Глава 13

Пространства имён и другие темы.

В этой главе рассказывается о пространствах имён (namespaces), функциях преобразования (conversion functions), статических (static) и постоянных (const) членах классах, а также о других необычных инструментах C++.

13.1. Пространства имён.

О пространствах имён коротко было рассказано в главе 1, сейчас мы рассмотрим это понятие более подробно. Пространства имён появились в C++, относительно недавно. Они предназначены для локализации имён идентификаторов во избежание конфликтов имён (name collisions). В среде программирования C++ сосуществует невероятное количество имён переменных, функций и классов. До введения понятия пространств имён все эти имена находились в одном глобальном пространстве имен, и возникало множество конфликтов. Например, если вы в своей программе определяли функцию **toupper()**, то она могла бы (в зависимости от списка своих параметров) подменить стандартную библиотечную функцию **toupper()**, поскольку имена обеих функций хранились бы в одном глобальном пространстве имён. Конфликты имён имеют место, когда в одной и той же программе используются библиотеки функций и классов разных производителей. В этом случае вполне возможно – и даже очень вероятно, - что имена, определённые в одной библиотеке, будут конфликтовать с теми же именами, но определёнными в другой библиотеке.

Все проблемы решило введение понятия пространств имён и ключевого слова **namespace**. Это ключевое слово позволяет локализовать область видимости имён, объявленных в данном пространстве имён. Пространство имён даёт возможность использовать одно и то же имя в разных контекстах и при этом почвы для конфликтов не возникает. Вероятно, больше выиграла от введения пространств имён стандартная библиотека C++ (C++ standard library). В ранних версиях языка вся библиотека C++ определялась в глобальном пространстве имён. Теперь библиотека C++ определяется в собственном пространстве имён **std**, что значительно снижает вероятность конфликтов имён. Вы также можете создавать в программе свои собственные пространства имён, чтобы локализовать область видимости тех имён, которые, по вашему мнению, могли бы вызвать конфликты. Это особенно важно при создании собственных библиотек классов или функций.

Ключевое слово **namespace** путём добавления именованных областей даёт возможность разделить глобальное пространство имён. По существу, пространство имён определяет область видимости. Ниже приведена основная форма использования ключевого слова **namespace**:

```
namespace имя {
        // объявления
}
```

Всё, что определено внутри инструкции **namespace**, находится внутри области видимости данного пространства имён.

Ниже приведен пример объявления пространства имён **MyNameSpace**:

```
namespace MyNameSpace {
    int i,k;
    void myfunc (int j) { cout << j; }

    class myclass {
    public:
        void seti (int x) { i = x; }
        int geti() { return i; }
    };
}
```

Здесь имена переменных **i** и **k**, функции **myfunc()**, а также класса **myclass** находятся в области видимости, определённой пространством имён **MyNameSpace**.

К идентификаторам, объявленном в пространстве имён, внутри этого пространства можно обращаться напрямую. Например, в пространстве имён **MyNameSpace** в инструкции **return i** переменная **i** указана явно. Однако поскольку ключевое слово **namespace** определяет некоторую область видимости, то при обращении извне пространства имён к объектам, объявленным внутри этого пространства, следует указывать оператор расширения области видимости. Например, чтобы присвоить значение 10 переменной **i** в той части программы, которая не входит в пространство имён **MyNameSpace**, необходимо использовать следующую инструкцию:

```
MyNameSpace::i = 10;
```

А чтобы объявить объект типа **myclass** в той части программы, которая не входит в пространство имён **MyNameSpace**, нужна такая инструкция:

```
MyNameSpace::myclass ob;
```

Таким образом, для доступа к члену пространства имён извне этого пространства перед именем члена следует указать имя пространства имён с оператором расширения области видимости.

Можно себе вообразить, что если в вашей программе обращения к членам пространства имён происходят достаточно часто, необходимость каждый раз указывать имя пространства имён и оператор расширения видимости может быстро надоесть. Для решения этой проблемы была разработана инструкция **using**. У этой инструкции имеются две основные формы:

using namespace *имя;*
using *имя::член;*

В первой форме параметр ***имя*** задаёт имя пространства имён, доступ к которому вы хотите получить. При использовании этой формы инструкции **using** все члены, определённые в указанном пространстве имён, становятся доступными в текущем пространстве имён и с ними можно работать напрямую, без необходимости каждый раз указывать имя пространства и оператор расширения области видимости. При использовании второй формы инструкции **using** видимым делается только указанный в инструкции член пространства имён. Например, пусть у нас имеется описанное выше пространство имён **MyNameSpace**, тогда правильны все представленные ниже инструкции **using** и операторы присваивания:

```
using MyNameSpace::k;    // видимой делается только переменная k
k = 10;   // инструкция правильна, поскольку переменная k видима

using namespace MyNameSpace;    // видимыми делаются все члены
                                // пространства имён MyNameSpace
i = 10;    // инструкция правильна, поскольку видимыми все члены
           // пространства имён MyNameSpace
```

Имеется возможность объявить более одного пространства имён с одним и тем же именем. Это позволяет разделить пространство имён на несколько файлов или даже разделить пространство имён внутри одного файла. Рассмотрим следующий пример:

```
namespace NS {
        int i;
}
// …

namespace NS {
        int j;
}
```

Здесь пространство имён **NS** разделено на две части. Несмотря на это, содержимое каждой части по-прежнему остаётся в одном и том же пространстве имён – пространстве **NS**.

