function set_terminate(terminate_function);
```

Здесь тип terminate_function – указатель функции-обработчика аварийного выхода, которая не должна иметь параметры и возвращать результат:

```
typedef void (*terminate_function)();
```

Функция set_terminate может вызываться несколько раз, устанавливая новый обработчик и возвращая в качестве результата указатель прежнего обработчика. Пример управления исключениями в Microsoft Visual C++:

```
#include <iostream.h>
#include <eh.h>
#include <process.h>
class CTest {
   int m;
public:
   CTest():m(0) {
      cout<<endl<<"Ctest(0) создан...";
   }
   CTest(int i):m(i) {
      cout<<endl<<"Ctest("<&ltm<<") создан...";
   }
   CTest(CTest &t):m(-t.m) {
      cout<<endl<<"Ctest("<&ltt.m<<")<=Ctest("<&ltm<<")...";
   }

   ~CTest() {
      cout<<endl<<"Ctest("<&ltm<<") разрушен...";
      m=0;
   }

   friend ostream& operator<<(ostream &, CTest &);
};
```

```
ostream & operator<<(ostream &out, CTest &item) {
   out<<"Исключение CTest("<&ltitem.m<<")... ";
   return out;
}

class CDemo {
public:
   CDemo() { cout<<endl<<"Создание CDemo."; }
   ~CDemo() { cout<<endl<<"Разрушение CDemo."; }
};

void Aproc() {
   CDemo D;
   CTest X(2);
   cout<<endl<<"Aproc: Выброс исключения CTest(1).";
   throw CTest(1);
   cout<<endl<<"Aproc: Что после исключения CTest(1) ?";
}

void proc_1() {
   cout<<endl<<"Начало proc_1.";
   try {
      cout<<endl<<"Блок try, вызов Aproc.";
      Aproc();
   }

  catch (CTest E) {
     cout<<endl<<"Ловушка исключения типа Ctest." <<endl
         <<"Поймано исключение: "<<E;
  }
  catch (char *str) {
     cout<<endl<<"Поймано исключение типа (char *): "<<str;
  }
  cout<<endl<< "Завершение proc_1.";
}

void proc_2() {
   cout<<endl<<"Начало proc_2.";
   try {
    cout<<endl <<"Выброс исключения int(0).";
     throw int(0);
   }
   catch (CTest E) {
      cout<<endl<<"Ловушка исключения типа Ctest."<<endl
         <<"Поймано исключение: "<<E;
   }
   catch (char *str) {
      cout<<endl<<"Поймано исключение типа (char *): "<<str;
   }
   catch (...) {
      cout<<endl<<"Что это такое ??? Переброс дальше!";
      throw;
   }
   cout<<endl<<"Завершение proc_2.";
}
```

```
void term_func() {
   cout<<endl<<"Обработчик завершения программы."<<endl
         <<"Возврат в порождающий процесс...";
   exit (-1);
}

void main() {
   cout<<endl<<"Начало программы: шаг 1.";
   try {
      cout<<endl<<"Вызов proc_1 из контролируемого блока.";
      proc_1();
   }
   catch (...) {
      cout<<endl<<"Шаг 1 прерван каким-то исключением...";
   }
   cout<<endl<<"Продолжение программы: шаг 2.";
   try {
      cout<<endl<<"Вызов proc_2 из контролируемого блока.";
      proc_2();
   }
   catch (...) {
      cout<<endl<<"Шаг 2 прерван каким-то исключением...";
   }

   cout<<endl<<"Заключительный шаг программы."<<endl
        <<"Установлен обработчик аварийного завершения.";

   set_terminate(term_func);
   cout<<endl<<"Выброс непредусмотренного исключения...";
   throw 1;
   cout<<endl<<"Последнее сообщение программы";
}
```

Результаты работы программы:

```
Начало программы: шаг 1.
Вызов proc_1 из контролируемого блока.
Начало proc_1.
Блок try, вызов Aproc.
Создание CDemo.
Ctest(2) создан...
Aproc: Выброс исключения CTest(1).
Ctest(1) создан...
Ctest(1)<=Ctest(-1)...

Ctest(2) разрушен...
Разрушение CDemo.
Ловушка исключения типа Ctest.
Поймано исключение: Исключение CTest(-1)...
Ctest(-1) разрушен...
Ctest(1) разрушен...
Завершение proc_1.
Продолжение программы: шаг 2.
Вызов proc_2 из контролируемого блока.
Начало proc_2.
```

```
Выброс исключения int(0).
Что это такое ??? Переброс дальше!
Шаг 2 прерван каким-то исключением...
Заключительный шаг программы.
Установлен обработчик аварийного завершения.
Выброс непредусмотренного исключения...
Обработчик завершения программы.
Возврат в порождающий процесс...
```

8.2. Понятие структурного управления исключениями

В современных системах программирования на языках C/C++ поддерживается механизм так называемого структурного управления исключениями (Structured Exception Handling, SEH), где исключения идентифицируются только типом unsigned int. Структурное управление исключениями позволяет наряду с обработкой потока явно порождаемых программой исключений обрабатывать и исключения, порождаемые операционной системой в аварийных ситуациях. Этим объясняется наличие единственного типа идентификации исключений и, как следствие, единственного блока обработки исключений в контролируемом блоке программы.

Различают 2 разновидности структурного управления исключениями:

– кадрированное управление – блок обработки исключений активизируется только в момент порождения исключения;

– завершающее управление – любой вид выхода из контролируемого блока программы завершается активизацией предопределенного блока операторов.

Операторы контролируемого блока могут явно породить исключение, используя функцию

