Министерство образования и науки Российской Федерации
Озерский технологический институт
(филиал)
Московского инженерно-физического института
(технического университета)

Кафедра прикладной математики

ООП на C++

Методическое пособие

Озерск, 2005

ООП на C++. Методическое пособие.
Подготовил В. Пономарев.
Озерск: ОТИ МИФИ, 2005. — 71 с.

Пособие предназначено для изучения основ традиционного объектно-ориентированного подхода в программировании студентами направления «Информатика и вычислительная техника». Предполагается, что знакомство с основами программирования вообще и на языке Си в частности изучается в курсе общеобразовательной дисциплины «Алгоритмические языки и программирование», поэтому здесь на рассматриваются аспекты синтаксиса и семантики языка Си, который лежит в основе Си++. Основной целью пособия является пояснение принципов, положенных в основу объектно-ориентированного подхода, таких, как инкапсуляция, наследование, полиморфизм и сопутствующих им понятий: целостности типа, абстракции и сокрытия данных, механизма позднего связывания. Тем не менее, пособие описывает большинство языковых средств Си++ и может служить справочником по этому языку.

Изучаемые понятия демонстрируются многочисленными примерами кода (178 примеров). Все примеры проверены в средах программирования Borland C++ версии 3.1 и Microsoft Visual Studio версии 6.0.

Озерский технологический институт (филиал) МИФИ (ОТИ МИФИ), 2005

Содержание

Отличия языка Си++ от Си

Ключевые слова

В Си++ добавлены следующие зарезервированные (ключевые) слова:

```
asm         catch       class       const       delete      friend
inline      new         operator    private     protected   public
template    this        throw       try         virtual
```

Константы

Для определения констант в Си++ используется директива **const**:

```
const csSomeStr = "Some Text";
```

В реализации языка Си Borland C++ ключевое слово **const** также может быть использовано для описания констант, но интерпретируются эти константы, как переменные, значения которых нельзя изменить. Си++, напротив, рассматривает переменные со словом **const** как истинные константные выражения. Поэтому в Си++ переменную-константу можно применять для спецификации размера массива:

```
const N = 2 * 220;      /* в Си Borland C++ допустимо */
unsigned a[N];          /* в Си недопустимо */
```

Одно из назначений слова **const** в Си Borland C++ — защита от изменения параметра функции, который передается через указатель. Например, один из параметров функции — указатель на строку (массив знаков). Поскольку имя массива является указателем, при передаче его как параметра внутри функции имеется возможность изменения значения строки. Ключевое слово **const** делает это невозможным (возникает ошибка времени компиляции). В следующем примере функция **stlen** вычисляет длину строки. Ключевое слово **const** предохраняет строку от непреднамеренного изменения.

```
int stlen(const char *a)
{
    int i = 0;
    while (a[i] != 0) i++;
    return i;
}
void main(void)
{
   char s[] = "Hello";
   int i = stlen(s);
}
```

Определения переменных

В языке Си локальные переменные объявляются только в начале блока. В Си++ переменная может быть объявлена в любом месте блока. Это снижает вероятность ошибок, так как переменную можно определить в непосредственной близости от кода, в котором она используется:

```
for (int i = 0; i < n; i++) a[i] = i;
```

Аргументы (функции) по умолчанию

В Си++ можно указывать значения по умолчанию для некоторых параметров функции. При вызове такой функции эти параметры можно опускать; компилятор подставит вместо них значения по умолчанию. Если параметр функции имеет значение по умолчанию, все последующие параметры также должны иметь их. Если при вызове функции параметр опускается, то должны быть опущены и все параметры справа. Значения по умолчанию указываются только в прототипе функции:

```
/* прототип со значениями по умолчанию */
int func(int i, int j = 1, int k = 2);
int func(int i, int j, int k)
{
   return i + j + k;
}
void main(void)
{
   int i, j, k;
   i = func(1, 3);
   printf("\n%d\n", i);
}
```

В качестве значений по умолчанию можно использовать не только константы, но и глобальные переменные и значения, возвращаемые функциями.

Ссылки

Ссылка — это указатель на объект, не требующий разыменования при применении. Для определения ссылки применяется унарный оператор **&**. Ссылки используются в основном как параметры и возвращаемые значения функций. Однако ссылка может использоваться и как псевдоним (другое имя) переменной, например:

```
void main(void)
{
   int j = 2;
   int &r = j;     /* определение ссылки */
   j += 8;         /* j = 10 */
   r /= 2;         /* j = 5  */
}
```

В отличие от указателя, ссылка-псевдоним должна быть инициализирована во время описания, после чего ей нельзя присвоить значение адреса другого объекта.

Параметры-ссылки

Использование ссылки в качестве параметра функции, значение которого изменяется функцией, более удобно, чем использование для этой же цели указателя. В качестве примера здесь приведены две функции, меняющие значения двух переменных — с использованием указателей и ссылок:

```
void swapp(int *a, int *b)
{
   int c = *a; *a = *b; *b = c;
}
```

```
void swapr(int& a, int& b)
{
   int c = a; a = b; b = c;
}
void main(void)
{
   int i = 1, j = 2;
   swapp(&i, &j);
   swapr(i, j);
}
```

Ссылки удобны также при передаче внутрь функции больших структур, так как передача адреса более эффективна, чем копирование структуры целиком. В этом случае для защиты структуры от изменения внутри функции (если это не предусматривается смыслом самой функции) перед формальным параметром функции следует указать ключевое слово **const**, которое вызовет сообщение об ошибке времени компиляции при попытке изменить значение структуры.

```
struct rec {
   int id;
   char name[20];
};

void printrec(const rec& a)
{
   printf("%d : %s\n", a.id, a.name);
}

void main(void)
{
   rec r = { 1, "Иванов И.И." };
   printrec(r);
}
```

Возвращаемое значение-ссылка

Если возвращаемое значение функции объявлено как ссылка, функция может быть использована как справа, так и слева от знака присваивания. В следующем примере функция **value** возвращает элемент глобального массива как ссылку. Если функция используется в правой части оператора присваивания, она возвращает значение элемента массива, индекс которого указывается как параметр функции. Если функция используется в левой части, она устанавливает значение элемента:

```
const size = 10;
static int a[size];

int& value(int index)
{
   if (index < 0 || index > size) index = 0;
   return a[index];
}

void main(void)
{
   for (int i = 0; i < size; i++) value(i) = i + 1;
   for (i = 0; i < size; i++) printf("%d\n", value(i));
}
```

Так как массив объявлен как статический, доступ к нему из других модулей программы возможен только посредством функции **value**, которая проверяет допустимость индекса. Это абстрагирует пользователя от структуры данных и позволяет избежать ошибок во время исполнения программы.

Подставляемые (inline) функции

Если функция объявлена с модификатором **inline**, компилятор по возможности будет компилировать тело функции вместо ее вызова. Это увеличивает код программы, но ускоряет ее работу, так как снижаются затраты на выполнение вызова. Как правило, в качестве подставляемых используют небольшие (**inline** — в строку) по размеру функции, например:

```
inline void swap(int& a, int& b) {
   a ^= b ^= a ^= b;
}
void main(void)
{
   int i = 1, j = 2;
   swap(i, j);
   printf("i = %d, j = %d\n", i, j);
}
```

Внутри **inline**-функции не допускается использование операторов **switch**, **for**, **do**, **while** и **goto** а также использование кода на ассемблере, иначе функция будет расширена. Подставляемая функция не может иметь параметров по умолчанию.

Определение **inline**-функции обязательно должно предшествовать ее вызову. Функция, описанная в прототипе как подставляемая, не будет расширяться при компиляции, пока не встретится ее определение как подставляемой функции:

```
inline void swap(int&, int&);
void f(int&, int&);
void main(void)
{
   int i = 1, j = 2;
   swap(i, j); /* не расширяется */
   printf("%d\n%d\n", i, j);
   f(i, j);
   printf("%d\n%d\n", i, j);
}
inline void swap(int& a, int& b) { a ^= b ^= a ^= b; }
void f(int& a, int& b)
{
   swap(a, b); /* расширяется */
}
```

Примечание: **inline**-функция не расширяется, если установлен параметр компилятора "*out-of-line inline functions*" (Borland C++).

Метки enum, struct и union

В языке Си++ метка в описании **enum**, **struct** и **union** является именем типа, поэтому при определении переменных нет необходимости указывать ее:

```
   enum bool { false, true };
   bool f;
```

Анонимные объединения

Если при определении объединения не указана метка, объединение называется анонимным. Доступ к элементам анонимного объединения осуществляется как к обычным переменным. Их можно использовать для экономии памяти:

```
static union {
   char name[40];
   int numbs[20];
};
void main(void)
{
   strcpy(name, "Name");
   printf("%s\n", name);
   for (int i = 0; i < 20; i++) numbs[i] = i;
}
```

Глобальные анонимные объединения должны объявляться как статические.

Перегрузка функций

В языке Си каждая функция должна иметь уникальное имя. В Си++ две и более функций могут иметь одно и то же имя, но при этом они должны различаться типами используемых аргументов. Это позволяет улучшить код за счет того, что функции, выполняющие одинаковые действия над разными типами, объявляются с использованием одного и того же (перегруженного) имени. Компилятор выбирает нужную функцию в зависимости от типов подставляемых аргументов:

```
void print(int a)
{
   printf("%d\n", a);
}

void print(const char *a)
{
   printf("%s\n", a);
}

void main(void)
{
   print(2);
   print("Hello");
}
```

Примечание: Функции, различающиеся только типом возвращаемого значения, не могут быть перегруженными. Ключевое слово `const` и знак ссылки `&` не изменяют тип параметра.

Операция разрешения видимости ::

В языке Си локальная переменная скрывает внутри блока глобальную переменную с тем же именем, например:

```
int i = 0;

void main(void)
{
   int i = 5;
   printf("%d\n", i);   /* выводит 5 */
}
```

Переменная `i`, расположенная вне основной функции, недоступна внутри нее, так как внутри блока **`main`** объявлена переменная с таким же именем. В Си++ доступ к глобальной переменной внутри блока разрешает операция разрешения видимости `::` (два двоеточия). Следующий пример поясняет это:

```
int i = 0;

void main(void)
{
   int i = 1;
   printf("%d\n", i);   /* выводит 1 */
   printf("%d\n", ::i); /* выводит 0 */
}
```

Операция разрешения видимости разрешает доступ только к глобальной переменной. В следующем примере внутри дополнительного блока в основной функции операция `::` осуществляет доступ к переменной вне функции **`main`**:

```
int i = 0;

void main(void)
{
   int i = 1;
   printf("%d\n", i);       /* выводит 1 */
   printf("%d\n", ::i);     /* выводит 0 */
   {
      int i = 2;
      printf("%d\n", i);    /* выводит 2 */
      printf("%d\n", ::i);  /* выводит 0 */
   }
}
```

Операторы распределения памяти

Для выделения динамической памяти в языке Си используются функции **`calloc`**, **`malloc`** и **`free`**. Их можно применять и в Си++, однако лучше использовать новые операции **`new`** и **`delete`**. Операция **`new`** возвращает указатель на тип, для которого выделяется память, в то время как **`malloc`** возвращает указатель **`void*`** (пустой указатель), который требует явного приведения к типу в Си++ (и не требует в Си). Кроме того, в Си++ есть библиотечная функция **`set_new_handler`**, которую можно использовать для определения пользовательской функции обработки ошибки во время выделения памяти. В следующем примере показано, как использовать операцию **`new`** для выделения памяти под переменную целого типа и для строки символов:

```
#include <stdio.h>
#include <string.h>
void main(void)
{
   int *p = new int;        /* выделение памяти для целого типа */
   *p = 1;
   printf("%d\n", *p);
   char *s = new char[6];   /* выделение памяти для массива */
   strcpy(s, "Hello");
   printf("%s\n", s);
   delete p;                /* высвобождение памяти */
   delete s;
}
```

Ниже показано, как установить обработчик ошибки распределения памяти:

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

void NewHandler(void)
{
   printf("Not enough memory\n");
}

void main(void)
{
   set_new_handler(NewHandler);
}
```

Перегрузка операций new и delete

Программист может определить собственные функции для выделения и освобождения памяти, перегрузив операции **new** и **delete**:

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

void * operator new(size_t size)
{
   printf("Request for bytes: %d\n", size);
   return malloc(size);
}

void operator delete(void *p)
{
   printf("Delete\n");
   free(p);
}

void main(void)
{
   long *p = new long;
   delete p;
}
```

Классы

Язык Си++ становится объектно-ориентированным с введением в него нового, фундаментального понятия — понятия класса. В принципе, класс — это расширение структуры языка Си. В Си структура может включать в себя только элементы данных, например:

```
struct struc_int {
   int ih;
};
```

В языке Си++ в структуру дополнительно могут быть включены функции:

```
#include <stdio.h>

struct class_int {
   int ih;
   void set(int i) { ih = i; }
   int get(void) { return ih; }
};

void main(void)
{
   class_int a;
   a.set(2);
   printf("%d\n", a.get());
}
```

Здесь в структуру включены две **inline**-функции: **set()** для установки значения элемента данных **ih** и **get()** для чтения элемента данных. В основной функции демонстрируется использование элементов-функций для установки и чтения.

Элементы-функции

Разберем этот пример подробнее. Если бы внутри структуры **class_int** не было определений функций, то использовать её элементы данных можно было бы через операцию взятия элемента структуры «точка» или «стрелка»:

```
void main(void)
{
   struc_int a;
   a.ih = 2;                  /* установка */
   printf("%d\n", a.ih);      /* чтение */
}
```

Кажется, что с введением функций в структуру ничего не изменилось:

```
void main(void)
{
   class_int a;
   a.ih = 2;                  /* установка */
   printf("%d\n", a.ih);      /* чтение */
}
```

Пока это действительно так. Мы по-прежнему можем использовать структуру обычным образом, как это показано выше. Обратим внимание на следующие два момента использования функций, определенных внутри структуры:

- Вызов функции (обращение к ней) осуществляется через операцию взятия элемента структуры, точно так же, как и обращение к элементу данных:

```
a.ih = 2;       /* обращение к элементу данных */
a.set(2);       /* обращение к элементу-функции */
```

- В теле элемента-функции элемент данных класса виден непосредственно, а не через операцию взятия элемента структуры, или, говоря иначе, элемент-функция **прямо** «видит» элементы данных:

```
void set(int i) { ih = i; }
```

Элемент-функция, с одной стороны, имеет непосредственное отношение к структуре, к ее определению, так как является ее частью, а с другой стороны — к конкретному представителю этой структуры, так как может быть вызвана только через конкретный представитель:

```
a.set(2);   /* обращение к элементу-функции */
```

Элемент-функция одинаковым образом действует на разных конкретных представителей класса:

```
class_int a, b;
a.set(2);
b.set(3);
```

Элемент-функция — неизменяемая, постоянная часть класса. Она определяется один раз для всех представителей данного класса, и существует в единственном экземпляре в конечном коде программы. Возникает естественный вопрос, — каким образом эта единственная функция знает, с какой областью памяти следует выполнить требуемое преобразование, когда она вызывается для различных представителей класса? Прежде нам следует выяснить, в чем разница между классом и объектом.

Введение понятия класса более четко, более явно разграничивает понятия типа и представителя типа. Тип — это описание, представитель — это объект. Тип никаким образом не присутствует в конечном коде программы, он имеет смысл только на момент компиляции. Тип определяет структуру данных (состав и *относительное* расположение элементов данных) и допустимые операции над ними; структура данных определяется элементами данных, а допустимые операции — элементами-функциями. Представитель типа *реально использует память* для хранения элементов данных.

Класс — это описание структуры данных вместе с допустимыми над типом операциями, это тип.

Объект — это представитель класса, физически представляющий собой **область памяти**, отведенную под элементы данных.