Пространство имён должно объявляться вне всех остальных областей видимости за исключением того, что одно пространство имён может быть вложено в другое. То есть вложенным пространство имён может быть только в другое пространство имён, но не в какую бы то ни было иную область видимости. Это означает, что нельзя объявлять пространства имён, например, внутри функции.

Имеется пространство имён особого типа – *безымянное пространство имён (unnamed space).* Безымянное пространство имён позволяет создавать идентификаторы, являющиеся уникальными внутри некоторого файла. Ниже представлена основная форма безымянного пространства имён:

```
namespace {
        // объявления
}
```

Безымянные пространства имён дают возможность задавать уникальные идентификаторы, известные только внутри области видимости одного файла. Таким образом, внутри файла, содержащего безымянное пространство имён, к членам этого пространства можно обращаться напрямую, без необходимости каких бы то ни было уточнений. Но вне этого файла такие идентификаторы неизвестны.

Вам вряд ли понадобится создавать свои пространства имён для небольших и средних по объёму программ. Однако, если вы собираетесь создавать библиотеки функций или классов, предназначенных для многократного использования, или хотите гарантировать своим программам широкую переносимость, вам следует рассмотреть возможность размещения своих кодов внутри некоторого пространства имён.

Примеры

1. В представленном ниже примере иллюстрируются атрибуты демонстрационных пространств имён.

```
// Демонстрационные пространства имен
#include <iostream>
using namespace std;

// определение первого пространства имен
namespace firstNS {
   class demo (
      int i;
   public:
      demo(int x) { i = x; }
      void seti(int x) { i = x; }
      int geti() { return i; }
   }

   char str[] = "Иллюстрация пространств имен\n";
   int counter;
}

// определение второго пространства имен
namespace secondNS {
   int x, y;
}

int main()
{
   // расширяем область видимости
   firstNS::demo ob(10);
   /* После объявления объекта ob, его функции-члены могут использоваться прямо, без какого бы то ни было
уточнения пространства имен
*/
   cout << "Значение объекта ob равно: " << ob.geti();
   cout << endl;
   ob.seti(99);
   cout << "Теперь значение объекта ob равно: " << ob.geti();
   cout << endl;

   // вводим строку str в текущую область видимости
   using firstNS::str;
   cout << str;

   // вводим все пространство имен firstNS
   // в текущую область видимости
   using namespace firstNS;
   for(counter=10; counter; counter--)
      cout << counter << " ";
   cout << endl;

   // используем пространство имен secondNS
   secondNS::x = 10;
   secondNS::y = 20;

   cout << "Переменные x, y равны: " << secondNS::x;
   cout << ", " << secondNS::y << endl;

   // вводим все пространство имен secondNS
   // в текущую область видимости
   using namespace secondNS;
   demo xob(x), yob(y);
```

```
    cout << "Значения объектов xob, yob равны: " << xob.geti();
    cout << ", " << yob.geti() << endl;

    return 0;
}
```

После выполнения программы на экран будет выведено следующее:

```
Значение объекта ob равно: 10
Теперь значение объекта ob равно: 99
Иллюстрация пространств имён
10 9 8 7 6 5 4 3 2 1
Переменные x, y равны: 10, 20
Значения объектов xob, yob равны: 10, 20
```

Программа иллюстрирует одно важное положение: при совместном использовании нескольких пространств имён одно пространство не подменяет другое. Когда вы вводите некоторое пространство имён в текущую область видимости, его имена просто добавляются в эту область, независимо от того, находятся ли в ней в это время имена из других пространств имён. Таким образом, ко времени завершения программы в глобальное пространство имён был добавлены пространства имён **std, firstNS** и **secondNS**.

2. Как уже упоминалось, пространство имён можно разделить либо между файлами, либо внутри одного файла, тогда содержимое этого пространства имён объединяется. Рассмотрим пример объединения разделённого пространства имён.

```
// Объединение пространства имен
#include <iostream>
using namespace std;

namespace Demo {
    int a; // объявление переменной a в пространстве имен Demo
}
    int x; // объявление переменной x в глобальном пространстве имен

namespace Demo {
    int b; // объявление переменной b в пространстве имен Demo
}
int main()
{
    using namespace Demo;
    a = b = x = 100;
    cout << a << " " << b << " " << x;
    return 0;
}
```

В данном примере обе переменные, **a** и **b**, оказались в одном пространстве имён – пространстве имён **Demo**, а переменная **x** оказалась в глобальном пространстве имён.

3. Как уже упоминалось, стандарт Standard C++ определяет целую библиотеку в собственном пространстве имён **std**. Именно по этой причине во всех программах этой книги имеется следующая инструкция:

```
using namespace std;
```

Эта инструкция делает пространство имён **std** текущим, что позволяет получить прямой доступ к именам функций и классов, определённых в библиотеке языка Standard C++, без

необходимости каждый раз с помощью оператора расширения области видимости уточнять, что используется пространство имён **std**.