```
WINBASEAPI VOID WINAPI RaiseException (
   DWORD dwExceptionCode,
   DWORD dwExceptionFlags,
   DWORD nNumberOfArguments,
   CONST DWORD *lpArguments );
```

(здесь типы параметров соответствуют системе программирования Microsoft Visual C++).

Интерпретация параметров функции RaiseException:

- dwExceptionCode – код исключения;
- dwExceptionFlags – флаг возобновления исключения;
- nNumberOfArguments – количество аргументов детализации описания исключения в массиве lpArguments.

Предопределенные коды исключений:

```
EXCEPTION_ACCESS_VIOLATION
EXCEPTION_DATATYPE_MISALIGNMENT
EXCEPTION_BREAKPOINT
EXCEPTION_SINGLE_STEP
EXCEPTION_ARRAY_BOUNDS_EXCEEDED
```

```
EXCEPTION_FLT_DENORMAL_OPERAND
EXCEPTION_FLT_DIVIDE_BY_ZERO
EXCEPTION_FLT_INEXACT_RESULT
EXCEPTION_FLT_INVALID_OPERATION
EXCEPTION_FLT_OVERFLOW
EXCEPTION_FLT_STACK_CHECK
EXCEPTION_FLT_UNDERFLOW
EXCEPTION_INT_DIVIDE_BY_ZERO
EXCEPTION_INT_OVERFLOW
EXCEPTION_PRIV_INSTRUCTION
EXCEPTION_IN_PAGE_ERROR
EXCEPTION_ILLEGAL_INSTRUCTION
EXCEPTION_NONCONTINUABLE_EXCEPTION
EXCEPTION_STACK_OVERFLOW
EXCEPTION_INVALID_DISPOSITION
EXCEPTION_GUARD_PAGE
```

Таким образом, предопределенные исключения достаточно подробно представляют ошибки, обнаруживаемые операционной системой.

8.3. Кадрированное управление исключениями

Синтаксис определения кадрированного управления исключениями:

```
__try {
    //
    // Операторы контролируемого блока
    //
}
__except (выражение_фильтра) {
    //
    // Операторы блока обработки исключения
    //
}
```

Блок обработки исключения можно рассматривать как условный оператор, где решение о продолжении процесса определяется вычисляемым после порождения исключения выражением фильтра. Выражение фильтра может принимать одно из значений:

- EXCEPTION_EXECUTE_HANDLER (1) – обработать исключение;
- EXCEPTION_CONTINUE_SEARCH (0) – продолжение поиска обработчика исключения на предшествующем уровне вложенности оператора __try;
- EXCEPTION_CONTINUE_EXECUTION (–1) – возврат управления в точку выброса исключения.

Как в выражении фильтра, так и в блоке обработки исключения можно получить детальную информацию о причине исключения, вызывая функции:

- unsigned long GetExceptionCode(void) – код исключения;
- (struct _EXCEPTION_POINTERS *) GetExceptionInformation(void) – указатель на структуру с детальным описанием исключения.

Пример демонстрационной программы:

```
#include <afxwin.h>
#include <fstream.h>
ofstream x("test.txt");

void samples() {
   if (x) {
      x<<endl<<"Начало samples. Шаг 1.";
      __try {
         x<<endl<<"Контролируемый блок 1.";
           <<endl<<"Преднамеренное исключение: код 19970405";
      RaiseException(19970405L, EXCEPTION_NONCONTINUABLE,0,0);
      x<<endl<<"Завершение шага 1.";
   }  __except (x<<endl<<"Фильтр 1 активен", 1) {
      x<<endl<<"Прерывание шага 1: код "<&ltGetExceptionCode();
   }

   x<<endl<<"Продолжение samples. Шаг 2.";

   __try {
      x<<endl<<"Контролируемый блок 2.";
        <<endl<<"Демонстративное деление на нуль...";
      int i=2, j=0;
      i/=j;
      x<<endl<<"Завершение шага 2.";
   } __except (x<<endl<<"Фильтр 2 активен", 1) {
   x<<endl<<"Прерывание шага 2: код "
    <<GetExceptionCode();
   }

   x<<endl<<"Продолжение samples. Шаг 3.";

   __try {
      x<<endl<<"Контролируемый блок 3.";
        <<endl<<"Попытка нарушения защиты памяти...";
      int *p = 0; // Пустой указатель !
      *p = 1997;
      x<<endl<<"Завершение шага 3.";
   } __except (x<<endl<<"Фильтр 3 активен", 1) {
       x<<endl<<"Прерывание шага 3: код "
         <&ltGetExceptionCode();
   }
   x<<endl<<"Завершение samples";
  }
}

class DemoTryExcept: public CWinApp {
public:
   BOOL InitInstance() {
      AfxMessageBox("ДЕМОНСТРАЦИЯ ОБРАБОТКИ ИСКЛЮЧЕНИЙ");
      samples();
      return FALSE;
}
} main; // Аналог функции main() в Visual C++
```

Результаты работы программы (файл test.txt):