Рассмотрим пример, не использующий классы:

```
void seti(int *i, newvalue)
{
   *i = newvalue;
}

void main(void)
{
   int a, b;
   seti(&a, 1);
   seti(&b, 2);
}
```

Здесь также используется два представителя типа **int** и одна функция **seti** для установки значения. Для того чтобы функция для установки значения «знала», какой элемент данных изменить, мы применили искусственный прием — передали в функцию *указатель* на этот элемент. Это естественно — как еще мы можем указать на конкретного представителя? То же самое происходит и при использовании класса:

```
void main(void)
{
   class_int a, b;
   a.set(1);
   b.set(2);
}
```

Здесь при вызове функции **set** ей тоже передается указатель. При вызове функции с объектом **a** функции передается указатель на объект **a**, а при вызове функции с объектом **b** ей передается указатель на этот объект. Происходит это *неявно* (невидимо для программиста). Фактически указатель на объект всегда передается элементу-функции класса *первым параметром*. Если у функции **set** определен один параметр, то фактически (в конечном коде программы) ей передается два параметра. Класс, таким образом, скрывает от программиста детали реализации, предоставляя ему удобство работы с объектами. Благодаря этому неявному указателю на объект элемент-функция и «видит» его.

Обычно программисту не требуется неявный параметр элемента-функции. Но при необходимости он может использовать его через ключевое слово **this**. Это слово всегда указывает на объект, с которым элемент-функция была вызвана. В следующем примере показано, как применяется **this**:

```
   void set(int i) { this->ih = i; }
```

Элементы-функции могут быть вызваны только с представителями класса, в котором они описаны. Откуда компилятор «знает», к какому классу относится функция? Рассмотрим сначала, как можно использовать функцию **seti** для разных типов:

```
void seti(int *, int);
void main(void)
{
   int a;
   short b;
   seti(&a, 1);
   seti(&b, 2);
}
void seti(int *i, newvalue)
{
   *i = newvalue;
}
```

Здесь мы видим, что функция **seti** с одинаковым успехом может быть применена к объектам разных типов. Попробуем теперь точно так же использовать функцию класса:

```
struct integer {
   int ih;
   void set(int i) { ih = i; }
};
void main(void)
```

```
{
   integer a;
   short b;
   a.set(1);
   b.set(2);  /* ошибка */
}
```

В результате мы получим ошибку времени компиляции, так как функция-элемент может быть применена только к представителю класса. Нет никакой возможности заставить функцию работать с представителем другого описания, сколь бы похожими они не были. Достигается это за счет того, что каждая элемент-функция имеет *декорированное имя*, состоящее из имени класса и имени функции. Например, если мы определим два похожих класса следующим образом:

```
struct inta {
   int ih;
   void set(int i) { ih = i; }
};
struct intb {
   int ih;
   void set(int i) { ih = i; }
};
void main(void)
{
   inta a;
   intb b;
   a.set(1);
   b.set(2);
}
```

то при компиляции фактическое (декорированное) имя функции **`set`** класса **`inta`** будет **`@inta@set$qi`**, а фактическое имя функции **`set`** класса **`intb`** будет **`@intb@set$qi`** (декорированные имена — особенность конкретного компилятора). Именно поэтому всегда вызывается правильная функция — та, которая относится к данному классу. Таким образом, несмотря на одинаковые имена элементов-функций, которые определил программист (в примере **`set`**), фактические имена функций **всегда** различаются. Никакого волшебства, все в точном соответствии с принципами обычного, процедурного программирования. Компилятор объектно-ориентированного языка просто помогает нам, скрывая от нас детали.

Дополнительные сведения об элементах класса

К элементам данных класса применяются правила:

- элементами данных могут быть (кроме встроенных в язык типов) перечислимые типы, битовые поля и представители другого предварительно объявленного класса;
- представитель данного класса *не может быть* элементом данных;
- элементом данных *может быть* указатель или ссылка на тип данного класса;
- элементы данных не могут быть объявлены с ключевыми словами **`auto`**, **`extern`** или **`register`**.

Функции-элементы могут также называться **методами**. Английский термин для элемента данных — **data member**, для функции-элемента — **member function**.

Инкапсуляция и целостность типа

Инкапсуляция является первым из важнейших понятий объектно-ориентированного программирования. Класс — это описание типа и операций, определенных над этим типом. Инкапсуляция обеспечивает *целостность типа* за счет того, что доступ к элементам данных класса ограничивается при помощи ключевых слов **`private`** и **`protected`**. Элементы данных как бы заключаются в «капсулу», которая *охраняет* данные от непредусмотренного изменения. Вместо непосредственного изменения элемента данных предполагается, что данные могут быть изменены только через элементы-функции. Рассмотрим класс, описывающий натуральные числа:

```
struct natural {
   int p;
   int get(void) { return p; }
   void set(int i) { if (i > 0) p = i; }
};
```

Здесь функция *set* проверяет, является ли новое значение натуральным числом, предохраняя данные от установки в недопустимые значения. Однако целостность типа может быть нарушена прямым изменением элемента данных, как это показано в следующем примере его использования:

```
void main(void)
{
   natural n;
   n.p = 0;
}
```

Поскольку полагаться на то, что программист всегда будет соблюдать правила типа, нельзя, объектно-ориентированный подход предусматривает защиту элементов данных:

```
struct natural {
private:
   int p;
public:
   int get(void) { return p; }
   void set(int i) { if (i > 0) p = i; }
};
```

Ключевое слово **`private`** запрещает компилировать код, напрямую обращающийся к элементу данных. Происходит *инкапсуляция* элемента данных **`p`**, его **сокрытие**. С другой стороны, элемент-функция для установки значения данных должна быть доступна, поэтому она объявлена после ключевого слова **`public`**. Элемент-функция *get* нужна для того, чтобы иметь возможность прочитать значение элемента данных. Никаких проверок и ограничений она обычно не выполняет. Мы просто вынуждены использовать ее для чтения элемента данных после его инкапсуляции.

Язык Си++ вводит также новое ключевое слово **`class`** для описания типа. Различие между структурой **`struct`** и структурой **`class`** заключается в том, что по умолчанию (без применения ключевых слов **`public`** и **`private`**) тип доступа к

элементам класса в структуре **struct** — это **public**, а в структуре **class** — это **private**. Поэтому приведенное выше описание класса **natural** можно следующим образом записать с использованием слова **class**:

```
class natural {
   int p;
public:
   int get(void) { return p; }
   void set(int i) { if (i > 0) p = i; }
};
```

Оба описания равнозначны. Выбор ключевого слова **struct** или **class** не имеет принципиального значения. Значение имеет только факт наличия в описании инкапсуляции и элементов-функций для обеспечения целостности. В случае отсутствия инкапсуляции и элементов-функций мы имеем дело со структурой данных, с так называемым *пользовательским типом данных*. Иначе мы имеем новый, защищенный, интегрированный в язык тип данных. Однако предпочтительнее использовать **struct** для описания структур, в которых инкапсуляция данных не требуется.

Конструктор класса

Рассмотрим определение переменной класса *natural*:

```
void main(void)
{
   natural n;    /* n = 0 */
}
```

В классе *natural* минимальное значение элемента данных должно быть 1, в то время как сразу после создания объекта класса *natural* значение элемента данных **p** будет равно 0, что является недопустимым. Так мы приходим к заключению, что требуется механизм для инициализации начального состояния представителя класса.

Рассмотрим теперь пример определения переменной обычного типа:

```
void main(void)
{
   int i = 1;
}
```

Попробуем выполнить аналогичное определение для класса *natural*:

```
void main(void)
{
   natural n = 1;
}
```

Результатом компиляции этого примера будет ошибка (невозможно привести тип *int* к типу *natural*). Следовательно, тип, определенный нами, отличается от типа, встроенного в язык, такого, как *int*. Класс не полностью интегрирован в язык. Для полной интеграции нового типа в язык программирования в классе должны быть определены специальные функции.

Одной из таких функций является *конструктор класса*. Назначение конструктора — сформировать представителя класса, то есть определить все элементы данных класса перед началом использования. Конструктор класса вызывается ком

пилятором автоматически всегда, когда требуется создать нового представителя класса. В языке Си++ класс может иметь несколько конструкторов, а также несколько разных типов конструкторов.

Во время создания объекта класса *natural* следует установить элемент данных в правильное значение. Выполнить это можно при помощи конструктора без параметров, иначе называемым конструктором по умолчанию (*default constructor*). Конструктор по умолчанию генерируется компилятором автоматически, если программист забыл определить его. При этом объект конструируется присвоением элементам данных нулевых начальных значений. Если это не нарушает целостности класса, конструктор по умолчанию можно не объявлять.

Конструктор, являясь функцией класса, должен иметь имя. Имя конструктора в языке Си++ совпадает с именем класса, в нашем случае с **`natural`**:

```
struct natural {
private:
   int p;
public:
   natural(void) { p = 1; }
   int get(void) { return p; }
   void set(int i) { if (i > 0) p = i; }
};

void main(void)
{
   natural n;   /* n = 1 */
}
```

Вследствие возможности перегрузки функций в языке Си++ класс может иметь несколько конструкторов. Поэтому мы можем определить дополнительные конструкторы для нашего класса, изменяя параметры. Например, следующий конструктор позволяет установить начальное значение объекта, используя целое число:

```
struct natural {
private:
   int p;
public:
   natural(void) { p = 1; }
   natural(int i) { if (i > 0) p = i; }
};
void main(void)
{
   natural n = 2;   /* n = 2 */
}
```

Этот новый конструктор осуществляет приведение типа **`int`** к типу **`natural`**, поэтому конструктор подобного вида (с одним параметром) называется **конструктором приведения** или **конструктором преобразования**. Наличие конструктора приведения сообщает компилятору о возможности приведения, само же приведение типа должен выполнить программист. В нашем простейшем случае это происходит автоматически за счет того, что тип элемента данных совпадает с типом аргумента конструктора или может быть приведен автоматически компилятором. Следующий пример показывает автоматическое приведение внутри конструктора:

```
void main(void)
{
   long a = 2;
   natural n = a;    /* n = 2 */
}
```

Несмотря на то, что используемый конструктор предназначен для приведения типа **int**, за счет автоматического преобразования во время передачи параметра конструктору мы можем выполнить присвоение начального значения от переменной типа **long**.

Упражнение: определите конструктор приведения типа **long** к типу **natural**.

В целом к конструктору применяются следующие правила:

- конструктор не имеет возвращаемого типа и не может возвращать значение;
- конструктор может иметь аргументы по умолчанию;
- конструктор не наследуется;
- конструктор не может быть объявлен с ключевыми словами **const**, **volatile**, **virtual** или **static**;
- конструктор может быть объявлен в части **public**, **protected** или **private**. Объекты класса, конструктор которого имеет защищенный доступ, могут быть созданы только из дружественных функций и из производных классов. Объекты класса, конструктор которого имеет закрытый доступ, могут быть созданы только из дружественных функций.

Деструктор

Рассмотрим более сложный класс, который в качестве элемента данных использует указатель. Известно, что язык Си не имеет встроенного типа «строка». Однако мы можем, используя объектно-ориентированные свойства языка, определить новый тип:

```
#include <string.h>

class string {
   char *p;
public:
   string(char *s) { p = new char[1 + strlen(s)]; strcpy(p, s); }
};
void f(void)
{
   string a = "abc";
}

void main(void)
{
   f();
}
```

Конструктор приведения этого класса использует операцию **new** для выделения области памяти для размещения строки и размещает строку при помощи функции **strcpy()**.

При завершении области действия объекта **a**, однако, оказывается, что область памяти, выделенная под строку при выполнении функции **f()**, более недоступна

для повторного использования, что может привести к неприятным последствиям. При разрушении объекта память следует освобождать. Для этой цели в классе может быть определен *деструктор*. Его единственное назначение — освободить память, которую объект использовал для размещения своих данных, поэтому деструктор определяется для классов, использующих указатели в качестве элементов данных. Деструктор вызывается компилятором автоматически при выходе объекта из области действия (при разрушении объекта).

В качестве имени деструктора в языке Си++ используется имя класса с предшествующим знаком тильда (знак тильда означает «дополнение» — дополнение до конструктора). Ниже приведено описание класса **string** с деструктором:

```
class string {
   char *p;
public:
   string(char *s) { p = new char[1 + strlen(s)]; strcpy(p, s); }
   ~string(void) { delete []p; }   /* деструктор */
};
```

Квадратные скобки в операции **delete** используются только при разрушении ссылок на массивы.

К деструктору применяются следующие правила:

- деструктор не может иметь возвращаемого типа и не возвращает значения;
- деструктор не может иметь аргументов;
- деструктор наследуется;
- деструктор не может быть объявлен с ключевыми словами **const**, **volatile** или **static**;
- деструктор **может быть** объявлен с ключевым словом **virtual**.

Компилятор генерирует деструктор по умолчанию, если программист не определил его.

Список инициализации

Инициализация элементов данных класса обычно осуществляется в теле конструктора. Есть и другой способ, называемый *списком инициализации*.

Список инициализации располагается непосредственно после списка аргументов после двоеточия. Инициализируемые элементы, если их несколько, разделяются запятыми. Инициализируемое значение заключается в скобки. Если инициализируемых значений несколько, они разделяются запятыми. Точка с запятой в списке инициализации недопустима.

```
struct natural {
private:
   int p;
public:
   natural(void) : p(1) {}
};
```

Список инициализации является единственным способом инициализации элементов-констант, ссылок и объектов, конструктор которых требует параметры (см. наследование). Если список инициализации полностью определяет все эле

менты данных, требующие инициализации, то тело конструктора может быть пустым, как в приведенном выше примере. Инициализация элементов класса может осуществляться и комбинированным способом — часть элементов в теле конструктора, а часть — через список инициализации.

Область видимости класса

До сих пор тела элементов-функций мы определяли непосредственно в описании класса. Это возможно только в случае простых функций. Сами функции при этом являются подставляемыми (**inline**). Любая функция-элемент может быть определена вне описания с использованием операции разрешения видимости **::**, которая вводит новое понятие — область видимости класса. В следующем примере функции класса определены вне описания:

```
struct natural {
private:
   int p;
public:
   natural(void);
   natural(int);
   int get(void);
   void set(int);
};

inline natural::natural(void) : p(1) {}

natural::natural(int i)
{
   if (i > 0) p = i;
}

int natural::get(void)
{
   return p;
}

void natural::set(int i)
{
   if (i > 0) p = i;
}

void main(void)
{
   natural n;
}
```

В целях демонстрации инициализация в конструкторе по умолчанию выполнена через список инициализации, и он объявлен как подставляемый. В приведенном примере видно, как следует указывать возвращаемое значение функции, если оно есть и как следует использовать операцию разрешения видимости. Под областью видимости класса понимается видимость всех элементов класса за операцией разрешения видимости, независимо от типа доступа к элементу — **private**, **protected** или **public**.

Особенность функций-элементов, определенных вне описания, заключается в том, что при трассировке программы можно наблюдать, когда какие функции

класса используются. Это очень полезно при изучении специальных функций класса, когда вызов той или иной функции не является очевидным.

Конструктор копии

Рассмотрим класс, использующий указатель в качестве элемента данных:

```
class string {
   char *p;
public:
   string(void) { p = new char[65]; p[0] = 0; }
};

void main(void)
{
   string a;
   string b = a;
}
```

В основной функции используется копирование одного объекта в другой. Класс не определяет, как получить копию объекта, поэтому компилятор выполняет *побитовое копирование*. В результате указатель объекта `b` получит значение указателя объекта `a`, а область памяти, выделенная для объекта `b`, теряется. Оба объекта указывают на одно и то же значение. Побитовое копирование не может быть использовано для объектов класса, содержащего ссылки или указатели.

Конструктор копии предназначен для описания процедуры получения копии объекта и имеет вид

```
имя_класса(имя_класса&) или имя_класса(const имя_класса&),
```

то есть в качестве аргумента конструктор принимает ссылку или константную ссылку на объект класса. Эта ссылка используется для извлечения элементов данных из копируемого объекта и присвоения их элементам данных новой (генерируемой) копии. Следующий пример поясняет, как это сделать для класса `string`.

```
#include <string.h>

class string {
   char *p;
public:
   string(void) { p = new char[65]; p[0] = 0; }
   string(string& s);
};

string::string(string& s)
{
   strcpy(p, s.p);
}

void main(void)
{
   string a;
   string b = a;
}
```

Операция присваивания

Рассмотрим пример, в котором используется операция присваивания:

```
void main(void)
{
   string a;
   string b = a;
   a = b;
}
```

Точно так же, как и в случае с получением копии, здесь производится побитовое копирование объекта **b** в объект **a**. Следовательно, дополнительно к конструктору копии мы должны описать процедуру *присваивания* одного объекта другому. Это осуществляется при помощи функции **operator=**. В следующем примере эта функция описана и вызывается в операции присваивания:

```
#include <string.h>
class string {
   char *p;
public:
   string(void) { p = new char[65]; p[0] = 0; }
   string(string& s);
   string operator= (string&);
};

string::string(string& s)
{
   strcpy(p, s.p);
}

string string::operator = (string& s)
{
   strcpy(p, s.p);
   return *this;
}

void main(void)
{
   string a;
   string b = a;
   a = b;
}
```

Примечание: для того, чтобы выполнять последовательные присваивания вида **a = b = c**, операция присваивания должна возвращать ссылку (**string&**):

Константная ссылка в качестве аргумента

Операция присваивания не может быть выполнена для случая, если в правой части операции находится константа. В этом случае в качестве аргумента операции присваивания следует использовать константную ссылку:

```
#include <string.h>

struct string {
   char *p;
public:
   string(void) { p = new char[65]; p[0] = 0; }
   string(const char *s) { strcpy(p, s); }
   string operator= (const string&);
};
```

```
string string::operator = (const string& s)
{
   strcpy(p, s.p);
```

```
   return *this;
}

void main(void)
{
   const char *s = "abc";
   string a;
   a = s;
}
```

Конструктор копии или операция присваивания?

Конструктор копии и операция присваивания выполняют копирование элементов данных объекта. Определить, какая функция будет вызвана компилятором в конкретном случае, можно с помощью правил:

- вызывается операция присваивания, если изменяется существующий объект;
- вызывается конструктор копии, если новый объект инициализируется значениями другого объекта.

Например, конструктор копии вызывается при передаче аргумента в функцию и при возврате объекта из функции в операции присваивания (если возвращаемый тип функции имеет тип класса). В операции присваивания может вызываться сначала конструктор копии, а затем операция присваивания класса.

Проверка на само-присвоение

Для исключения присваивания объекта самому себе в операции присваивания следует проверять аргумент функции `operator=`, сравнивая его с указателем `this`:

```
string string::operator = (const string& s)
{
   if (&s = this) error . . .
   strcpy(p, s.p);
   return *this;
}
```

Для того чтобы предотвратить само-присвоение, следует использовать исключения.

Запрещение копирования и присваивания

Если класс содержит указатели или ссылки в качестве элементов данных, он должен определять операцию присваивания и конструктор копии. Другое решение заключается в том, чтобы запретить эти операции, объявив их в закрытой части описания. В этом случае попытка выполнить присваивание или копирование объекта вызовет ошибку времени компиляции. Однако эти операции будут доступны внутри класса и из дружественных функций:

```
struct string {
   char *p;
   string operator= (const string&);
public:
   string(void) { p = new char[65]; p[0] = 0; }
   string(const char *s) { strcpy(p, s); }
};
```

Операции приведения

Операции приведения выполняют преобразование типа класса к другому типу. Например, для класса **string** невозможно выполнить присваивание переменной типа указатель на знак:

```
#include <string.h>

struct string {
   char *p;
public:
   string(void) { p = new char[65]; p[0] = 0; }
   string(char *s) { strcpy(p, s); }
};

void main(void)
{
   string a = "abc";
   char *s = a;       /* ошибка */
}
```

Чтобы такое присваивание стало возможным, следует определить операцию приведения к типу **char ***.

Операция приведения имеет вид **operator имя_нового_типа(void)** и к ней применяются следующие правила:

- операция приведения не имеет аргументов;
- операция приведения не имеет явной спецификации типа возвращаемого значения (возвращаемый тип указывается после слова **operator**);
- операция приведения может быть описана как **virtual**;
- операция приведения наследуется.

Следующий пример показывает, как определить операцию приведения **string** к **char***.

```
#include <string.h>

struct string {
   char *p;
public:
   string(void) { p = new char[65]; p[0] = 0; }
   string(char *s) { strcpy(p, s); }
   operator char *(void);
};

string::operator char * (void)
{
   char *a = new char[65];
   strcpy(a, p);
   return a;
}

void main(void)
{
   string a = "abc";
   char *s = a;       /* все нормально */
}
```

В функции приведения мы получили копию строки для того, чтобы защитить данные класса. Если функция приведения просто возвращает указатель **p**, приведение также выполняется, но данные могут быть изменены:

```
#include <string.h>
struct string {
   char *p;
public:
   string(void) { p = new char[65]; p[0] = 0; }
   string(char *s) { strcpy(p, s); }
   operator char *(void) { return p; }
};

void main(void)
{
   string a = "abc";
   char *s = a;
   s = new char[1];  /* объект изменен */
}
```

Упражнение: определите операцию приведения **natural** к **int**.

Перегрузка операций

Язык Си++ разрешает использовать знаки операций в качестве функций класса. Это дает возможность использовать знаки операций по их прямому назначению и позволяет получить тип, поведение которого полностью соответствует семантике встроенного в язык типа, такого, например, как **int**.

Следующий пример показывает, что можно получить, используя перегрузку операций. Класс **natural**, ограничивая множество допустимых значений, пока не разрешает использовать объекты этого класса в таких простых операциях, как сложение и вычитание:

```
class natural {
   int p;
public:
   natural(void) { p = 1; }
   natural(int i) { if (i > 0) p = i; }
};

void main(void)
{
   natural a, b = 2, c;
   a = a + b;
}
```

Определим операцию «сложение» (выполним перегрузку этой операции):

```
class natural {
   int p;
public:
   natural(void) { p = 1; }
   natural(int i) { p = i; }
   natural operator +(natural&);
};

natural natural::operator +(natural& a)
{
   return p + a.p;
}
```

```
void main(void)
{
   natural a = 5, b = 3, c;
   c = a + b;
}
```

Операция сложения двухместная, то есть требует два операнда. Левый операнд, **a**, в функции сложения доступен прямо, через неявный указатель **this**. Правый операнд, **b**, в функции сложения доступен через аргумент функции. Возвращаемое значение имеет тип класса. Чтобы лучше понять соотношение между операндами, перегрузим операцию вычитания, в которой порядок операндов имеет значение:

```
class natural {
   int p;
public:
   natural(void) { p = 1; }
   natural(int i) { p = i; }
   natural operator -(natural&);
};
natural natural::operator -(natural& a)
{
   natural b;
   int i = p - a.p;
   if (i > 0) b.p = i; else b.p = 1;
   return b;
}
void main(void)
{
   natural a = 5, b = 3, c;
   c = a - b;
}
```

В случае одноместной операции функция перегрузки не имеет параметров. В следующем примере перегружается операция «унарный минус», смысл которой — получение отрицательного числа. Так как отрицательные числа не входят во множество натуральных, мы не можем перегрузить эту операцию для класса **natural**. Вместо этого в примере используется класс рациональных чисел, представляемых в виде дроби, числитель и знаменатель которой — целые. Элемент данных **num** — это числитель (numerator), а элемент данных **den** — знаменатель (denominator):

```
class rational {
   int num, den;
public:
   rational(void) : num(0), den(1) {}
   rational(int n, int d = 1) { if (d != 0) { num = n; den = d; } }
   rational operator -(void) { return rational(-num, den);  }
};
void main(void)
{
   rational a(2), b;
   b = -a;
}
```

Здесь используются дополнительные приемы: значение аргумента по умолчанию, явный вызов конструктора для генерирования нового анонимного объекта.

В качестве примера определим также функцию сравнения двух объектов:

```
class natural {
   int p;
public:
   natural(void) { p = 1; }
   int operator ==(natural&);
};
int natural::operator ==(natural& a)
{
   return p = a.p;
}
void main(void)
{
   natural a, b;
   int equal = a == b;
}
```

Приведенных примеров достаточно для того, чтобы понять, как выполняется перегрузка основных операций. Дополнительно см. разделы «Перегрузка ++», «Операция индексирования», «Операция функция», «Операции класса new и delete» и «Перегрузка операций: функция-элемент vs функции-друга».

Правила перегрузки

- нельзя перегрузить следующие 4 операции: `.` `.*` `::` `?:` ;
- приоритеты перегруженных операций — принятые для встроенных типов данных;
- функция-операция не может иметь аргументов по умолчанию;
- за исключением `operator=`, функции-операции наследуются;
- нельзя перегрузить операции для встроенных типов;
- семантика перегружаемой операции должна соответствовать общепринятой. Например, не следует выполнять вычитание при помощи операции «плюс». Это требует серьезной проработки для получения законченного класса.

Перегрузка ++

Операция `++` (и `--`) имеет две разновидности: префиксную и постфиксную. Для получения префиксной формы оператора используется функция-элемент без аргументов. Для получения постфиксной формы используется функция-элемент с одним аргументом, который не используется и фактически равен нулю:

```
#include <stdio.h>
class natural {
   int p;
public:
   natural(void) { p = 1; }
   natural(int i) { if (i > 0) p = i; }
   natural operator ++(void);   /* префиксная форма */
   natural operator ++(int);    /* постфиксная форма */
};
natural natural::operator ++(void)
{
   return ++p;
}
natural natural::operator ++(int)
{
   return p++;
}
```

```
void main(void)
{
   natural a;
   printf("\n%d", a++);    /* выводит 1 */
   printf("\n%d", ++a);    /* выводит 3 */
}
```

Операция индексирования

В некоторых случаях требуется получить доступ к отдельному элементу данных класса по его индексу или номеру. Для этого нужно перегрузить операцию индексирования `[]`. Например, для класса `string` можно определить индексирование элементов строки следующим образом:

```
#include <string.h>

struct string {
   char *p;
public:
   string(void) { p = new char[65]; p[0] = 0; }
   string(char *s) { strcpy(p, s); }
   char& operator [](int);
};

char& string::operator [](int n)
{
   return p[n];
}

void main(void)
{
   string a = "abc";
   char c = a[1];
}
```

Если возвращаемое значение операции индексирования — ссылка, как в примере, ее можно использовать в левой части операции присваивания:

```
void main(void) {
   string a = "abc";
   a[2] = 'b';
}
```

Операция функция

Функция-элемент `operator()` определяет вызов функции через переменную типа класса. Эта функция используется для типов с единственной операцией (вызов функции) и часто применяется для построения итераторов — типов, возвращающих упорядоченные (последовательные) значения из некоторого набора. Набор при этом является другим типом (классом). В качестве примера рассмотрим класс `sarray`, предназначенный для работы с массивом строк фиксированной длины, и дружественный ему класс `saitem`, назначение которого — предоставлять последовательный доступ к элементам массива строк класса `sarray`. Класс `sarray` имеет три закрытых элемента данных — указатель на массив символов `array`, счетчик строк `count` и размер строки `isize`, указываемые при инициализации объекта:

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

```
class sarray {
   int isize, count;
   char *array;
public:
   sarray(int, int);
   friend class saitem;
};
```

Конструктор этого класса выделяет память для указанного количества строк указанной длины:

```
sarray::sarray(int n, int size) : count(n), isize(size + 1)
{
   array = new char[count * isize];
}
```

Класс итератора содержит закрытые элементы данных для хранения указателя на объект итерации и номер очередного элемента массива этого объекта. Эти элементы инициализируются в конструкторе. Единственной функцией-элементом этого класса (кроме конструктора) является оператор функция:

```
class saitem {
   int n;
   sarray *sa;
public:
   saitem(sarray& a) : n(0), sa(&a) {}
   char* operator ()(void);
};
```

Реализация этого оператора проверяет текущее значение номера элемента, вычисляет и возвращает его адрес. По достижении последнего элемента номер очередного элемента обнуляется для инициирования нового цикла обращений:

```
char* saitem::operator ()(void)
{
   if (n >= sa->count) n = 0;
   return sa->array + (n++ * sa->isize);
}
```

В основной функции инициализируется новый массив из трех строк размером 20 символов и итератор **next** для этого массива. Массив заполняется значениями последовательными вызовами итератора и затем выводится на дисплей такими же последовательными обращениями:

```
void main(void)
{
   sarray b(3, 20);
   saitem next(b);
   strcpy(next(), "String 1");
   strcpy(next(), "String 2");
   strcpy(next(), "String 3");
   for (int i = 0; i < 3; i++) printf("%s\n", next());
}
```

Друзья

Доступ к элементам класса извне класса возможен только для элементов, которые описаны в открытой части, например, после ключевого слова **public**. Если требуется получить доступ к элементам закрытой или защищенной частей класса, следует использовать **друзей**, то есть функции и классы, которым разрешается доступ к любым элементам класса при помощи ключевого слова **friend**.

Дружественные классы

В качестве друга класса может выступать класс целиком. Для объявления класса `B` другом класса `A` в описание класса `A` следует включить определение класса `B` как друга:

```
class A {
   friend class B;
};
```

Дружественные функции

Особый доступ к элементам класса может быть разрешен для отдельной независимой функции или для отдельной функции-элемента другого класса, например:

```
void f(void)
{
}
class B {
public:
   void f(void);
};
class A {
   friend void f(void);
   friend void B::f(void);
};
```

Правила относительно друзей

- на описания `friend` не влияют спецификаторы `public`, `protected` и `private`;
- описания `friend` не взаимны: если `A` объявляет другом `B`, это не значит, что `A` является другом для `B`;
- дружественность не наследуется: если `A` объявляет `B` другом, классы, производные от `B`, не становятся автоматически друзьями `A`;
- дружественность не транзитивна: если `A` объявляет `B` другом, классы, производные от `A`, не будут автоматически признавать дружественность `B`.

Перегрузка операций: функция-элемент vs функции-друга

Дружественные функции могут быть использованы для перегрузки операций. В этом случае функция будет иметь на один аргумент больше, причем первый аргумент будет соответствовать указателю `this`.

В качестве примера приведена дружественная функция-оператор для класса `natural`:

```
class natural {
   int p;
public:
   natural(void) { p = 1; }
   natural(int i) { if (i > 0) p = i; }
   friend natural operator +(natural, natural);
   natural operator -(natural&);
};

natural operator +(natural x, natural y)
{
   return natural(x.p + y.p);
}
```

```
natural natural::operator -(natural& y)
{
   return natural(p + y.p);
}

void main(void)
{
   natural a = 5, b = 3, c;
   c = a + b;
   c = a - b;
}
```

В каких случаях следует использовать функцию-элемент, а в каких — функцию-друга? Функция-элемент может быть вызвана только для «настоящего» объекта, в то время функция-друг может быть вызвана для объекта, созданного с помощью неявного преобразования типа. Для класса, описанного в приведенном выше примере, операция

```
   c = 9 + b;
```

допустима, а операция

```
   c = 9 - b;
```

нет. Правила для выбора функции-элемента или дружественной функции следующие:

- Если операция изменяет состояние объекта, она должна быть элементом класса. Это справедливо для операций типа `=`, `*=`, `++` и т.п.
- Если требуется неявное преобразование для всех операндов операции, функция должна быть другом. Это справедливо для операций типа `+`, `-`, `||` и т.п.
- Если для класса не определены никакие преобразования типа, выбор элемента или друга безразличен.

В случае сомнений следует использовать функции-элементы.

Статические элементы класса

Объекты класса представляют собой собственные, отдельные копии элементов класса. Однако некоторые типы реализуются наиболее элегантно и просто, если все объекты этого типа могут совместно использовать те или иные данные. Для обеспечения целостности класса предпочтительно, чтобы эти данные являлись частью класса. Объявление элемента данных с ключевым словом `static` указывает, что данный элемент будет единственным для всех объектов и его следует рассматривать как глобальный в пределах области видимости данного класса. Правила:

- память под статический элемент класса выделяется даже при отсутствии представителей класса;
- класс со статическим элементом данных должен не только объявлять, но и определять его.

```
class natural {
   static int counter;       /* описание статического элемента */
   int p;
public:
   natural(void) { p = counter++; }
};
```

```
int natural::counter = 1;   /* определение статического элемента */
void main(void)
{
   natural a, b, c;
}
```

В этом примере статический элемент данных **counter** является счетчиком представителей класса, сгенерированных при помощи конструктора по умолчанию. Дополнительно каждый новый объект является последующим натуральным числом.

Если статический элемент имеет тип доступа **public**, то доступ к нему извне класса возможен через операцию разрешения видимости или через любой представитель класса. Например:

```
class natural {
   int p;
public:
   static int counter;
   natural(void) { p = counter++; }
};

int natural::counter = 1;   /* определение статического элемента */

void main(void)
{
   natural a, b, c;
   natural::counter = 1;
   a.counter = 1;
   b.counter = 1;
}
```

Все три способа равнозначны

Статические функции-элементы

Функция-элемент, объявленная как статическая, не ассоциируется с отдельными представителями класса и при вызове ей не передается указатель **this**. К ней применяются правила:

- статическая функция-элемент может вызываться независимо от того, существует или нет хотя бы один представитель класса;
- статическая функция-элемент может обращаться только к статическим элементам данных и к другим статическим функциям-элементам класса;
- доступ к статической функции-элементу осуществляется так же, как к статическому элементу данных;
- статическая функция-элемент *не может* быть объявлена с ключевым словом **virtual**.