```
Начало samples. Шаг 1.
Контролируемый блок 1.
Преднамеренное исключение: код 19970405
Фильтр 1 активен
Прерывание шага 1: код 19970405
Продолжение samples. Шаг 2.
Контролируемый блок 2.
Демонстративное деление на нуль...
Фильтр 2 активен
Прерывание шага 2: код 3221225620
Продолжение samples. Шаг 3.
Контролируемый блок 3.
Попытка нарушения защиты памяти...
Фильтр 3 активен
Прерывание шага 3: код 3221225477
Завершение samples
```

8.4. Завершающее управление исключениями

Синтаксис определения завершающего управления исключениями:

```
__try {
   //
   // Операторы контролируемого блока
   //
} __finally {
   //
   // Операторы блока обработки факта завершения
   // контролируемого блока
   //
}
```

Завершение контролируемого блока может быть нормальным или преждевременно прерванным.

Причины досрочного выхода:

– выполнение оператора return, goto, break или continue;

– вызов функции, подобной longjump();

– порождение исключения;

– выполнение оператора выхода из контролируемого блока __leave;

Любой исход завершения контролируемого блока безусловно приводит к гарантированному выполнению операторов блока finally. Очевидно, что допускается совмещение кадрированного и завершающего управления исключениями. Пример демонстрационной программы:

```
#include <stdio.h>
void main() {
   puts("Начало программы");
   int *p = 0x00000000; // Пустой указатель !
```

```
__try {
    puts("Начало блока контроля исключения");
    __try {
       puts("Начало блока контроля завершения");
       puts("Попытка нарушения защиты памяти...");
       *p = 13;
       puts("Продолжение работы");
    } __finally {
    puts("Блок завершения активен");
    }
    puts("Конец блока контроля исключения");
    }__except(puts("Фильтр активен"), 1) {
    puts("Исключение обработано");
    }
    puts("Завершение программы");
}
```

Результаты работы программы:

```
Начало программы
Начало блока контроля исключения
Начало блока контроля завершения
Попытка нарушения защиты памяти...
Фильтр активен
Блок завершения активен
Исключение обработано
Завершение программы
```

В блоке __finally можно проверить состояние контролируемого блока – функция

```
BOOL AbnormalTermination(VOID);
```

возвращает истину, если контролируемый блок завершен нормально.

По соображениям эффективности кода в контролируемый блок не рекомедуется помещать операторы преждевременного выхода return, goto, break или continue.

8.5. Иерархическое управление исключениями

Сравнивая схемы структурной обработки исключений и стандартной схемы обработки исключений языка C++, можно заметить их сходство. Кадрированное управление практически воспроизводит схему обработки исключений языка C++. Однако структурная обработка исключений обеспечивается механизмами поддержки на более низком уровне, чем прикладная программа. В связи с этим обработка таких исключений не связана с высокоуровневым процессом планирования вызова конструкторов и деструкторов. Как следствие, в контролируемых блоках структурной обработки исключений объявление объектов класса не разрешается.

Производные классы в языке C++ образуют основу определения сложных иерархий объектов, но функции, использующие такие объекты, не лишены возможности быть источниками исключений. Схема управления исключениями остается неизменной. Однако механизм динамической связи базовых и производных классов в языке C++ через указатели и/или ссылки открывает путь для детализации процесса обработки исключений.

Пример демонстрационной программы:

```
class TIO { // Буфер ввода-вывода
protected:
   int data;
public:
   TIO():data(0) {}
   virtual void is_error() {
      cout<<endl<<"Какая-то ошибка ввода-вывода ????";
   }
};
class Read: public TIO { // Процесс ввода
public:
   Read() {}
   void read() {
      cout<<endl<<"Значения данных -? ";
      cin>>data;
      if (!cin) throw *this;
   }
   virtual void is_error() {
      cout<<endl<< "Ошибка ввода...";
   }
};
class Write: public Read { // Процесс ввода-вывода
public:
   Write() {}
   void write() {
      cout<<endl<<"Значение элемента данных: "<<data;
      if (data<0) throw *this;
   }
   virtual void is_error() {
      cout<<endl<< "Некорректное значение данных";
   }
};
class Work: public Write { // Создание данных
public:
   operator int() { return data!=0; }
   Work() {}
   virtual void is_error() {
      cout<<endl<< "Ошибка создания данных";
   }
};
void main() {
   try {
      Work x;
      x.read();
      x.write();
      if (!x) throw x;
```

```
    } catch (TIO& ioerror) { ioerror.is_error(); }
}
```

Здесь в контролируемом блоке выполняется несколько шагов преобразования данных, сохраняемых в буфере – представителе базового класса TIO. Многочисленные исключения, порождаемые разными уровнями наследования производными классами, обрабатываются единственной ловушкой со ссылкой на класс TIO. Однако виртуальная функция is_error(), определенная в каждом классе, будет вызвана в точном соответствии с причиной ошибки.

Некоторые функции из стандартных библиотек C++ выбрасывают исключения, которые могут быть обработаны. Эти исключения называются стандартными и описываются производными классами (наследниками) от класса std::exeption.

8.6. Порождение исключений в конструкторах и деструкторах

Отличительная черта конструкторов и деструкторов – отсутствие возможности возврата значений, что создает определенные неудобства для фиксации ошибки в процессе их явного либо неявного вызова. Выделение же в конструируемом объекте некоторого признака ошибочного состояния либо громоздко, либо принципиально невозможно. Порождение исключений в рассматриваемых функциях-элементах класса – элегантный прием моделирования возврата значений.

Пример демонстрационной программы:

```
#include <iostream.h>
class SomeClass {
   int variant; // Переключатель источника исключений
public:
   SomeClass(int i);
   ~SomeClass();
   class ErrorClass { // Класс ошибок
   public:
     int ErrorCode; // Код ошибки
     ErrorClass(int error): ErrorCode(error) {
       cout<<endl<<"КОД ОШИБКИ: "<<ErrorCode;
     }
   };
};
   // Версия конструктора с порождением исключений
SomeClass::SomeClass(int i):variant(i) {
   if (variant>0) throw ErrorClass(1);
}

// Версия деструктора с порождением исключений
SomeClass::~SomeClass() { if (variant<0) throw ErrorClass(2); }
  void main(void) {
   cout<<endl<<"Начало программы"<<endl;
   try {
```

```
        //...
        cout<<endl<<"Провоцирование исключения в конструкторе";
        SomeClass error_in_construct(1);
        cout<<endl<<"Объект \"error_in_construct\" создан...";
        //...
    } catch (SomeClass::ErrorClass)  {
          cout<<endl<<"Исключение в конструкторе обработано";
    }
    cout<<endl<<endl<<"Продолжение программы"<<endl;
    try {
        //...
        cout<<endl<<"Провоцирование исключения в деструкторе";
        SomeClass error_in_destruct(-1);
        cout<<endl<<"Объект \"error_in_destruct\" создан...";
        //...
    } catch (SomeClass::ErrorClass Error_Object) {
          cout<<endl
                  <<"Исключение в деструкторе обработано"
                  <<endl
                  <<"Код "<<Error_Object.ErrorCode<<" ???";
    }
    cout<<endl<<endl<<"Завершение программы";
}
```

Результаты работы программы:

```
Начало программы
Провоцирование исключения в конструкторе
КОД ОШИБКИ: 1
Исключение в конструкторе обработано
Продолжение программы
Провоцирование исключения в деструкторе
Объект "error_in_destruct" создан...
КОД ОШИБКИ: 2
Исключение в деструкторе обработано
Код 2 ???
Завершение программы
```

При порождении исключений в последней программе создается объект класса SomeClass::ErrorClass, размещаемый в стековой памяти. Так как здесь в первом операторе catch такой объект не идентифицируется, то отображение ошибки реализуется лишь выводом сообщения в конструкторе класса SomeClass::ErrorClass. Объект такого или подобного класса может быть использован для передачи детальной информации о причине порождаемого исключения из конструктора и/или деструктора.

8.7. Спецификация исключений в функциях

Спецификацией исключений называется перечисление всех исключений в определении и во всех объявлениях функции, которые она может выбросить.

Синтаксис:

```
возвращаемый_тип имя_функции(список_параметров)
 throw (список_типов_исключений)
```

Если список типов исключений пуст, то это значит, что функция не выбрасывает никаких исключений:

```
void foo(void) throw(char*,int);
void foo(void) throw();
```

Если в объявлении функции спецификация исключений отсутствует, то функция может выбросить исключение любого типа. Если функция выбрасывает исключение, тип которого не описан в спецификации исключений, то вызывается функция void unexpected() из стандартной библиотеки C++, которая вызывает в свою очередь функцию terminate(). Такие исключения называются неожидаемыми или непредусмотренными. Для обработки этих исключений можно установить «свой обработчик» при помощи описаной в файле <eh.h> функции

```
unexpected_handler set_unexpected(unexpected_handler) throw();
```

Спецификатор исключений можно также задавать при объявлении указателя на функцию. Присваиваемое такому указателю значение должно указывать на функцию, которая может выбросить только исключения (возможно не все) из спецификации исключений в указателе:

```
void (*pf)(void) throw (int,double);
void f() throw(int);
pf=f;
```

Схема перехвата непредусмотренных исключений:

```
#include <exception>
#include <iostream>
using namespace std;
void unfunction(){
   cout << "Unexpected returning." << endl;
   terminate();
}
void main() {
   unexpected_handler oldHand = set_unexpected(unfunction);
   unexpected( );
}
```

Связь механизмов обработки исключений C++ и структурного управления исключениями в среде Microsoft Visual C++ выполняет функция

```
typedef void (*_se_translator_function)
            (unsigned int, struct _EXCEPTION_POINTERS* );

_se_translator_function _set_se_translator(
 _se_translator_function seTransFunction);
```

9. ДИНАМИЧЕСКАЯ ИДЕНТИФИКАЦИЯ И ПРИВЕДЕНИЕ ТИПА

9.1. Динамическая идентификация типа

Динамическая идентификация типа базируется на информации о типе во время исполнения программы (Runtime Type Information, RTTI). Поддержка RTTI введена в C++ сравнительно недавно и реализована, например, в системах программирования Borland C++ 4.0, Microsoft Visual C++ 4.0 и их более поздних версиях. Основу механизма поддержки RTTI составляют:

– определенный в файле typeinfo.h класс Type_info;

– специальная операция typeid, результат выполнения которой – объект типа const Type_info &.

Необходимый для понимания RTTI фрагмент определения класса Type_info имеет вид

```
class Type_info {
public:
   virtual ~Type_info();
   int operator==(const Type_info &) const;
   int operator!=(const Type_info &) const;

   int before(const Type_info&) const;
   const char* name() const;
   const char* raw_name() const;
private:
// Запрет создания и копирования объектов пользователем
   Type_info(const Type_info &);
   Type_info& operator=(const Type_info &);
   //
   // Определение других элементов
   //
};
```

Назначение неочевидных по определению элементов класса Type_info:

- before – сравнение имен типов посредством посимвольного сопоставления;
- name – имя типа в тексте программы;
- raw_name – расширенное (декорированное) имя типа – обычное имя типа дополнено системно-зависимой информацией об особенностях типа.

Синтаксис операции typeid:

```
typeid(имя_типа)
typeid(выражение)
```

Таким образом, операция typeid применима как к объекту – результату вычисления выражения, так и к идентификатору типа. Так как ее использование порождает определенные надстройки в коде объектного модуля программы, то

поддержка RTTI обычно требует запроса соответствующего режима компиляции (ключ /GR в Microsoft Visual C++).

Демонстрационный пример идентификации типов:

```
#include <iostream.h>
#include <typeinfo.h>
class test { int data; /* ... */ };

void main() {
   int  i, *pi=&i, &ri=i;
   test x, *px=&x, &rx=x;

   cout<<endl<<"  1. "<<typeid(i).name()
       <<endl<<"  2. "<<typeid(pi).name()
       <<endl<<"  3. "<<typeid(*pi).name()
       <<endl<<"  4. "<<typeid(ri).name()
       <<endl<<"  5. "<<typeid(int).name()
       <<endl<<"  6. "<<typeid(double *).name()
       <<endl<<"  7. "<<typeid(1997).name()
       <<endl<<"  8. "<<typeid(x).name()
       <<endl<<"  9. "<<typeid(*px).name()
       <<endl<<" 10. "<<typeid(rx).name()
       <<endl<<" 11. "<<typeid(test).name();
}
```

Результаты работы программы:

```
 1. int
 2. int *
 3. int
 4. int
 5. int
 6. double *
 7. int
 8. class test
 9. class test
10. class test
11. class test
```

Можно заметить, что правила формирования имени типа для базовых типов и классов различаются. Действительная полезность RTTI проявляется для **полиморфных** типов, т. е. классов с виртуальными функциями. Для таких типов указатели и ссылки существенно определяют выбор наследуемых **производными** классами функций. Однако значение указателя не содержит информацию о типе реально существующего объекта. Подобная информация, как результат операции typeid, часто оказывается весьма полезной при программировании на основе иерархии производных классов.

Пример программы выявления и сравнения типов объектов во время исполнения программы:

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

```
#include <typeinfo.h>
// Базовый класс без виртуальных функций
class C1_base {
   int data_b;
public:
   // ...
};

// Производный класс без переопределения виртуальных функций
class C1_derived: public C1_base {
   int data_d;
public:
   // ...
};

// Базовый класс с виртуальными функциями (полиморфный класс)
class C2_base {
   int data_b;
public:
   virtual ~C2_base() {}
   // ...
};

// Производный класс с переопределением виртуальных функций
class C2_derived: public C2_base {
   int data_d;
public:
   // ...
};

void main(void) {
   C1_derived d1; // Объект производного класса
   C1_base &br1=d1; // Ссылка на объект производного класса
   cout<<typeid(br1).name()<<endl
       <<typeid(C1_base).name()<<endl
       <<typeid(C1_derived).name()
       <<endl;
   cout<<"typeid(br1) "
       <<(typeid(br1)==typeid(C1_base)? "==":"!=")
       <<" typeid(C1_base)"
       <<endl;
   cout<<"typeid(br1) "
       <<(typeid(br1)==typeid(C1_derived)? "==":"!=")
       <<" typeid(C1_derived)"
       <<endl;
   C2_derived d2; // Объект производного класса
   C2_base &br2=d2; // Ссылка на объект производного класса
   cout<<endl<<typeid(br2).name()<<endl
       <<typeid(C2_base).name()<<endl
       <<typeid(C2_derived).name()
       <<endl;
   cout<<"typeid(br2) "
       <<(typeid(br2)==typeid(C2_base)? "==":"!=")
       <<" typeid(C2_base)"
       <<endl;
   cout<<"typeid(br2) "
```

```
        <<(typeid(br2)==typeid(C2_derived)? "==":"!=")
        <<" typeid(C2_derived)"
        <<endl;
}
```

Результаты работы программы:

```
class C1_base
class C1_base
class C1_derived
typeid(br1) == typeid(C1_base)
typeid(br1) != typeid(C1_derived)

class C2_derived
class C2_base
class C2_derived
typeid(br2) != typeid(C2_base)
typeid(br2) == typeid(C2_derived)
```

Очевидно различие в результатах операции typeid в зависимости от наличия виртуальных функций в базовых классах. Здесь класс C1_base не является полиморфным и ссылка на него (br1) соответствует имени его типа. Класс C2_base является полиморфным, а ссылка на него (br2) через производный класс соответствует имени типа, использованного для инициализации ссылки объекта производного класса.

Операция typeid для полиморфных классов может применяться как альтернатива виртуальным функциям:

```
#include <typeinfo.h>
class Base {
   //...
public:
   virtual ~Base() {} // Полиморфный класс
};

class Derive_1: public Base {
public:
   void action() { /* ... */ }
};
class Derive_2: public Base {
public:
   void action() { /* ... */ }
};
class Derive_N: public Base {
public:
   void action() { /* ... */ }
};

void selector(Base *pobj) {
   //...
   Type_info &obj_type=typeid(pobj);
   if (obj_type==typeid(Derive_1))
      ((Derive_1 *)pobj)->action();
   else if (obj_type==typeid(Derive_2)) action_2();
```