В качестве примере приведена статическая функция-элемент, возвращающая значение счетчика объектов класса **natural**:

```
class natural {
   static int counter;
   int p;
public:
   natural(void) { p = counter++; }
   static int Counter() { return counter; }
};
```

```
int natural::counter = 1;
void main(void)
{
   natural a, b, c;
   int i = natural::Counter();
}
```

Константные объекты и константные функции-элементы

Если представитель класса создан с ключевым словом **const**, то его нельзя изменять после инициализации. Если такой объект используется с не константной функцией-элементом, компилятор генерирует ошибку. Константная функция-элемент:

- объявляется с ключевым словом **const**, которое следует за списком параметров;
- *не может* изменять элементы данных класса;
- *не может* обращаться к не константным функциям-элементам класса;
- *может* вызываться как для константных, так и для не константных представителей класса.

В следующем примере используется константная функция-элемент для чтения элементов данных:

```
class coord {
   int xh, yh;
public:
   coord(int x, int y) : xh(x), yh(y) {}
   void set(int x, int y) { xh = x; yh = y; }
   void get(int& x, int& y) const;
};

void coord::get(int& x, int& y) const { x = xh; y = yh; }

void main(void)
{
   coord p1(1, 2);
   const coord p2(2, 2);
   int x, y;
   p1.get(x, y);
   p2.get(x, y);
   p2.set(3, 3);   /* ошибка */
}
```

Операции класса new и delete

Класс может определять свои варианты функций **new** и **delete** (перегружать их). Они вызываются всякий раз, когда создается или уничтожается динамический объект. Если предполагается создание массива динамических объектов, то эти операции определяются в модификации **new[]** и **delete[]**. Класс может определять обе модификации **new** и **delete**. К этим функциям применяются следующие правила:

- функция **new** должна иметь тип возвращаемого значения **void***;
- функция **delete** должна иметь тип возвращаемого значения **void**;

- требуют параметр типа **size_t**, определенный в **<stddef.h>**.

В качестве примера рассмотрим класс «строка». Каждый раз при создании динамического объекта типа «строка» требуется выделить память под объект, а при уничтожении объекта освободить ее. Мы можем уменьшить затраты времени на эти процессы, если возьмем управление памятью на себя. Если предположить, что при выполнении программы нам потребуется только ограниченное количество динамических строк, мы можем выделить память под них один раз и по мере необходимости использовать ту или иную область этой памяти. Для управления нам понадобятся дополнительные элементы данных. Элемент **bus** показывает, что данный объект используется. Вместе с элементом **buf** он составляет «тело» объекта «строка». Статический элемент **sa** указывает на область памяти, в которой будут расположены динамические строки. Статический элемент **count** предназначен для подсчета количества используемых динамических строк, а статический элемент **maxcount** определяет максимальное количество динамических строк, которое можно использовать в программе. Длина строки (максимальная) определяется константой **SLEN**. Константы **EMPTY** и **BUSY** представляют собой значения, указывающие на статус конкретной динамической строки:

```
#include <stdio.h>
#include <iostream.h>
#include <string.h>
const SLEN = 20;
const signed char EMPTY = 0, BUSY = 1;

class string {
   char bus;
   char buf[SLEN + 1];
   static char *sa;
   static int count, maxcount;
public:
   string(void)     { bus = BUSY; buf[0] = 0; }
   string(char *s) { bus = BUSY; strcpy(buf, s); }
   operator char* (void) { return buf; }
   void* operator new(size_t);
   void operator delete(void *, size_t);
};
```

После описания класса статические элементы данных инициализируются. В примере максимальное количество строк определено равным двум:

```
char* string::sa = 0;
int string::count = 0;
int string::maxcount = 2;
```

При создании нового объекта при помощи генератора **new** вызывается функция-элемент **new**, определенная для нашего класса. Если эта функция вызывается первый раз, указатель **sa** равен нулю и мы выделяем память для заданного количества объектов. Размер объекта определяется как длина строки **SLEN** плюс байт для завершающего строку символа «ноль» и плюс байт для признак использования строки. Первый байт каждой области, выделенной под объект, помечается признаком **EMPTY** (в цикле *for*):

```
void* string::operator new(size_t)
{
   if (!sa) {
      sa = new char[maxcount * (SLEN + 2)];
```

```
      for (int i = 0; i < maxcount; i++) {
         *(sa + (SLEN + 2) * i) = EMPTY;
      }
   }
```

Далее в функции **new** проверяются первые байты с целью найти свободную область под объект. Если такая область найдена, увеличивается значение количества использованных строк **count** и возвращается указатель на область. Если свободной области нет, возвращается указатель на первую область.

```
   char *p;
   for (int i = 0; i < maxcount; i++) {
      p = (sa + (SLEN + 2) * i);
      if (*p == EMPTY) {
         count++;
         return p;
      }
   }
   return sa;
}
```

Для правильного уничтожения объектов в классе предусмотрена функция-элемент **delete**. Эта функция принимает в качестве первого аргумента указатель на удаляемый объект. Используя этот указатель, мы очищаем признак использования области памяти и саму строку, уменьшаем счетчик использованных строк **count** и если он стал равен нулю, освобождаем область памяти, выделенную под динамические строки. Таким образом, освобождение памяти происходит только после удаления всех строк:

```
void string::operator delete(void *s, size_t)
{
   ((string*)s)->bus = EMPTY;
   ((string*)s)->buf[0] = 0;
   if (--count == 0) delete []sa;
}
```

В основной программе проверяется правильность работы функций управления памятью. В качестве примера показано, как инициализировать эти строки и выводить на дисплей при помощи стандартной функции вывода **printf** и в стандартный поток **cout**. В двух последних строчках показано использование статической строки этого типа:

```
void main(void)
{
   string *a = new string("Hello a\n");
   string *b = new string;
   *b = "Hello b\n";
   printf("%s", (char*)*b);
   cout << (char*)*a;
   delete b;
   delete a;
   string c("Hello c\n");
   cout << (char*)c;
}
```

Наследование

Наследование — второе важнейшее понятие объектно-ориентированного программирования. Оно позволяет объектам одного класса наследовать элементы другого класса, то есть наследовать атрибуты и поведение. Класс, элементы ко

торого наследуются, называется **базовым**. Новый класс называется **производным**. Наследование дает возможность использовать код базового класса, при необходимости добавляя новые функциональные черты производным классам.

Предположим, что нам нужно описать геометрические фигуры, такие, как круг и квадрат. Каждая из этих фигур обладает некоторыми свойствами, которые являются общими, например, периметр, площадь, основной размер. Тогда мы можем (и должны) сначала описать класс, отражающий наиболее общие черты. Назовем этот класс *figure*:

```
class figure {
   double ah, sh;
public:
   figure(double size) : ah(0), sh(size) {}
   double area(void) { return ah; }
   double size(void) { return sh; }
};
```

Можно определить представителя класса *figure*, но смысла в этом нет, так как этот класс не описывает никакого реального объекта. Для описания конкретных фигур мы используем дополнительные классы. В следующем примере показано, как описать два новых класса как производные от базового *figure*:

```
class circle : public figure {
public:
   circle(double size) : figure(size) { ah = 0.78 * size * size; }
};
class square : public figure {
public:
   square(double size) : figure(size) { ah = size * size; }
};
```

Базовый класс указывается после имени производного класса через двоеточие. Перед именем базового класса указывается тип доступа к элементам базового класса, в нашем случае **public**. Производный класс наследует элементы базового класса, как данные, так и функции, то есть объект производного класса в нашем случае обладает функциями-элементами **area** и **size**. Для производных классов мы определили конструкторы с одним параметром — основным размером объекта. Так как конструктор базового класса также требует один параметр, мы инициализируем базовый класс в списке инициализации. В теле конструктора по заданному основному размеру мы вычисляем площадь объекта.

Однако попытка скомпилировать эту программу вызывает ошибку — элемент данных **ah** базового класса недоступен для производного класса. Здесь мы должны знать, какой доступ имеет производный класс к элементам базового. Результирующий доступ определяется типом доступа элемента базового класса и типом доступа наследования в соответствии со следующей таблицей:

Доступ наследования	*Доступ в базовом классе*	*Доступ в производном классе*
public	**public** **protected** **private**	**public** **protected** **private**
protected	**public** **protected** **private**	**protected** **protected** **private**
private	**public** **protected** **private**	**private** **private** **private**

Как видно из этой таблицы, результирующий тип доступа получается не выше типа доступа наследования. При наследовании **public** результирующий тип доступа не изменяется: для элемента, объявленного в базовом классе как **private**, тип доступа будет **private** — только для элементов базового класса.

Тип доступа **protected** означает, что элемент базового класса доступен только элементам базового и производных классов. Этот тип доступа, таким образом, имеет смысл только тогда, когда существует наследование. В нашем случае мы должны объявить элементы данных базового класса как **protected**:

```
class figure {
protected:
   double ah, sh;
public:
   . . .
};
```

Чтобы убедиться, что объекты производного класса обладают функциями-элементами базового, следует попробовать пример:

```
void main(void)
{
   circle c(1);
   double a = c.area();
}
```

Мы можем добавить в производные классы дополнительные функции, например, для вывода фигуры:

```
class circle : public figure {
public:
   circle(double size) : figure(size) { ah = 0.78 * size * size; }
   void draw(void) { printf("\nThis is cirlce.draw"); }
};
class square : public figure {
public:
   square(double size) : figure(size) { ah = size * size; }
   void draw(void) { printf("\nThis is square.draw"); }
};
void main(void)
{
   circle c(4);
   square s(3);
   c.draw();
   s.draw();
}
```

Наследование, в котором участвует один базовый класс, называется **простым**.

Частное наследование

Если базовый класс наследуется со спецификатором доступа **private**, его открытые элементы будут недоступны в производном классе. Можно разрешить некоторым элементам базового класса частный доступ, объявив их в секции **public** производного класса:

```
class A {
public:
   void f(void) {}
   void g(void) {}
};
```

```
class B : private A {
public:
   A::f;
};
```

Множественное наследование

Множественное наследование возникает, когда производный класс имеет несколько базовых. Например, мы можем создать класс *color*, который выполняет операции с цветом, и использовать его в качестве одного из базовых для геометрических фигур:

```
class color {
   long ch;
public:
   color(long c) : ch(c) {}
   void halftone(void) {}
};

class figure {
protected:
   double ah, sh;
public:
   figure(double size) : ah(0), sh(size) {}
   double area(void) { return ah; }
   double size(void) { return sh; }
};

class circle : public figure, public color {
public:
   circle(double size) : figure(size), color(0) {
     ah = 0.78*size*size;
   }
};

class square : public figure, public color {
public:
   square(double size) : figure(size), color(0) { ah = size * size; }
};

void main(void)
{
   circle a(1);
   double d = a.area();
   a.halftone();
}
```

Неоднозначность при множественном наследовании

В иерархии классов с множественным наследованием может возникнуть неоднозначность, когда класс косвенно наследует два экземпляра одного базового класса. В следующем примере класс `D` наследует два экземпляра класса `A`:

```
class A {
   int ih;
public:
   int get(void) { return ih; }
};

class B: public A {};
class C: public A {};
class D: public B, public C {};
```

```
void main(void)
{
   D d;
   d.get();   /* неоднозначность */
}
```

Избежать этой неоднозначности можно, используя операцию разрешения видимости:

```
void main(void)
{
   D d;
   d.B::get();   /* или d.C::get() */
}
```

Рис. 1. Неоднозначность и ее разрешение при множественном наследовании

Виртуальный базовый класс

Исключить неоднозначность при сложном множественном наследовании можно, используя наследование, при котором производный класс наследует только один экземпляр базового класса. Для этого в спецификацию базового класса нужно включить ключевое слово **virtual**:

```
class A {
   int ih;
public:
   int get(void) { return ih; }
};
class B: virtual public A {};
class C: virtual public A {};
class D: public B, public C {};
void main(void)
{
   D d;
   d.get();   /* нет неоднозначности */
}
```

Конструкторы, деструкторы и наследование

При наследовании мы сталкиваемся с множеством проблем, связанных с инициализацией производных классов. Конструкторы, как известно, не наследуются. Если конструктор базового класса не имеет параметров, как в следующем примере, то конструктор базового класса **не вызывается** в конструкторе производного:

```
class figure {
protected:
   double ah, sh;
public:
   figure(void) : ah(0), sh(0) {}
   double area(void) { return ah; }
   double size(void) { return sh; }
};

class square : public figure {
public:
   square(double size) { sh = size; ah = sh * sh; }
};
```

В случае, когда базовый класс **figure** не имеет конструктора, в производных классах вызов конструктора базового класса также следует исключить:

```
class figure {
protected:
   double ah, sh;
public:
   double area(void) { return ah; }
   double size(void) { return sh; }
};

class square : public figure {
public:
   square(double size) { sh = size; ah = sh * sh; }
};
```

Если в производном классе не требуется инициализировать базовый класс, конструктор производного класса может отсутствовать. Если конструктор базового класса требует параметр (или несколько параметров), в конструкторе производных классов мы должны вызвать конструктор базового класса в списке инициализации. В этом случае конструктор производного класса **обязателен**:

```
class figure {
   double ah, sh;
public:
   figure(double size) : ah(0), sh(size) {}
   . . .
};
class square : public figure {
public:
   square(double size) : figure(size) { ah = size * size; }
   void draw(void) { printf("\nThis is square.draw"); }
};
```

При множественном наследовании мы также получаем множество вариантов. Если производный класс имеет два базовых с конструкторами с параметрами, мы должны определить конструктор производного класса и вызвать конструкторы базовых в списке инициализации (см. ранее пример с классами *figure*, *color*, *circle* и *square*). Мы можем также варьировать число параметров конструктора производного класса. Вот еще один пример иерархии классов, в которой в производном классе конструктор требует определить все характеристики объекта:

```
class color {
   long ch;
public:
   color(long c) : ch(c) {}
   void halftone(void) {}
};
```

```
class figure {
protected:
   double ah, sh;
public:
   figure(double size) : ah(0), sh(size) {}
};

class circle : public figure, public color {
public:
   circle(double s, long c) : figure(s), color(c) {
      ah = 0.78 * size * size;
   }
};

void main(void)
{
   circle a(3, 256);
}
```

Может оказаться, что порядок инициализации базовых классов имеет значение. В этом случае следует уточнить список инициализации, используя трассировку программы для контроля правильности инициализации объектов. В целом разработка иерархии классов оказывается трудоемким процессом, требующим тщательного обдумывания каждого элемента как базовых, так и производных классов, и выбора наиболее элегантного, красивого и простого варианта.

С деструкторами проще. Деструкторы наследуются и вызываются для базовых классов автоматически при разрушении объекта производного класса. При необходимости деструктор базового класса может быть вызван явно как обычная функция класса. Дополнительно о деструкторах см. «Виртуальные деструкторы».

Полиморфизм

Это третье из ключевых понятий объектно-ориентированного программирования, непосредственно связанное с наследованием, потому что полиморфизм может возникать только в случае иерархии классов и при выполнении определенных условий. Это понятие является наиболее сложным для восприятия и требует некоторых умственных усилий для осмысления. Еще более сложным является применение полиморфизма. Как правило, полиморфизм есть *необходимость*, а не следствие иерархии классов. В конкретных случаях программирования применение полиморфизма должно обоснованно вытекать из невозможности достичь желаемый результат как-то иначе. Это не значит, что полиморфизм — крайняя мера при решении той или иной задачи. Напротив, все современное программирование зиждется на концепции полиморфизма, потому что она наиболее полно и точно отвечает отношениям между сущностями реального мира, которые программист описывает при помощи иерархий классов. Однако полиморфизм проявляется не автоматически, а в результате вполне определенных, осознанных действий программиста. Поэтому программист должен хорошо разбираться в условиях возникновения полиморфизма и соблюдать их для получения требуемого эффекта.

Ключевую роль в возникновении полиморфизма играют **указатели**. Перед изучением полиморфизма следует быть уверенным, что вы понимаете сущность

указателей и приемы работы с ними. Прежде всего, следует обратить внимание на то, что в языке Си указатели **типизированы**. Указатель, являясь адресом области памяти (то есть просто числом), не несет никакой информации о том, как эта область распределяется между элементами данных, из которых она на самом деле состоит. Эту информацию формирует компилятор, основываясь на типе, с которым указатель объявлен. В применении к классам нас интересуют только указатели на структуры данных, которые, как правило, состоят из множества элементов. Рассмотрим простейший пример:

```
struct structa {
   char i, j;
};

struct structb {
   int m;
};

void main(void)
{
   structa *a = new structa;
   a->i = 0;
   a->j = 1;
   int k = ((structb*)a)->m;
}
```

В примере объявлен указатель на структуру `structa`, которая состоит из двух байтовых элементов. Далее этот указатель инициализируется с созданием новой (динамической) структуры и элементам этой структуры присваиваются некоторые значения. В последней строке мы меняем тип указателя при помощи конструкция преобразования типа `(structb*)`, что заставляет компилятор рассматривать область памяти, выделенную под структуру `structa`, как структуру типа `structb`. В результате мы получаем доступ к элементу `m` структуры `structb`. В данном случае структуры подобраны таким образом, чтобы получить осмысленный результат, поэтому мы получаем число 256, которое легко вычислить, если вспомнить, как хранятся в памяти компьютера целые числа типа `int`. Этот пример показывает, что через указатель можно получить произвольный доступ к элементам области памяти, если подобрать подходящий (вспомогательный) тип. Иначе можно сказать, что указатель может указывать на произвольные, разные структуры, независимо от объявленного типа указателя. В конечном итоге то, что мы можем получить через указатель, определяется его типом (или типом его преобразования).

Вернемся к полиморфизму. Прежде всего нам нужна задача, которую мы не сможем решить обычными, известными нам методами. Предположим, что нам требуется список геометрических фигур, определенных ранее в виде иерархии классов. Легко представить себе пример приложения, которому такой список может понадобиться — это может быть графический редактор. Исключительно для простоты восприятия мы реализуем список в виде массива указателей, а описания классов упростим до предела. Перед нами сразу возникает проблема — при объявлении массива мы должны указать тип хранящихся в нем указателей. Так как во время работы приложения могут быть созданы разные объекты в произвольном порядке, мы не можем использовать указатели на производные

классы **circle** и **square**. Однако мы можем использовать указатели на базовый класс:

```
class figure {
};
class circle : public figure {
public:
   void draw(void) { printf("\nThis is cirlce"); }
};
class square : public figure {
public:
   void draw(void) { printf("\nThis is square"); }
};
void main(void)
{
   figure *f[2];
   f[0] = new circle;
   f[1] = new square;
}
```

Все замечательно — в массиве запоминаются указатели на представителей производных классов, хотя объявлены они как указатели на базовый класс. Но одной из целей приложения является рисование объектов. Так как каждая фигура рисуется по-своему, мы в производных классах определили соответствующие методы. Теперь мы пытаемся вызвать эти методы, чтобы нарисовать всю картинку:

```
void main(void)
{
   figure *f[2];
   f[0] = new circle;
   f[1] = new square;
   f[0]->draw();   /* ошибка */
   f[1]->draw();   /* ошибка */
}
```

Ничего не выходит. Указатель имеет тип базового класса, в котором метод **draw** не описан. Вспоминая о возможности преобразования указателя, мы пробуем следующее:

```
void main(void)
{
   figure *f[2];
   f[0] = new circle;
   f[1] = new square;
   ((circle*)f[0])->draw();
   ((square*)f[1])->draw();
}
```

Теперь все работает, но... но в реальной программе мы не можем знать наверняка, какой элемент массива указатель на какой объект хранит. Реально мы должны написать:

```
void main(void)
{
   figure *f[2];
   f[0] = new circle;
   f[1] = new square;
   for (int i = 0; i < 2; i++) {
      f[i]->draw();
   }
}
```

Опять мы возвращаемся к ошибке — класс **figure** не имеет метода **draw**. Здесь мы начинаем понимать, что базовый класс **обязан** описывать этот метод, иначе мы никогда не сможем его вызвать. Редактируем базовый класс:

```
class figure {
   double sh;
public:
   figure(double s) : sh(s) {}
   void draw(void) { printf("\nThis is figure"); }
};
```

Снова пробуем нарисовать объекты. Увы... Вместо того чтобы «нарисовать» круг и квадрат, наша программа «рисует» (абстрактные) фигуры. В чем же теперь дело?

Именно здесь мы сталкиваемся с неразрешимой обычными методами проблемой. Дело в том, что функция-элемент **draw** определяется как принадлежащая определенному классу. **Всегда** будет вызываться та функция, которая принадлежит типу указателя. Раз указатель у нас имеет тип **figure**, то и вызывается функция **draw** этого класса. Проблема в том, что мы не можем во время компиляции программы точно указать, какая конкретно функция **draw** (из трех имеющихся) должна быть вызвана (фактически для компиляции требуется адрес функции). О том, какая функция нам действительно нужна, мы узнаем после компиляции, во время работы программы, когда пользователь выберет для рисования те или иные объекты. Нам нужен механизм, при помощи которого во время исполнения программы обращение к методу **draw** через указатель на базовый тип вызовет метод нужного производного класса.

Виртуальные функции и позднее связывание

Virtual **(англ.)** — фактический

Для того чтобы наша программа заработала, не хватает всего лишь одного этого слова! Попробуем:

```
class figure {
   double sh;
public:
   figure(double s) : sh(s) {}
   virtual void draw(void) { printf("\nThis is figure"); }
};
```

Ключевое слово (вот уж действительно — ключевое!) **virtual** предписывает компилятору внести глобальные изменения в код программы, а именно: включить в состав *каждого класса* некоторую таблицу **vtbl**, а в состав *каждого представителя* класса — указатель на нее **vptr** (в виде одного из элементов данных). Таблица **vtbl**, называемая *таблицей виртуальных методов*, содержит адреса фактических функций данного класса. Когда мы генерируем новый динамический объект при помощи генератора **new**, в него автоматически и неявно добавляется указатель на таблицу виртуальных методов **требуемого** класса, указанного в генераторе **new**.

Например, при компиляции строки:

```
   f[0] = new circle;
```

область памяти нового объекта содержит указатель на **vtbl**, сгенерированную для класса **circle**.

Что же происходит во время работы программы с таблицами виртуальных методов? В основе полиморфизма лежит **механизм позднего связывания**. Вспомним — мы не можем на этапе компиляции (сейчас, в тот момент, когда пишем программу) сказать, какая функция нам потребуется. Поэтому и компилятор (компоновщик) не может подставить в соответствующую машинную инструкцию адрес этой нужной функции. Мы узнаем, какая функция нужна, позднее, во время работы программы. Именно тогда, позже, мы сможем вычислить адрес правильной функции. Этот механизм называется поздним связыванием, в противоположность *раннему связыванию* (связыванию на этапе компиляции), которое используется обычно.

При позднем связывании адрес функции вычисляется следующим образом: через указатель на объект получить указатель на **vtbl** (положение которого в объекте фиксировано), затем по таблице виртуальных методов для базового класса вычислить смещение адреса требуемой функции относительно начала **vtbl** (это, кстати, выполняется и может быть выполнено только на этапе компиляции), добавить это смещение к указателю на **vtbl** объекта и косвенно вызвать функцию, используя полученное число как адрес нужной подпрограммы.

Сложно, не правда ли? Да, и сложно, и долго. Позднее связывание замедляет работу программы, но взамен мы получаем полиморфизм — возможность трактовать объекты одного типа как разные. Полиморфизм означает *множественность форм*. Используемый при полиморфизме указатель — всегда базового типа. Вызываемая функция — всегда производного типа (если она существует, разумеется, — иначе вызывается функция базового типа). Через указатель на базовый тип мы получаем множество форм, множество проявлений, которые присущи конкретным объектам разных типов.

Более точное определение полиморфизма — возможность доступа к фактическим методам производных классов через указатель на базовый тип. Условия наступления полиморфизма — указатели на базовый тип, используемые для объектов, виртуальные функции в базовом типе и соответствующие (не обязательно с ключевым словом **virtual**) функции в производном типе. Виртуальные функции выбираются *в зависимости от типа объекта*, на который указывает указатель, а *не* в зависимости от типа указателя, и позволяют наиболее полно использовать *объектно-ориентированную парадигму* программирования!

Важно — полиморфизм возможен *только при использования указателей или ссылок*.

Далее в качестве примера показано, как реализовать связанный список объектов.

```
#include <stdio.h>
class figure {
public:
   figure *next;
   figure(void) : next(0) {}
   virtual void draw(void) {}
};
```

```
class figures : public figure {
   figure *first;
public:
   figures(void) : first(0) {}
   void insert(figure *);
   void drawall(void);
};

void figures::insert(figure *p)
{
   p->next = first;
   first = p;
}

void figures::drawall(void)
{
   figure *temp = first;
   while (temp != 0) {
      temp->draw();
      temp = temp->next;
   }
}

class circle : public figure {
public:
   void draw(void) { printf("\nCirlce"); }
};

class square : public figure {
public:
   void draw(void) { printf("\nSquare"); }
};

void main(void)
{
   figures *f = new figures;
   f->insert(new circle);
   f->insert(new square);
   f->drawall();
}
```

В число элементов данных базового класса включен указатель **next***, который служит для связи объектов в односвязный список. Сам список представляет класс **figures**, производный от базового, что, вообще говоря, не обязательно. В функции этого класса входит управление списком. Элемент **first*** класса **figures** — это указатель на начало списка объектов. Функция **insert** включает новый объект в начало списка, используя указатель на включаемый элемент в качестве параметра. Функция **remove** исключает объект из списка. Эта функция не включена в состав класса для облегчения восприятия. Функция **show** выводит список («рисует») на устройство вывода. Следует обратить внимание на то, как происходит перемещение по списку при помощи вспомогательного временного указателя **temp**. Дополнительные функции могут включать перемещение одного из элементов в начало или в конец списка. В основной функции показано, как включить новые объекты в список. Наилучшим примером построения иерархии классов может служить система **Turbo Vision**. Исходные тексты классов прилагаются к среде разработки **Borland C++ 3.01** для возможности построения при

ложений для **MS-DOS**. Их изучение поможет лучше разобраться с тем, как используются классы, наследование и полиморфизм на практике.

В классе **figure** метод **draw** пустой. Предполагается, что объекты этого класса никогда не генерируются, поэтому определение каких-либо действий внутри этой функции не имеет смысла. Ее назначение заключается исключительно в обеспечении базового типа определенным виртуальным методом. В объектно-ориентированном программировании часто случается, что функции (методы) не содержат кода. Тем не менее, роль этих пустых функций чрезвычайно велика. Они определяют общее поведение объектов производных классов, которое мы можем использовать во время полиморфизма.

Что делать, если некоторый производный класс определяет метод, который отсутствует в базовом классе и поэтому недоступен через указатель на базовый класс? Разработка классов с учетом возможного полиморфизма обязывает программиста тщательно продумывать набор методов базового класса и, тем не менее, иногда включение дополнительного метода в базовый класс представляется неоправданным. В этом случае следует точно знать тип объекта, на который указывает указатель базового типа и применить явное преобразование указателя к типу производного класса, например:

```
#include <stdio.h>
class figure {
public:
   virtual void draw(void) {}
};

class circle : public figure {
public:
   void draw(void) { printf("\nCirlce"); }
   void paint(void) { /* некоторые действия */ }
};
void main(void)
{
   figure *c = new circle;
   ((circle*)c)->paint();
}
```

Виртуальный деструктор

В иерархии классов деструкторы базовых классов могут и *должны быть виртуальными*. Если в базовом классе деструктор не описан как виртуальный, то при разрушении объекта производного класса, на который ссылается указатель, имеющий тип базового класса, будет вызван деструктор базового класса, а не производного. Если деструктор производного класса выполняет дополнительные действия по отношению к деструктору базового класса, эти действия не будут выполнены и правильное прекращение жизни объекта не произойдет. Следующий пример поясняет это:

```
struct figure_data {
   int a, b, c;
};
class figure {
public:
   ~figure(void) {}
};
```

```
class circle : public figure {
   figure_data *d;
public:
   circle(void) { d = new figure_data; }
   ~circle(void) { delete d; }
};

void main(void)
{
   figure *c = new circle;
   delete c;   /* вызывается деструктор ~figure */
}
```

Файлы и модули

Общепринятый подход к созданию и использованию классов заключается в разбиении кода на модули (файлы). При этом также достигается так называемое *сокрытие данных*. Как известно, язык Си поддерживает исходные модули двух типов — заголовочные файлы (типа **.h**) и файлы текста на языке Си (типа **.c**). В дополнение к ним язык Си++ поддерживает файлы текста на языке Си++ (типа **.cpp**). В языке Си заголовочные файлы содержат описания макросов, в том числе констант, глобальных переменных, описания структур, перечислений и т. п. При использовании классов в заголовочные файлы дополнительно включают описания классов. Реализацию классов размещают в отдельных файлах текста на языке Си++.

Возникает несколько вариантов размещения описаний и реализаций. Описания, как правило, занимают небольшой объем, поэтому описания нескольких классов могут быть размещены в одном заголовочном файле. Реализации, напротив, обычно значительно больше по размеру, поэтому реализации разных классов размещают в разных файлах. Одним из подходов является размещение каждого отдельного класса в двух файлах — заголовочном для описания и файле текста для реализации. Файл заголовка и файл реализации в этом случае имеют одинаковое имя и различаются только расширением. Это обеспечивает наибольшую гибкость при использовании классов. Однако в случае, если классов много, такой подход порождает большое количество маленьких заголовочных файлов и в конечном итоге значительно затрудняет правильное использование директивы **#include**. Для больших проектов эти трудности могут оказаться значительными, вплоть до полной невозможности построить программу.

Компромиссом является включение в один заголовочный файл описаний нескольких связанных между собой классов, например, классов, образующих иерархию. В качестве примера можно привести иерархию классов **Turbo Vision**. Базовый класс **TView** описывает абстрактное прямоугольное изображение. Производными от **TVew** являются **TGroup** и **TProgram** — абстрактная группа изображений и абстрактное приложение. Все классы можно описать в одном заголовочном файле с именем **views.h**, а реализации описать в отдельных файлах с соответствующими классам именами.

Некоторые методы классов могут быть подставляемыми (**inline**). В этом случае их реализация может быть определена внутри описания класса, внутри заголовочного файла (но вне описания класса), и внутри файла реализации.

Для сокрытия реализации вместо исходных файлов текстов программист может предоставлять пользователям класса объектные модули (файлы типа **.obj**) или файлы библиотек (файлы типа **.lib**), которые могут быть включены в проект наравне с файлами текста.