```
        ((Derive_2 *)pobj)->action();
    //...
    else if (obj_type==typeid(Derive_N)) action_N();
        ((Derive_N *)pobj)->action();

    //...
}
```

Однако использование механизма позднего связывания виртуальных функций более элегантно и эффективно в вычислительном отношении:

```
#include <typeinfo.h>

class Base {
    //...
public:
   virtual ~Base() {} // Полиморфный класс
   virtual void action() {} // Переопределяемая виртуальная
                            // функция может быть строго
                            // виртуальной
};

class Derive_1: public Base {
public:
   virtual void action() { /* ... */ }
};

class Derive_2: public Base {
public:
   virtual void action() { /* ... */ }
};

//...

class Derive_N: public Base {
public:
   virtual void action() { /* ... */ }
};

void selector(Base *pobj) {
   //...
   pobj->action(); // Автоматическая классификация возможна
                   // из-за наличия виртуальной функции
                   // action в базовом классе
   //...
}
```

Очевидно, что классификация функций на основе обращения к операции typeid вместо виртуальных функций удобна в случаях невозможности либо нежелания модификации базового класса. Использование операции typeid часто можно исключить, применяя вместо этого динамическое приведение типа (см. подразд. 9.3).

9.2. Обзор новых возможностей приведения типа

Традиционное приведение типа в языках C и C++ имеет следующие существенные недостатки:

– разрешение программисту производить не контролируемые компилятором преобразования типов, которые могут породить ошибку на этапе выполнения программы;

– синтаксис традиционных операций приведения типа не отражает действительные намерения программиста;

– указатель на базовый класс не может быть явно преобразован в указатель на производный класс, в котором базовый класс объявлен виртуальным.

Пример демонстрационной программы:

```
// Некорректные последствия преобразования типов
int *pvi=new int[512];
const int *pci=pvi;
delete[] (int *)pci; // Подавление сообщения об ошибке
// Недопустимость преобразования указателей
class Base {};
class D1: public virtual Base {};
class D2: public virtual Base {};
class D12: public D1, public D2 {};
void f(Base *pb) {
   D12* dp=(D12 *)pb; // Подавление сообщения об ошибке
   //...
}

void main() {
   D12 d_object;
   //...
   f(&d_object);
}
```

Новые возможности приведения типа устраняют указанные недостатки путем использования:

– однозначно интерпретируемого синтаксиса операций;

– контроля ошибок как на этапе компиляции, так и на этапе выполнения программы;

– разрешения преобразования указателей на базовый виртуальный класс в указатели на наследующий его производный класс.

9.3. Динамическое приведение типа

Динамическое приведение типа опирается на механизм RTTI и позволяет автоматизировать контроль корректности преобразования указателей и ссылок на объекты полиморфных классов без явного применения операции typeid.

Синтаксис операции динамического приведения типа:

```
dynamic_cast<type>(expression)
```

Здесь type – тип, а expression – выражение, подлежащее преобразованию к типу type, удовлетворяющие следующим требованиям:

– type должен определять указатель либо ссылку (допустимо использование типа void *);

– преобразуемое выражение expression должно быть указателем, если type – указатель; в противном случае expression может быть только ссылкой.

Результат успешно выполненной операции dynamic_cast<type> имеет тип type. Неудача преобразования проявляется в зависимости от вида модификатора в определении типа type:

– возврат нулевого указателя, если type – указатель;

– порождение исключения bad_cast, если type – ссылка.

Класс bad_cast определен в файле typeinfo.h в следующем виде:

```
typedef const char *__exString;
class  exception {
public:
   exception();
   exception(const __exString&);
   exception(const exception&);
   exception& operator=(const exception&);
   virtual ~exception();
   virtual __exString what() const;
private:
   __exString _m_what;
   int _m_doFree;
};

class bad_cast: public exception {
public:
   bad_cast(const __exString& what_arg):exception(what_arg) {}
};
```

Наследуемая из базового класса exception функция-элемент what() возвращает строку комментария причины исключения. Пример программы нисходящего приведения типа:

```
#include <iostream.h>
#include <eh.h>
#include <typeinfo.h>
class A {
public:
   virtual ~A() {}
};

class B: public A {};

class B1: virtual public A {};
class B2: virtual public A {};
```

```
class C: public B1, public B2 {};

void f1(A *pa) { // Приведение ссылки
   try {
      cout<<endl<<endl<<"R-test: ";
      B& rb=dynamic_cast<B&>(*pa);
      cout<<"OK";
   } catch (bad_cast &x) {
       cout<<x.what();
   }
}

void f2(A *pa) { // Приведение указателя
   try {
      cout<<endl<<"P-test: ";
      B* pb=dynamic_cast<B *>(pa);
      if (!pb) throw "Bad cast";
      cout<<"OK";
   } catch (char *msg) {
        cout<<msg;
   }
}

void f3(A *pa) { // Приведение указателя
   try {
      cout<<endl<<"P-test: ";
      C* pb=dynamic_cast<C *>(pa);
      if (!pb) throw "Bad cast";
      cout<<"OK";
   } catch(char *msg) {
       cout<<msg;
   }
}

void main() {
   A* pa=new A; // Указатель на объект базового класса

   f1(pa);      // Преобразование в объект производного
   f2(pa);      // класса должно быть запрещено...
   delete pa;
   pa=new B;    // Указатель на объект базового класса в
                // объекте производного класса

   f1(pa);      // Преобразование в объект производного
   f2(pa);      // класса должно быть разрешено...
   delete pa;

   pa=new C;    // Указатель на объект базового класса в
                // объекте производного класса

   f1(pa);      // Преобразование в объект производного
   f2(pa);      // класса должно быть разрешено...
   f3(pa);      // Преобразование в объект производного
                // класса должно быть разрешено...
   delete pa;
}
```

Результаты работы программы:

```
R-test: Bad dynamic_cast!
P-test: Bad cast

R-test: OK
P-test: OK

R-test: Bad dynamic_cast!
P-test: Bad cast
P-test: OK
```

Здесь продемонстрировано и нисходящее приведение типа виртуального базового класса.

Одно из полезных применений операции dynamic_cast – так называемое перекрестное приведение типа, позволяющее безопасно связывать классы, созданные в разные моменты времени:

```
#include <iostream.h>
class my_out {
public:
   virtual ~my_out() {}
   void output(char *s) { cout<<endl<<s; }
};

class Complex{
protected:
   int re, im;
public:
   Complex(int r=0, int i=0):re(r), im(i) {}
   friend ostream & operator<<(ostream &out,Complex &x);
   virtual ~Complex() {}
};
//
//  Переопределение операции вывода основано на несвязанных до
//  этого момента классах Complex и my_out
//
ostream & operator<<(ostream &out,Complex &x) {
   my_out *px=dynamic_cast<my_out *>(&x);
   if (px) px->output("Class Complex: ");
   out<<"re="<<x.re<<", im="<<x.im;
   return out;
}
//
//  Варианты связей классов Complex и my_out
//
class test_1: public Complex, public my_out { // Вариант 1
public:
   test_1(int r=0, int i=0):Complex(r,i) {}
};

class test_2: public my_out { };

class test_3: public Complex, public test_2 { // Вариант 2
```

```
public:
   test_3(int r=0, int i=0):Complex(r,i) {}
};

void main() {
   cout<<Complex(1,2);
   cout<<test_1(3,4);
   cout<<test_3(5,6);
}
```

Результаты работы программы:

```
re=1, im=2
Class Complex: re=3, im=4
Class Complex: re=5, im=6
```

Подобно ранее рассмотренному примеру использования операции typeid, операция dynamic_cast позволила отказаться от модификации базового класса.

9.4. Статическое приведение типа

Ранее рассмотренное динамическое приведение типа действительно полезно для полиморфных типов, но в других случаях может быть заменено более точно отражающим существо задачи статическим приведением. Реализующая статическое приведение новая в языке C++ операция

```
static_cast<идентификатор_типа>(выражение)
```

преобразует выражение к заданному типу на этапе компиляции. Статическое приведение типа не предполагает проверку безопасности преобразования на этапе исполнения. Примеры использования операции static_cast:

```
int i;
long l=static_cast<long>(i);
float f=static_cast<float>(i);
enum set {One, Two, Tree, Four,Five} estim;
char c=static_cast<char>(estim);
Base *pb=static_cast<Base *>(&d);
Derived &dr=static_cast<Derived &>(b);
```

Статическое приведение типа по последствиям аналогично традиционному приведению типа, но использование синтаксиса операции static_cast означает отказ от использования RTTI и, что более важно, отражает явно намерения программиста. Преобразования указателей и ссылок на объекты иерархии производных классов рекомендуется выполнять посредством операции dynamic_cast, которая более надежна и универсальна. Для безопасных вариантов ее реализация полностью совпадает с реализацией операции static_cast.

9.5. Преобразования типа с сохранением значений

Для пояснения намерений программиста, кроме рассмотренных выше, в языке C++ существуют другие виды операций преобразования типа:

– const_cast<идентификатор_типа>(выражение) – игнорирование атрибутов const, volatile или __unaligned заданного выражения;

– reinterpret_cast<идентификатор_типа>(выражение) – изменение точки зрения компилятора на тип объекта без модификации объекта.

Результат этих операций – преобразованное к заданному типу значение указанного выражения.

Пример использования операции const_cast:

```
int *ipv=new int[100];
//...
const int *ipc=ipv;
//...
delete[] const_cast<int *>(ipc);
```

Последний оператор при традиционном приведении типа в форме

```
delete[] (int *)(ipc);
```

не отражал бы действительную сущность преобразования.

Пример использования операции reinterpret_cast:

```
unsigned long dwData=0x01020304L;
//...
char *pointer;
//...
dwData=reinterpret_cast<unsigned long>(pointer);
//...
pointer=reinterpret_cast<char *>(dwData);
```

Ответственность программиста за корректность преобразования при этом не снимается:

```
class Base {} base_object;
class Derived: public Base {};
Derived *pd=reinterpret_cast<Derived *>(&base_object);
```

Последний оператор не будет считаться синтаксически неверным, но использование результата преобразования далее наверняка приведет к нежелательному исходу. Однако намерение программиста зафиксировано явно, что облегчает поиск ошибок.

ЗАКЛЮЧЕНИЕ

Язык C++ считается одним из сложнейших для изучения и использования [9, 12]. Хотя набор синтаксических конструкций, расширяющий язык C, незначителен, трудность заключается в том, что язык C++ ориентирован на совершенно отличную от технологии структурно-организованного процедурного программирования технологию объектно-ориентированного программирования. Идеология объектно-ориентированного проектирования программ – отдельный вопрос, требующий самостоятельного изучения [1–8].

Строго говоря, язык C++ для объектно-ориентированного стиля программирования не является идеальным, но его популярность объясняется наследованной от языка C чрезвычайной гибкостью и мобильностью. Трудности же освоения языка для его профессионального использования, естественно, приходится преодолевать. Рассмотренный в настоящем пособии материал отражает основные концепции построения и возможности новых языковых средств систем программирования на языке C++. Следует отметить, что их реализация в разных системах программирования отличается некоторыми деталями. В популярных в последнее время системах программирования фирм Borland и Microsoft поддерживаются все упомянутые в представленном пособии возможности. Механизм шаблонов в настоящее время широко используется в библиотеках классов STL. Обработка исключений, динамическая идентификация типа и новые формы операций приведения типа интенсивно применяются в библиотеках Microsoft Foundation Class (MFC) [7, 8].

Язык C++ к настоящему времени продолжает развитие [11, 15–16]. В современных системах программирования на языке C++ доминирует тенденция использования дружественных интегрированных сред разработки программ. Детальные сведения об элементах языка, особенностях реализации можно получить, пользуясь средствами оперативной подсказки. Однако необходимый начальный набор базовых сведений требуется накопить, изучив представленное пособие и/или упомянутые литературные источники.

ЛИТЕРАТУРА

1. Буч, Г. Объектно-ориентированный анализ и проектирование с примерами применения / Г. Буч; пер. с англ. – М. : Вильямс, 2012. – 720 с.

2. Приемы объектно-ориентированного проектирования. Паттерны проектирования / Э. Гамма [и др.]; пер. с англ. – СПб. : Питер, 2013. – 368 с.

3. Голуб, А. И. С и С++. Правила программирования / А. И. Голуб. – М. : БИНОМ, 1995. – 272 с.

4. Грэхем, И. Объектно-ориентированные методы. Принципы и практика / И. Грэхем; пер. с англ. – 3-е изд. – М. : Вильямс, 2013. – 736 с.

5. Ларман, К. Применение UML 2.0 и шаблонов проектирования / К. Ларман; пер. с англ. – М. : Вильямс, 2007. – 736 с.

6. Рамбо, Дж. UML 2.0. Объектно-ориентированное моделирование и разработка / Дж. Рамбо, М. Блаха; пер. с англ. – 2-е изд. – СПб. : Питер, 2007. – 544 с.

7. Скотт, К. UML. Основы. Краткое руководство по стандартному языку объектного моделирования / К. Скотт, М. Фаулер; пер. с англ. – 3-е изд. – СПб: Символ-Плюс, 2009. – 192 с.

8. Соммервилл, И. Инженерия программного обеспечения / И. Соммервилл; пер. с англ. – 6-е изд. – М. : Вильямс, 2002. – 624 с.

9. Страуструп, Б. Язык программирования С++. Специальное издание / Б. Страуструп; пер. с англ.– М. : БИНОМ, 2012. – 1136 с.

10. Шилдт, Г. Полный справочник по С++ / Г. Шилдт; пер. с англ. – М. : Вильямс, 2011. – 800 с.

11. Александреску, А. Современное проектирование на С++. Обобщенное программирование и прикладные шаблоны проектирования / А. Александреску; пер. с англ. – М. : Вильямс, 2008. – 336 с.

12. Троелсен, Э. Язык программирования C# 5.0 и платформа .NET 4.5 / Э. Троелсен; пер. с англ. – М. : Вильямс, 2013. – 1312 с.

13. Корнелл, Г. Java 2. В 2 т. Т. 1 : Основы. Библиотека профессионала / Г. Корнелл, К. Хорстманн; пер. с англ. – М. : Вильямс, 2013. – 816 с.

14. Лафоре, Р. Объектно-ориентированное программирование в С++ / Р. Лафоре; пер. с англ. – СПб. : Питер, 2007. – 928 с.

15. Уилсон, М. Расширение библиотеки STL для С++. Наборы и итераторы / М. Уилсон; пер. с англ. – СПб. : BHV, 2008. – 608 с.

16. Сейни, А. С++ и STL : справочное руководство / А. Сейни, Д. Мюссер, Ж. Дердж; пер. с англ. – М. : Вильямс, 2010. – 432 с.

Св. план 2013, поз. 23

Учебное издание

Ревотюк Михаил Павлович

ОБЪЕКТНО-ОРИЕНТИРОВАННОЕ ПРОГРАММИРОВАНИЕ И ПРОЕКТИРОВАНИЕ

В 2-х частях

Часть 1

ТЕХНОЛОГИИ ОБЪЕКТНОГО ПРОГРАММИРОВАНИЯ

УЧЕБНО-МЕТОДИЧЕСКОЕ ПОСОБИЕ

Редактор *Е. И. Герман*
Корректор *Е. Н. Батурчик*
Компьютерная правка, оригинал-макет *А. А. Лысеня*

Подписано в печать 28.03.2014. Формат 60x84 1/16. Бумага офсетная. Гарнитура «Таймс».
Отпечатано на ризографе. Усл. печ. л. 11,5. Уч.-изд. л. 10,0. Тираж 100 экз. Заказ 251.

Издатель и полиграфическое исполнение: учреждение образования
«Белорусский государственный университет информатики и радиоэлектроники»
ЛИ №02330/0494371 от 16.03.2009. ЛП №02330/0494175 от 03.04.2009.
220013, Минск, П. Бровки, 6