В качестве примера рассмотрим разбиение на файлы для иерархии классов геометрических фигур. Заголовочный файл **objetcs.h** определяет вспомогательные структуры и глобальные константы и переменные:

```
//objects.h
#if !defined(__OBJECTS_H)
#define      __OBJECTS_H
/* позиция объекта */
struct location {
   int x, y;
};
/* данные объекта */
struct objectdata {
   double s;
   location loc;
};
/* начальное положение объекта */
const location initloc = { 0, 0 };
#endif
```

Заголовочный файл **figures.h** содержит описания основных классов:

```
// figures.h
#if !defined(__FIGURES_H)
#define      __FIGURES_H
#include "objects.h"
/* класс figure */
class figure {
protected:
   objectdata *data;
public:
   figure *next;
   figure(void);
   virtual ~figure(void);
   void moveto(location);
   virtual void draw(void);
};
/* класс figures */
class figures : public figure {
   figure *first;
public:
   figures(void);
   ~figures(void);
   void insert(figure *);
   void remove(figure *);
   void drawall(void);
};
/* класс circle */
class circle : public figure {
public:
   circle(double);
   ~circle(void);
   void draw(void);
};
/* класс square */
class square : public figure {
public:
```

```
   square(double);
   ~square(void);
   void draw(void);
};
#endif
```

Файл **figure.cpp** определяет класс **figure**:

```
// figure.cpp
#include "figures.h"
/* конструктор */
figure::figure(void) : next(0), data(0)
{
}
/* деструктор */
figure::~figure(void)
{
}
/* устанавливает положение объекта */
void figure::moveto(location newloc)
{
   if (data != 0) data->loc = newloc;
}
/* рисует объект */
void figure::draw(void)
{
}
```

Файл **figures.cpp** определяет класс **figures**:

```
//figures.cpp
#include "figures.h"
/* конструктор */
figures::figures(void) : first(0)
{
}
/* деструктор */
figures::~figures(void)
{
   figure *temp;
   while (first) {
      temp = first;
      first = first->next;
      delete temp;
   }
}
/* добавляет новый объект в группу */
void figures::insert(figure *p)
{
   p->next = first;
   first = p;
}
/* удаляет объект из группы */
void figures::remove(figure *p)
{
   if (first == 0) return;
   if (first == p) {
      first = first->next;
      delete p;
      return;
   }
   figure *temp = first;
   while (temp->next != 0) {
```

```
      if (temp->next == p) {
         temp->next = p->next;
         delete p;
         break;
      }
      temp = temp->next;
   }
}
/* рисует все объекты группы */
void figures::drawall(void)
{
   figure *temp = first;
   while (temp != 0) {
      temp->draw();
      temp = temp->next;
   }
}
```

Файл **circle.cpp** определяет класс **circle**:

```
//circle.cpp
#include <stdio.h>
#include "figures.h"
/* конструктор */
circle::circle(double size)
{
   data = new objectdata;
   data->s = size;
   moveto(initloc);
}
/* деструктор */
circle::~circle(void)
{
   delete data;
}
/* рисует объект */
void circle::draw(void)
{
   printf("\nCircle");
}
```

Класс **square** определяется аналогично классу **circle** в файле **square.cpp**. Последний файл **figs.cpp** содержит основную функцию:

```
//figs.cpp
#include "figures.h"
figures *f;
void main(void)
{
   f = new figures;
   f->insert(new square(2));
   circle *c = new circle(1.5);
   f->insert(c);
   f->insert(new square(2));
   f->remove(c);
}
```

Действия со списком фигур в основной функции приведены для демонстрации. Полезно выполнить трассировку приведенного примера иерархии для лучшего понимания взаимодействия классов. Для тестирования этого примера все файлы **.cpp** нужно включить в состав нового проекта.

Чистые виртуальные функции и абстрактные классы

Чистой виртуальной функций называется виртуальная функция, тело которой не задано и представляет собой *чистый спецификатор =0*. В примере с иерархией классов геометрических фигур виртуальная функция **draw** класса **fugure** определена с пустым телом и может быть объявлена как чистая:

```
class figure {
public:
   figure *next;
   figure(void) : next(0) {}
   virtual void draw(void) = 0;
};
```

Для чистой виртуальной функции тело не требуется. Предполагается, что такая функция переопределяется в производных классах.

Абстрактным классом называется класс, который содержит минимум одну чистую виртуальную функцию. Абстрактные классы могут быть использованы только в качестве базовых для других классов. В приведенном выше примере класс **fugure** является абстрактным. К абстрактным классам применяются следующие правила:

- абстрактный класс нельзя использовать в качестве типа аргумента или возвращаемого значения;
- абстрактный класс нельзя использовать для явного преобразования;
- нельзя объявить представителя абстрактного класса;
- можно объявить указатель или ссылку на абстрактный класс;
- если класс, производный от абстрактного, не определяет *все* чистые виртуальные функции, он также является абстрактным.

В комплекте поставки сред разработки есть множество абстрактных классов и шаблонов (см. далее «Шаблоны»), которые предназначены для использования в качестве базовых при определении новых классов.

Абстрактные классы и шаблоны имеют большое значение при использовании новых технологий программирования, таких, как **ActiveX**.

Шаблоны

Шаблон — это обобщенное относительно типа определение функции или класса. Шаблоны являются основой для создания функций или классов, предназначенных для применения с определенным типом, являющимся параметром конкретизации шаблона. Шаблоны Си++ иногда называют параметризованными типами. Они дают возможность компилировать новые функции или классы, задавая типы в качестве параметров.

Шаблоны функций

Шаблон функции — это обобщенное определение функции, используемое для генерирования представителя данной функции для использования с определенным типом. Синтаксис шаблона функции:

```
template <список_аргуметнов_шаблона> возвращаемый_тип
имя_функции(параметры)
{
   // тело функции
}
```

Например, следующий шаблон описывает функцию обмена двух значений:

```
template <class T> void swap(T& a, T& b)
{
   T temp = a; a = b; b = temp;
}
```

Использовать шаблон функции просто:

```
void main(void)
{
   int a = 1, b = 2;
   double c = 3.3, d = 4.4;
   swap(a, b);
   swap(c, d);
}
```

Еще один пример показывает использование шаблона для функции сортировки массива значений некоторого типа. Определение шаблона расположено в заголовочном файле **sorttpl.h**:

```
//sorttpl.h

#if !defined(__SORTTPL_H)
#define      __SORTTPL_H
#include <stddef.h>

/* сортировка вставками */

template <class T> void sort(T a[], size_t n)
{
   for (int i = 1; i < n; i++) {
      T t = a[i];
      for (int j = i - 1; ; j--) {
         if ((j < 0) || (a[j] <= t)) break;
         a[j + 1] = a[j];
      }
      a[j + 1] = t;
   }
}
#endif
```

В основной функции шаблон **sort** используется для сортировки массивов двух разных типов:

```
//sorttpl.cpp

#include <stdio.h>
#include "sorttpl.h"

int a[] = { 3, 1, 5, 2, 4 };
double b[] = { 3.3, 1.1, 5.5, 2.2, 4.4 };

void main(void)
{
   sort(a, sizeof(a) / sizeof(int));
   sort(b, sizeof(b) / sizeof(double));
}
```

Для предварительного объявления шаблона функции можно создать прототип:

```
template <class T> void sort(T a[], size_t n);
```

Шаблон функции **sort** можно использовать для определяемых пользователей типов (классов), если тип перегружает операцию **<=**, которая используется для сравнения элементов массива.

Перегрузка шаблонов функции

Шаблон функции может быть перегружен, если два или более шаблонов определяют функцию с одним именем и разным числом или разным типом параметров. Например:

```
template <class T> T getmax(T a, T b) { /* . . . */ }
template <class T> T getmax(T a[], size_t n) { /* . . . */ }
```

Если обобщенный шаблон не годится для некоторого типа данных, то можно перегрузить шаблонную функцию для этого типа. Эту функцию называют *специализированной* функцией шаблона, а определение такой функции — *специализацией* шаблона функции. Например, функция шаблона **getmax** не может быть применена к строкам, поэтому для типа **char*** определяется специализированная функция:

```
template <class T> T getmax(T a, T b)
{
   return a > b ? a : b;
}

char* getmax(char *a, char *b)
{
   return strcmp(a, b) > 0 ? a : b;
}
```

Разрешение ссылки на функцию

Компилятор разрешает ссылку на функцию по правилам:

ищет не шаблонную функцию, параметры которой соответствуют параметрам вызова;
если функция не найдена, ищет шаблон, по которому можно сгенерировать функцию с точным соответствием параметров;
если никакой шаблон не обеспечивает точного соответствия параметров, рассматривает не шаблонные функции на предмет возможного преобразования типов параметров.

Шаблоны классов

Шаблон класса дает обобщенное определение класса для произвольного типа или константы. Конкретный тип (конкретная константа) указывается во время создания представителя класса. Шаблон определяет как элементы данных, так функции-элементы класса. Синтаксис шаблона класса:

```
template <список_параметров_шаблона> class имя_класса {
   /* тело класса */
};
```

Параметры шаблона класса заключаются в угловые скобки и отделяются друг от друга запятыми. Параметром может быть:

- имя типа, за которым следует идентификатор (нетипированный параметр);
- ключевое слово **class**, за которым следует идентификатор (типированный параметр).

Значением нетипированного параметра может быть только константное выражение. Значением типированного параметра может быть имя действительного типа.

Следующий пример определяет шаблон класса «массив элементов»:

```
//atpl.h
#if !defined(__ATPL_H)
#define      __ATPL_H
template <class T, int n> class Array {
   T a[n];
public:
   T& operator [](int i) { return a[i]; }
};
#endif
```

При использовании этого шаблона нужно указать параметры (в угловых скобках):

```
#include "atpl.h"
void main(void)
{
   Array<int, 5> a;
   a[0] = 1;
   int j = a[0];
}
```

Если функция-элемент шаблона класса определяется вне класса, используется следующий синтаксис:

```
template <список_параметров_шаблона>
возвращаемый_тип имя_класса<параметры_шаблона>::имя_функции(параметры)
{
   /* тело функции */
}
```

В качестве примере рассмотрим класс, предназначенный для работы со списками объектов произвольного типа. В качестве класса объектов используется иерархия классов геометрических фигур, определенная в файле **figurest.h** следующим образом:

```
#if !defined(__FIGUREST_H)
#define      __FIGUREST_H
#include <stdio.h>
struct figure {
   virtual void draw(void) = 0;
};
struct circle : public figure {
   void draw(void) { printf("\nCircle"); }
};
struct square : public figure {
   void draw(void) { printf("\nSquare"); }
};
#endif
```

Вместо класса **figures** введем два новых класса — элемент списка **node** и список **list**. Прежде, чем перейти к шаблону, определим эти два класса обычным обра

зом. Класс **node** содержит два элемента данных: **next** для указания на следующий элемент списка и **obj** для указания на объект. Для этого класса определен конструктор (файл **figurest.cpp**):

```
#include "figurest.h"
struct node {
   node *next;
   figure *obj;
   node(figure *f) : next(0), obj(f) {}
};
```

В этом же файле ниже идет описание класса **list**. Класс упрощен в целях демонстрации:

```
class list {
   node *first;
public:
   list(void) : first(0) {}
   ~list(void);
   void insert(figure *);
   void drawall(void);
};
```

Далее в файле определяются деструктор и методы класса **list**:

```
list::~list(void)
{
   node *temp;
   while (first) {
      temp = first;
      first = first->next;
      delete temp;
   }
}
void list::insert(figure *p)
{
   node *temp = new node(p);
   temp->next = first;
   first = temp;
}
void list::drawall(void)
{
   node *temp = first;
   while (temp) {
      temp->obj->draw();
      temp = temp->next;
   }
}
```

В основной функции демонстрируется использование класса **list**:

```
void main(void)
{
   list a;
   figure *c = new circle;
   a.insert(c);
   a.insert(new circle);
   a.insert(new square);
   a.drawall();
}
```

Теперь мы можем определить шаблоны для классов **node** и **list**. Сделать это легко. Объявляем параметр шаблона и заменяем тип **figure** на параметр. Для класса **node** шаблон имеет вид:

```
template <class T> struct node {
   node *next;
   T *obj;
   node(T *f) : next(0), obj(f) {}
};
```

Так как класс **list** использует класс **node**, каждое вхождение типа **node** должно быть конкретизировано параметром класса **list**:

```
template <class T> class list {
   node<T> *first;
public:
   list(void) : first(0) {}
   ~list(void);
   void insert(T *);
   void drawall(void);
};
```

Методы класса **list**, вынесенные из описания, определяются как шаблоны так:

```
template <class T> list<T>::~list(void)
{
   node<T> *temp;
   while (first) {
      temp = first;
      first = first->next;
      delete temp;
   }
}
template <class T> void list<T>::insert(T *p)
{
   node<T> *t = new node<T>(p);
   t->next = first;
   first = t;
}
template <class T> void list<T>::drawall(void)
{
   node<T> *temp = first;
   while (temp) {
      ((T *)temp->obj)->draw();
      temp = temp->next;
   }
}
```

В основной функции шаблон класса **list** конкретизируется для класса **figure**:

```
void main(void)
{
   list<figure> a;
   a.insert(new circle);
   a.insert(new square);
   a.drawall();
}
```

Шаблоны классов **node** и **list** теперь могут быть использованы теперь для генерирования классов, которые формируют список любых объектов. Предполагается, что классы объектов определяют метод **void draw(void)**.

Так же, как для шаблона функции, отдельные методы шаблона класса или класс целиком могут быть специализированы для некоторого типа. Если специализи

руется класс целиком, следует сделать это после определения шаблона класса и определить все функции-элементы.

Дополнительные сведения о классах

Структуры и объединения

Объединение (**union**) также может является классом, если в его описание включены функции-элементы. Тип доступа по умолчанию для объединения — **public**.

Указатели на функции-элементы

Можно определить указатель на функцию-элемент класса. Первоначально указатели на функцию-элемент класса использовались вместо виртуального механизма. Синтаксис определения:

```
возвращаемый_тип (имя_класса::*имя_указателя)(параметры);
```

Указатель на функцию-элемент используется с операциями **.*** или **->***. Пример:

```
#include <stdio.h>
struct figure {
   figure *next;
   figure(void) : next(0) {}
   virtual void draw(void) {}
   void callfunc(void (figure::*func)(void)) { (*this.*func)(); }
};
struct circle : public figure {
   void draw(void) { printf("\nCircle"); }
};
```

Здесь в классе **figure** определяется функция, использующая указатель **func** на функцию класса, описанную как **void имя(void)**. В теле функции используется операция **.*** для обращения к функции через этот указатель. В основной функции определяется указатель на функцию **figure::draw**:

```
void main(void)
{
   void (figure::*print)(void) = &figure::draw;
   figure *c = new circle;
   c->callfunc(print);
}
```

Использовать функцию **callfunc** можно для вызова любой функции класса **figure**, которая определена как **void имя(void)**. В следующем примере используется дополнительный класс **list** для формирования списка объектов, производных от **figure**. Функция **exec** этого класса предназначена для выполнения произвольной функции класса, производного от **figure**, для всех элементов списка. В этой функции используется операция **->***:

```
class list {
   figure *first;
public:
   list(void) : first(0) {}
   void add(figure *p) { p->next = first; first = p; }
   void exec(void (figure::*)(void));
};
void list::exec(void (figure::*func)(void))
{
   figure *temp = first;
```

```
   while (temp) {
      (temp->*func)();
      temp = temp->next;
   }
}
```

В основной функции метод **exec** используется для вызова функции **draw** класса **figure** с каждым объектом, включенным в список:

```
void main(void)
{
   void (figure::*print)(void) = &figure::draw;
   list a;
   a.add(new circle);
   a.add(new circle);
   a.exec(print);
}
```

Выключение виртуального механизма

Если необходимо вызвать виртуальную функцию не производного, а базового класса, следует указать квалификатор класса (имя класса и операцию разрешения видимости) перед именем функции:

```
struct figure {
   virtual void draw(void) { printf("\nFigure"); }
};
struct circle : public figure {
   void draw(void) { printf("\nCircle"); }
};
void main(void)
{
   figure *c = new cirlce;
   c->figure::draw();
}
```

Потоки

Для управления вводом/выводом язык Си++ предоставляет набор классов, называемый библиотекой потоков **IOStream**. По сравнению с функциями ввода/вывода языка Си потоки обладают рядом преимуществ:

- потоки надежны. Функции **ptintf**, **scanf** языка Си не предусматривают проверку типов и могут вызывать ошибки при вводе или выводе информации. Механизм потоков основан на перегрузке функций, что обеспечивает вызов правильной функции для соответствующего типа данных;
- потоки расширяемы. Мы можем легко определить функции для ввода/вывода новых классов;
- использование потоков предполагает единообразный способ ввода/вывода для разных типов.

Предопределенные потоки

Библиотека **iostream** предоставляет четыре предопределенных потока, ассоциированных со стандартным вводом и выводом:

- **cin** — ассоциирован со стандартным вводом (клавиатурой);
- **cout** — ассоциирован со стандартным выводом (дисплеем);
- **cerr** — ассоциирован со стандартным устройством ошибки с не буферизованным выводом;
- **clog** — ассоциирован со стандартным устройством ошибки с буферизованным выводом.

Операция помещения

Для вывода информации используется класс **ostream** библиотеки потоков и операция помещения (в поток) **<<**. В примере на экран выводится приветствие:

```
#include <iostream.h>
void main(void)
{
   cout << "Hello";
}
```

Сцепление операций помещения

Операция помещения возвращает ссылку на объект класса **ostream**, поэтому несколько операций вывода можно сцеплять в одном операторе:

```
#include <iostream.h>
void main(void)
{
   int i = 1;
   cout << "Number " << i;
}
```

Операция извлечения

Для чтения информации используется объект класса **istream** и операция извлечения из потока **>>**. В следующем примере из стандартного ввода читается несколько значений при помощи *сцепления*:

```
#include <iostream.h>
void main(void)
{
   int i, j, k;
   cout << "Введите 3 целых: ";
   cin >> i >> j >> k;
}
```

Форматирование

Библиотека потоков предусматривает три способа управления форматом выходных данных: форматирующие *функции-элементы*, *флаги* и *манипуляторы*.

Форматирующие функции-элементы

Форматирующие функции-элементы определены в классе **ios**. Функция **width** устанавливает/возвращает значение внутренней переменной ширины поля. При вводе при помощи этой функции задают максимальное число читаемых символов:

```
#include <iostream.h>
const MAXW = 10;
void main(void)
{
   char n[MAXW];
   cin.width(MAXW);
   cin >> n;
}
```

При выполнении этого примера можно ввести с клавиатуры любое число символов, однако в массив **n** попадет не более **девяти**.

При выводе функция **width** задает минимальную ширину поля:

```
#include <iostream.h>
const MAXW = 10;
void main(void)
{
   char n[MAXW];
   cout.width(MAXW);
   cout << "12345";
}
```

Результирующий вывод будет состоять из пяти пробелов и пяти цифр:

```
"     12345".
```

- по умолчанию значение **width** равно **0**;
- **width** обнуляется после каждого помещения в поток.

Функция **fill** устанавливает/возвращает символ-заполнитель поля. Символом-заполнителем по умолчанию является *пробел*. В примере пять лидирующих пробелов заменяются нулями:

```
#include <iostream.h>
const MAXW = 10;
```

```
void main(void)
{
   char n[MAXW];
   cout.width(MAXW);
   cout.fill('0');
   cout << "12345";
}
```

Для вывода вещественного числа можно применить функцию **precision**, которая задает точность. Если установлен один из флагов **scientific** или **fixed**, функция **precision** задает число знаков после десятичной точки, иначе задается общее число цифр.

Флаги форматирования

Флаги задают формат ввода и вывода и являются битовыми полями, хранящимися в переменной **long**.
Флаги можно устанавливать и снимать. Для установки флага используется функция **setf**, для снятия — функция **unsetf**. В качестве параметров функций выступают предопределенные в классе **ios** константы, приведенные в следующей таблице:

Флаг	*По умолчанию*	*Описание*
ios::skipws	установлен	если установлен, при вводе игнорируются предшествующие символы-заполнители
ios::left		если установлен, данные при выводе выравниваются по левой границе поля
ios::right	установлен	если установлен или если сброшены флаги left и internal, данные при выводе выравниваются по правой границе
ios::internal		если установлен, знак числа выводится у левого края поля, число у правого поля
ios::dec	установлен	если установлен, числа выводятся по основанию 10
ios::oct		если установлен, числа выводятся по основанию 8
ios::hex		если установлен, числа выводятся по основанию 16
ios::showbase		если установлен, при выводе добавляется индикатор основания системы счисления
ios::showpoint		если установлен, при выводе вещественного числа показывается десятичная точка
ios::uppercase		если установлен, буквы A..F в шестнадцатиричных числах выводятся в верхнем регистре
ios::showpos		если установлен, выводится знак + для положительных значений
ios::scientific		если установлен, числа выводятся в научной нотации
ios::fixed		если установлен, числа выводятся в нотации с точкой
ios::unitbuf		если установлен, буфер потока опорожняется после каждой операции помещения
ios::stdio		если установлен, потоки stdout и sterr опорожняются после каждой операции помещения

В качестве примера показано применение флага **showpos**:

```
#include <iostream.h>
void main(void) {
   int i = 123;
   cout.setf(ios::showpos);
   cout << i;
}
```

Манипуляторы

Манипуляторы — это функции, которые можно включать цепочку операций помещения и извлечения. Некоторые из манипуляторов управляют флагами. За исключением **setw**, изменения, внесенные манипуляторами, сохраняются до следующей установки. Манипуляторы, не требующие параметров, называются *простыми*. Манипуляторы с параметрами называются *параметризованными*.

Простые манипуляторы приведены в следующей таблице:

Манипулятор	*Описание*
endl	помещает в выходной поток символ конца строки **\n** и вызывает манипулятор **flush**
ends	помещает в выходной поток нулевой символ
flush	принудительно записывает данные (из буфера) на физические устройства
dec	соответствует **ios::dec**
hex	соответствует **ios::hex**
oct	соответствует **ios::oct**
ws	соответствует **ios::skipws** (игнорировать ведущие символы-заполнители)

В примере после вывода значения производится переход на новую строку манипулятором **endl** (в конце слова не единица, а буква «эл» — от **endl**ine):

```
#include <iostream.h>
void main(void)
{
   cout << "12345" <<
endl;
}
```

Параметризованные манипуляторы требуют указания параметра при вызове.

Манипулятор	*Описание*
setbase(int)	задает основание 0, 8, 10, 16. Если 0, при выводе используется основание 10, при вводе используется основание 10, если число не восьмеричное или шестнадцатиричное
resetiosflags(long)	сбрасывает флаги, биты которых установлены в параметре
setiosflags(long)	устанавливает флаги, биты которых установлены в параметре
setfill(int)	задает символ-заполнитель
setprecision(int)	задает ширину поля

Пример показывает применение параметризованных манипуляторов для формирования таблицы вещественных чисел. Обратим внимание на использованный заголовочный файл **iomanip.h**:

```
#include <iomanip.h>
void main(void)
{
   double a[] = { 1.23, 0.0004, -5.6789 };
   cout << setfill('.') << setprecision(3)
        << setiosflags(ios::fixed | ios::right | ios::showpoint);
   int n = sizeof(a) / sizeof(a[0]);
   for (int i = 0; i < n; i++) {
      cout << i << setw(16) << a[i] << endl;
   }
}
```

Ошибки

Элемент данных **state** потока показывает его текущее состояние. Биты этого элемента имеют следующие значения (маски определены перечислением **ios::io_state**):

Бит	*Маска*	*Описание*
0	**eofbit** (=1)	достигнут конец файла
1	**failbit** (=2)	ошибка форматирования при выводе или преобразования при вводе; использование потока возможно после сброса бита
2	**badbit** (=4)	серьезная ошибка, связанная с буфером; использовать поток скорее всего больше нельзя
3	**hardfail** (=8)	неисправимая ошибка (неисправность аппаратуры)

Отсутствие установленных битов описывается маской **goodbit**, равной нулю. Чтение и установка битов состояния производится следующими функциями:

Метод	*Описание*
int rdstate()	возвращает текущее состояние потока
int eof()	возвращает ненулевое значение, если установлен флаг ios::eofbit
int fail()	возвращает ненулевое значение, если установлен один из флагов ios::failbit, ios::badbit или ios::hardfail
int good()	возвращает ненулевое значение, если все флаги сброшены
void clear(int=0)	если параметр равен нулю, все биты очищаются, иначе параметр принимается как состояние ошибки
operator void*()	возвращает нулевой указатель, если установлен один из флагов ios::failbit, ios::badbit или ios::hardfail
int operator!()	возвращает нулевое значение, если установлен один из флагов ios::failbit, ios::badbit или ios::hardfail

Примечание: операция **void*()** неявно вызывается всякий раз при сравнении с нулем переменной, ссылающейся на поток. Это удобно использовать для проверки текущего нормального состояния потока:

```
while (переменная_потока) {
   /* состояние потока нормальное */
}
```

Способы управления состоянием потока

Проверка бита состояния:

```
if (имя_потока.rdstate() & ios::маска_состояния)
```

Сброс бита состояния:

```
имя_потока.clear(имя_потока.rdstate() & ~ios::маска_состояния)
```

Установка бита состояния:

```
имя_потока.clear(имя_потока.rdstate() | ios::маска_состояния)
```

Установка бита состояния:

```
имя_потока.clear(ios::маска_состояния)
```

Сброс всех флагов:

```
имя_потока.clear()
```

Перегрузка операций помещения и извлечения для типов пользователя

Для помещения и извлечения определенных пользователем типов мы должны определить две функции. Для чтения данных из потока перегружается функция **operator >>**, имеющая сигнатуру:

```
istream& operator >> (istream& is, имя_типа& имя_параметра);
```

Для записи в поток перегружается функция **operator <<**, имеющая сигнатуру:

```
ostream& operator << (ostream& os, имя_типа& имя_параметра);
```

В качестве примера рассмотрим тип, состоящий из двух целых чисел, описывающих дробь:

```
#include <iostream.h>
#include <assert.h>
struct rational {
   int num, den;
};
```

Далее мы определяем функцию извлечения, предполагая, что число записывается в формате **num/den**:

```
istream& operator >> (istream& is, rational& x)
{
   int n;
   cin >> n;           /* принимаем числитель */
   char c;
   cin >> c;           /* принимаем разделитель */
   assert(c == '/');   /* проверяем разделитель */
   int d;
   cin >> d;           /* принимаем знаменатель */
   x.num = n;
   x.den = d;
   return is;
}
```

Функция помещения:

```
ostream& operator << (ostream& os, rational& x)
{
   os << x.num << '/' << x.den;
   return os;
}
```

В основной функции проверяем работу этих операций:

```
void main(void)
{
   rational a;
   cin >> a;
   cout << a;
}
```

В случае использования классов рекомендуется определять операции помещения и извлечения как дружественные. Это даст возможность доступа к закрытым элементам данных класса. Например, для класса рациональных чисел описание класса может иметь вид:

```
class rational {
   int num, den;
public:
   rational(void) : num(0), den(1) {}
   friend istream& operator >> (istream&, rational&);
```

```
   friend ostream& operator << (ostream&, rational&);
};
```

Операции извлечения и помещения и основная функция для тестирования этого класса не изменились по сравнению с предыдущим примером.

Функции ввода (поток istream)

Функция **read**

```
istream& istream::read(signed char*, int);
istream& istream::read(unsigned char*, int);
```

Извлекает из потока указанное число символов, записывая их в указанный буфер. Пример:

```
#include <iostream.h>
char buf[11];
void main(void)
{
   cin.read(buf, 10);
}
```

Функция **get**

```
int istream::get(void);
```

Извлекает из потока в переменную один символ. В примере в переменную **c** принимается один (первый) символ из введенных с клавиатуры. Далее он выводится на дисплей:

```
#include <iostream.h>

void main(void)
{
   char c = cin.get();
   cout << c;
}
```

При помощи этой функции можно посимвольно читать поток и анализировать его. Если переменная **c** будет иметь тип **int**, то при выводе в поток **cout** будет выведено число, а не символ. Для вывода символа в этом случае следует применить явное преобразование:

```
#include <iostream.h>

void main(void)
{
   int c = cin.get();
   cout << (char)c;
}
```

Два других варианта этой функции возвращают ссылку на поток:

```
istream& istream::get(unsigned char&);
istream& istream::get(signed char&);
```

В примере эта ссылка используется для обращения к функции **void*()**:

```
char c;
while (cin.get(c)) {
   /* обработка принятых символов и выход из цикла */
}
```

Следующие два варианта **get** читают поток и помещают символы в указанный буфер:

```
istream& istream::get(signed char*, int len, char = '\n');
istream& istream::get(unsigned char*, int len, char = '\n');
```

Чтение завершается при обнаружении в потоке указанного последним параметром символа (по умолчанию конец строки) или при завершении файла. Независимо от завершающего символа в буфер будет помещено не более указанного вторым параметром количества символов. Фактически функция извлекает строку, а не символ.

```
#include <iostream.h>

char buf[] = "abcdef";

void main(void)
{
   cin.get(buf, sizeof(buf));
}
```

Завершающий символ не извлекается из потока и не записывается в буфер, поэтому перед повторным применением функции следует очистить поток:

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

char buf1[20];
char buf2[20];

void main(void)
{
   cout << "Input 1: ";
   cin.get(buf1, sizeof(buf1));
   cin.ignore(INT_MAX, '\n');
   cout << "Input 2: ";
   cin.get(buf2, sizeof(buf2));
}
```

Функция **getline**

```
istream& istream::getline(char*, int, char);
```

Извлекает из потока строку подобно **get(char*, int, char)**. В отличие от функции **get**, эта функция извлекает из потока завершающий символ, но не помещает его в буфер.

Функция **ignore**

```
istream& istream::ignore(int n = 1, int delim = EOF);
```

При вводе извлекает из потока не более **n** символов (по умолчанию 1). Если при извлечении встретится ограничитель (второй параметр), он также извлекается и операция завершается. Функция используется для пропуска некоторого числа символов.

Функция **gcount**

```
int istream::gcount(void);
```

Возвращает число символов, извлеченных последней функцией неформатированного ввода (при помощи **get**, **getline**, **read**). Некоторые функции форматированного ввода используют функции неформатированного ввода, поэтому это число следует использовать осторожно.

Функция **peek**

```
int istream::peek(void);
```

Возвращает значение очередного символа, не извлекая его из потока (заглядывает вперед). Если флаги состояния потока имеют ненулевое значение, возвращается **EOF**.

Функция **putback**

```
istream& istream::putback(char);
```

Возвращает во входной поток последний извлеченный символ. Символ можно заменить.

Функция **seekg**

```
istream& istream::seekg(long);
```

Устанавливает указатель извлечения входного потока на указанную абсолютную позицию. Другой вариант этой функции устанавливает позицию относительно текущей в указанном направлении:

```
istream& istream::seekg(long, ios::seek_dir);
```

Направление задается при помощи перечисления **seek_dir**:

```
enum seek_dir { beg, cur, end };
```

- если направление равно **ios::cur**, смещение отсчитывается от текущей позиции;
- если направление равно **ios::beg**, смещение отсчитывается от начала;
- если направление равно **ios::end**, смещение отсчитывается от конца.

Функция **tellg**

```
long istream::tellg(void);
```

Возвращает текущую абсолютную позицию входного потока.

Функции вывода (поток ostream)

Функция **write**

```
ostream& ostream::write(const signed char*, int);
ostream& ostream::write(const unsigned char*, int);
```

Помещает в поток указанное число символов из буфера. Пример:

```
#include <iostream.h>
char buf[] = "abcdefgh";
void main(void)
{
   cout.write(buf, 5);
}
```

Функция **put**

```
ostream& ostream::put(char);
```

Помещает в выходной поток один символ.

Функция **seekp**

```
ostream& ostream::seekp(long);
ostream& ostream::seekp(long, seek_dir);
```

Устанавливает позицию в выходном потоке. Первый вариант устанавливает абсолютную позицию, второй — относительную в соответствии с заданным направлением.

Функция **tellp**

```
long ostream::tellp();
```

Возвращает абсолютную позицию в выходном потоке.

Функция **flush**

```
ostream& ostream::flush();
```

Заставляет записать данные из буфера выходного потока на диск.

Перенаправление ввода и вывода

Имена **cin** и **cout** можно назначить потокам пользователя. Например, перенаправить вывод в файл:

```
#include <fstream.h>

ofstream ofs;
void main(void)
{
   ofs.open("c:\\file.txt");
   if (ofs) cout = ofs;
   cout << "Sample text" << endl;
   ofs.close;
}
```

О файловом вводе и выводе см. «Файловый ввод/вывод».

Файловый ввод/вывод

Для файлового ввода/вывода библиотека Си++ предлагает три класса:

- **ifstream** (производный от **istream**) — для операций чтения файла;
- **ofstream** (производный от **ostream**) — для операций записи в файл;
- **fstream** (производный от **iostream**) — для операций чтения и записи.

Являясь производными от классов потоков, эти классы наследуют все функциональные свойства, описанные ранее, например, операции извлечения и помещения, флаги, манипуляторы и т. п.

Открытие файла

Открыть файл можно при помощи конструктора потока или при помощи метода **open**. Для каждого из трех классов файловых потоков предусмотрено четыре конструктора. Конструктор без параметров создает объект без открытия файла, например:

```
ifstream ifs;
```

Для открытия файла с этим потоком нужно использовать метод **open**:

```
ifs.open(имя_файла);
```

В качестве имени файла используется знаковый массив или строковый литерал.

Следующие конструкторы позволяют создать поток и открыть файл:

```
ifstream(const char *name, int omode=ois::in, int prot=filebuf::openprot);
ofstream(const char *name, int omode=ois::out, int prot=filebuf::openprot);
fstream(const char *name, int omode, int prot = filebuf::openprot);
```

Пример открытия файла для записи:

```
ofstream ofs("c:\\file.txt");
```

Файл для чтения открывается аналогично, только используется другой класс потока (**ifstream**). Дополнительно об открытии файла см. «Режимы доступа к файлу».

Для создания нового потока и прикрепления его к файлу, открытому при помощи дескриптора, используется конструктор с одним параметром типа **int**:

```
ifstream::ifstream(int);
ofstream::ofstream(int);
fstream::fstream(int);
```

Например:

```
#include <io.h>
#include <fcntl.h>
#include <fstream.h>
void main(void)
{
   int handle = open("TEST.TXT", O_CREAT | O_TEXT);
   ofstream ofs(handle);
   ofs << "Hello world" << endl;
   close(handle);
}
```

Четвертый конструктор позволяет создать поток, связать его с уже открытым файлом и буфером:

```
ifstream::ifstream(int f, char *b, int len);
ofstream::ofstream(int f, char *b, int len);
fstream::fstream(int f, char *b, int len);
```

Пример использования:

```
#include <io.h>
#include <fcntl.h>
#include <fstream.h>
char buf[128];
void main(void)
{
   int handle = open("TEST1.TXT", O_CREAT | O_TEXT);
   ofstream ofs(handle, buf, sizeof(buf));
   ofs << "Hello world" << endl;
   close(handle);
}
```

Режимы доступа к файлу

Второй параметр функции открытия файла **open** указывает режим доступа, который определяется, исходя из установленных битов режима. Битовые маски заданы перечислением **ios::open_mode**:

Бит	*Маска*	*Описание*
0	**in** (=1)	открыт для чтения
1	**out** (=2)	открыт для записи
2	**ate** (=4)	после открытия указатель позиции в конце файла
3	**app** (=8)	режим добавления к концу файла
4	**trunc** (=16)	существующий файл удалить (очистить)
5	**nocreate** (=32)	не открывать, если файл не существует
6	**noreplace** (=64)	не открывать, если файл существует
7	**binary** (=128)	двоичный (не текстовый) файл

Следующий пример открывает новый файл для вывода, только если он уже не существует, записывает в него строку и закрывает. Затем открывает полученный файл с указателем в конце файла, записывает в него еще одну строку и закрывает. Далее файл открывается для чтения и посимвольно выводится на стандартный вывод:

```
#include <fstream.h>
char name[] = "TEST2.txt";
void main(void)
{
   ofstream ofs;
   ofs.open(name, ios::out | ios::noreplace);
   if (!ofs) {
      cout << "Open error." << endl;
   }
   else {
      ofs << "Line 1\n";
      ofs.close();
      fstream fs;
      fs.open(name, ios::out | ios::ate);
      if (!fs) {
         cout << "Open error.\n";
      }
      else {
         fs << "Line 2\n";
         fs.close();
         ifstream ifs;
         ifs.open(name);
         if (ifs) {
            while (!ifs.eof()) {
               char c;
               ifs.get(c);
               cout << c;
            }
            ifs.close();
         }
      }
   }
}
```

Режим открытия файла можно указывать также и при использовании конструктора второго типа.

Закрытие файла

Производится при помощи метода `close`. Этот метод вызывает появление файла в файловой системе, если файл был открыт для записи и разрешает доступ к нему. См. предыдущий пример.

Неформатируемый ввод/вывод данных

Каждый тип данных имеет минимум два вида представления. Первое представление, — внутренний формат, который используется процессором. Второе представление, — строковое (текстовое), которое используется для отображения информации в форме, понятной пользователю. Для примера рассмотрим число `12345`. Текстовое представление этого числа — запись в виде последовательности из пяти символов `1`, `2`, `3`, `4` и `5`. Внутренний формат этого числа как целого

типа **int** — два байта. Младший байт внутреннего представления равен **57**, старший байт равен **48**. Это следует из представления этого числа в двоичной форме: **0011|0000|0011|1001**. Вертикальными чертами выделены тетрады бит. Первые две тетрады образуют число $30_{16} = 48_{10}$ (старший байт), две вторые тетрады образуют число $39_{16} = 57_{10}$ (младший байт).

Операции помещения и извлечения перегружены для всех основных типов данных и осуществляют форматированный ввод/вывод данных. Эти операции производят форматирование информации в соответствии с ее типом. Многие функций потоков, однако, не производят никаких преобразований. К ним относятся функции **read**, **write**, **get**, **put**, **getline**. Функции неформатированного ввода/вывода помещают в поток или извлекают из него некоторое количество байт. Интерпретация такой информации возлагается на программиста. Операции форматированного ввода/вывода предназначены для работы с текстовыми файлами. Неформатирующие операции предназначены для работы с двоичными файлами, то есть с файлами, информация в которых записывается во внутреннем формате данных.

Двоичный файл нельзя «посмотреть», так как составляющие его байты не образуют текст. При записи в текстовый файл числа **12345** файл можно открыть при помощи, например, приложения **Блокнот**, и увидеть число **12345**. При записи этого же этого числа в двоичный файл (неформатированно) будет записано **2** байта. Если открыть этот файл при помощи **Блокнота**, мы увидим **90**, что соответствует байтам **57** и **48** (см. таблицу **ASCII**).

В следующем примере открывается двоичный файл и в него записывается информация при помощи функции неформатированного вывода **write**. В качестве информации выступает число **12345**, которое представлено двумя внутренними форматами — в виде целого **int** и в виде вещественного **double**:

```
#include <fstream.h>
void main(void)
{
   int i = 12345;
   double d = 12345.0;
   ofstream fs("bin.txt", ios::binary);
   if (!fs) return;
   fs.write((char*)&i, sizeof(i));
   fs.write((char*)&d, sizeof(d));
   fs.close();
}
```

Результирующий файл содержит 10 байт — 2 байта для целого представления и 8 байт для представления формата с плавающей запятой. Для чтения этого файла используется функция неформатированного ввода **read**. Файл открывается как двоичный:

```
#include <fstream.h>
void main(void)
{
   int i;
   double d;
   ifstream fs("bin.txt", ios::binary);
   if (!fs) return;
   fs.read((char*)&i, sizeof(i));
   fs.read((char*)&d, sizeof(d));
```

```
   fs.close();
}
```

Фактически при записи внутренний формат числа побайтно копируется в файл, а при чтении наоборот — байты из файла копируются во внутренний формат. Внутренний формат выступает в качестве буфера для хранения информации. Конструкция типа `(char*)&i` указывает адрес внутреннего представления в виде последовательности байт.

Форматирование в памяти

В библиотеке потоков Си++ предусмотрены операции ввода/вывода над данными в памяти (резидентными потоками). Для этой цели определено три класса:

istrstream, **ostrstream** и **strstream**.

Класс **istrstream** обеспечивает извлечение форматированных данных из памяти. При создании объекта этого класса нужно указать буфер и его размер. Если буфер заканчивается нулевым символом, указание размера не обязательно. В следующем примере буфер содержит форматированные элементы данных трех типов (целое, строковое и с плавающей точкой), разделенные пробелами. Извлечение из этого потока производится в переменные соответствующих типов:

```
#include <strstrea.h>
char b[] = "1 January 12.34";
void main(void)
{
   istrstream s(b);
   int i;
   char a[10];
   double d;
   s >> i >> a >> d;
}
```

Класс **ostrstream** обеспечивает помещение в память форматированных данных. Конструктор по умолчанию этого класса создает поток, буфер которого изменяется динамически по мере помещения в него новых данных. В следующем примере создается поток с динамическим буфером и в него накапливаются данные разных типов. Для «замораживания» буфера применяется функция **str**, которая возвращает указатель на буфер. После «замораживания» изменение размера буфера невозможно (вызывает непредсказуемые последствия). Для очистки и удаления буфера его нужно вернуть потоку, разморозив при помощи метода **freeze**. Перед фиксацией буфера его нужно дополнить нулевым байтом:

```
#include <strstrea.h>

void main(void)
{
   int i = 2;
   char a[] = " February ";
   double d = 5.6;
   ostrstream os;
   os << i << a << d << ends;
   cout << os.str() << endl;
   os.rdbuf()->freeze(0);
}
```

Конструктор с параметрами создает резидентный поток с заданным буфером. Следующий пример поясняет использование потока с фиксированным буфером:

```
#include <strstrea.h>
char b[20];
void main(void)
{
   ostrstream os(b, sizeof(b));
   os << 5 << " May " << 9.09 << ends;
   cout << b << endl;
}
```

Иерархия классов потоков

Иерархия основных классов библиотеки **IOStream**:

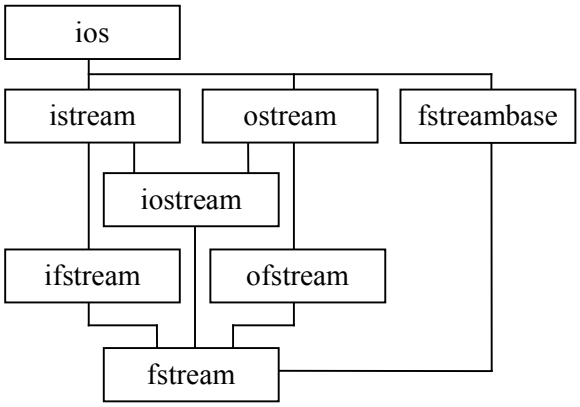

Рис. 2. Иерархия классов библиотеки потоков (частично)

Библиотека содержит также большое количество других классов. Описание этих классов не входит в задачи настоящего пособия и рекомендуется для самостоятельного изучения.

Управление исключениями

Во время выполнения программы часто возникают непредвиденные, необычные состояния и события, которые не позволяют продолжить нормальное исполнение основного кода программы. Примерами таких состояний могут быть деление на ноль или отсутствие файла, предназначенного для обработки. Проблема заключается в том, что в момент программирования предусмотреть все возможные состояния программы невозможно. Управление исключениями предназначено для выявления исключительных ситуаций и изменения нормального потока управления функции на ту часть кода, которая готова принять контроль над возникшей конкретной ситуацией.

Управление исключениями Си++ является частью стандарта ANSI Си++. Этот стандарт поддерживает *окончательную* модель управления: после того, как исключение зафиксировано, управление передается *специальной* секции кода текущей функции, а исполнение основного кода функции *заканчивается*.

Для управления исключениями используются три новых ключевых слова: **try** (попробуй), **catch** (захвати) и **throw** (выброси). Блок **try {}** предназначен для ограничения той части кода, исполнение которой *предположительно* может привести к исключению. В случае возникновения непредвиденной ситуации внутри этого блока происходит *прекращение* его исполнения и *выброс* исключения. Выброс исключения вызывает переход управления на часть кода, заключенную в тот или иной блок **catch {}**.

Рассмотрим простейший пример управления исключениями. Он состоит из одного блока **try** и одного блока **catch**. Блок **catch** должен следовать непосредственно за блоком **try**. Блок **catch** предназначен для обработки исключения *определенного* типа, поэтому за ключевым словом **catch** следуют круглые скобки, внутри которых описывается тип обрабатываемого данным блоком исключения и, возможно, специфицирктся переменная. Многоточие в качестве параметра блока **catch** в приведенном ниже примере обозначет «прочие исключения»:

```
#include <iostream.h>

int main(void)
{
   try {
      int i = 0;
      i = i / i;
      return 0;
   }
   catch(...) {
      cout << "Error ... ";
      return 1;
   }
}
```

В **try** блоке искусственно генерируется исключительная ситуация «деление на ноль». Так как эта ситуация исключает продолжение выполнения этой секции кода, управления передается в секцию **catch**, которая в примере является единственной и предназначенной для обработки всех необработанных исключений.

Внутри обработчика исключения на дисплей выводится сообщение об ошибке и выполнение программы завершается.

Блоку **try** может соответствовать несколько обработчиков исключений (несколько блоков **catch**). Выбор обработчика определяется типом, который выбрасывается в блоке **try**. В следующем абстрактном примере показано, как происходит переход в тот или иной обработчик:

```
#include <iostream.h>

int main(void)
{
   try {
      int i;
      char m[] = "string exception";
      cout << "1 - Integer Exception" << endl;
      cout << "2 - String Exception" << endl;
      cin >> i;
      if (i == 1) throw i;
      if (i == 2) throw m;
   }
   catch(int) {
      cout << "Error integer";
      return 1;
   }
   catch(char* msg) {
      cout << "Error " << msg;
      return 2;
   }
   return 0;
}
```

Для того, чтобы «выбросить» определенный тип, в блоке **try** используется оператор **throw** с параметром. *Тип параметра определяет тип выбрасываемого исключения* и, соответственно, тип вызываемого обработчика **catch**. Местоположение оператора **throw** называют *точкой выброса*. Если после блока **try** нет подходящего блока **catch**, управление передается обработчику прочих ошибок **catch(...)**, который должен быть последним среди блоков **catch**. Если обработчик прочих ошибок отсутствует, исключение вызывает прекращение работы программы.

Оператор **throw** позволяет выбрасывать не только встроенные, но и определенные пользователем типы и классы. Это значительно расширяет возможности управления исключениями, так как при возникновении ошибки в обработчик может быть передана дополнительная информация, например, код ошибки, сообщение, источник, справочная информация и т. п. В следующем примере в качестве выбрасываемого типа используется класс **errmsg**:

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

struct errmsg {
   char msg[81];
   errmsg ( const char *s = "" ) { strcpy( msg, s ); }
   operator char* (void) { return msg; }
};
```

Для генерирования исключения в примере используется функция **func**, которая выбрасывает исключение с переменной типа **errmsg**:

```
void func(void)
{
   errmsg err("Exception");
   throw err;
}
```

В основной функции **func** вызывается в блоке **try** с последующим обработчиком типа **errmsg**:

```
int main(void)
{
   try {
      func();
   }
   catch(errmsg m) {
      cout << (char*)m;
      return 1;
   }
   return 0;
}
```

Данный пример демонстрирует также косвенный выброс исключения: исключение выбрасывается не только непосредственно блоком **try**, но и любой секцией кода, которая явно или неявно вызывается из этого блока.

В следующем примере показано, как влияет последовательность обработчиков исключений на перехват выброса. Блок **try** выбрасывает указатель на символ. После блока **try** первым стоит блок **catch** с типом **void***, а следующим — блок **catch** с типом **char***. Так как выбрасываемый тип сравнивается с типами блоков **catch** последовательно в порядке объявления, обработчик типа **char*** не вызывается:

```
#include <iostream.h>
int main(void)
{
   try {
      char a[] = "Char * exception";
      throw a;
   }
   catch(void *) {
      cout << "Void * exception";
      return 1;
   }
   catch(char *msg) {
      cout << msg;
      return 2;
   }
   return 0;
}
```

Применение оператора **throw** без операнда выбрасывает то исключение, которое обрабатывается. Это означает, что такая форма оператора может быть использована только внутри обработчика исключения или в функции, которая явно или неявно им вызывается.

Спецификация исключений функции

Спецификация исключений указывает типы исключений, которые прямо или косвенно может выбросить данная функция. Спецификация исключений присоединяется к заголовку функции и состоит из ключевого слова **throw** и скобок, внутри которых перечисляются все типы, которые предположительно могут быть выброшены функцией. Если спецификация типов отсутствует, функция описывается как не выбрасывающая никаких исключений. Если спецификация исключений отсутствует, предполагается, что функция может выбрасывать исключения любых типов. Далее приведен пример функции, которая предположительно выбрасывает исключения типов **long** и **char***:

```
void func(void) throw(long, char*)
{
   /* тело функции */
}
```

Спецификация исключений ни к чему не обязывает. Функция может выбросить исключение, которое не было заявлено в ее спецификации исключений. Непредвиденные исключения вызывают функцию **unexpected**, которая вызывает **terminate**.

Подробнее об управлении исключениями а также о структурированных исключениях см. [2].

Литература

1. Страуструп Б. Язык программирования Си++: Пер. с англ. — М.: Радио и связь, 1991. — 352 с.: ил.
2. Бруно Бабэ. Просто и ясно о Borland C++: Пер. с англ. — М.: БИНОМ, 1995. — 400 с.: ил.
3. Лукас П. C++ под рукой: Пер. с англ. — Киев: «ДиаСофт», 1993. — 176 с.: ил.
4. Стефан Дьюхарст, Кэти Сарк. Программирование на C++: Пер. с англ. — Киев: «ДиаСофт», 1993. — 272 с.: ил.