Тем не менее, если пожелаете, может перед каждым идентификатором ставить имя пространства имён **std** и оператор расширения области видимости – ошибки не будет. Например, в следующей программе библиотека языка Standard C++ не введена в глобальную область видимости.

```
// Явное задание используемого пространства имен
#include <iostream>

int main()
{
   double val;

   std::cout << "Введите число: ";

   std::cin >> val;

   std::cout << "Вот ваше число: ";
   std::cout << val;

   return 0;
}
```

Как показано в данной программе, чтобы воспользоваться стандартными потоками ввода и вывода **cin** и **cout**, перед именами этих потоков необходимо явно указывать их пространство имён.

Если в вашей программе не предусмотрено широкое использование библиотеки языка Standard C++, вы можете не вводить пространство имён **std** в глобальную область видимости. Однако, если в вашей программе содержатся тысячи ссылок на стандартные библиотечные имена, включить в программу идентификатор **std** гораздо проще, чем добавлять его чуть ли не к каждой инструкции.

4. Если в своей программе вы используете только несколько имён из стандартной библиотеки, может оказаться удобнее с помощью инструкции **using** отдельно задать эти несколько имён. Преимущество такого подхода в том, что указанные имена можно вносить в программу без необходимости их уточнения, и в то же время не нужно вводить всю стандартную библиотеку в глобальное пространство имён. Рассмотрим пример:

```
// Введение в глобальное пространство только нескольких имен
#include <iostream>

// обеспечение доступа к потокам cin и cout
using std::cout;
using std::cin;

int main()
{
   double val;
   cout << "Введите число: ";
   cin >> val;
   cout << "Вот ваше число: ";
   cout << val;
   return 0;
}
```

Здесь стандартными потоками ввода и вывода **cin** и **cout** можно пользоваться напрямую, но в то же время остальные имена из пространства имён **std** оставлены вне текущей области видимости.

5. Ранее уже говорилось, что библиотека исходного языка C++ была определена в глобальном пространстве имён. Если вам придётся модернизировать старую программу на C++, то вам понадобится либо включить в неё инструкцию **using namespace std**, либо перед каждой ссылкой на член библиотеки дописывать имя пространства имён с оператором расширения области видимости **std::**. Это особенно важно, если вы замените прежние заголовочные файлы заголовками нового стиля (без расширения **.h**). Помните, прежние заголовочные файлы размещают своё содержимое в глобальном пространстве имён, а заголовки нового стиля – в пространстве имён **std**.
6. В языке C, если вы хотите ограничить область видимости глобального имени только тем файлом, в котором это имя добавлено, вы должны объявить его как статическое, т.е. с идентификатором **static**. Например, предположим, что следующие два файла являются частью одной программы:

Первый файл

```
Static int counter;
void f1() {
counter = 99; // OK
}
```

Второй файл

```
extern int counter;
void f2() {
counter = 10; // Ошибка
}
```

Поскольку переменная **counter** определена в первом файле, т и использовать её можно в первом файле. Во втором файле, несмотря на указание в инструкции с переменной **counter** идентификатора **extern**, попытка использования этой переменной ведёт к ошибке. Объявляя в первом файле переменную **counter** статической, мы ограничиваем её область видимости этим файлом.

Хотя объявление глобальных переменных с идентификатором **static** в C++ по-прежнему доступны, для достижения той же цели здесь имеется лучший путь – использовать, как показано в следующем примере, безымянное пространство имён.

Первый файл

```
namespace {
  int counter;
}
void f1() {
counter = 99; // OK
}
```

Второй файл

```
extern int counter;
void f2() {
counter = 10; // Ошибка
}
```

13.2. Функции преобразования.

Иногда бывает полезно преобразовать объект одного типа в объект другого типа. Хотя для этого можно воспользоваться перегруженной оператор-функцией, часто более лёгким (и лучшим) способом такого преобразования является функция преобразования. *Функция преобразования (conversion function)* преобразует объект в значение, совместимое с другим типом данных, который часто является одним из встроенных типов данных C++. Уточним, функция преобразования автоматически преобразует объект в значение, совместимое с типом выражения, в котором этот объект используется.

Здесь показана основная форма функции преобразования:

operator *тип()* { return *значение;*}

Здесь ***тип*** – это целевой тип преобразования, а ***значение*** – это значение объекта после выполнения преобразования. Функции преобразования возвращают значение типа ***тип***. У функции преобразования не должно быть параметров, и она должна быть членом класса, для которого выполняется преобразование.

Как будет показано в примерах, обычно функция преобразования обеспечивает более ясный подход к преобразованию значения объекта в другой тип, чем любой другой метод C++, поскольку она позволяет явно включить объект в выражение, содержащее переменные целевого типа.

Примеры

1. В следующей программе класс **coord** содержит функцию преобразования, которая преобразует объект в целое. В данном случае функция возвращает произведение двух координат, однако в каждом конкретном случае допускается любое необходимое действие:

```
// Простой пример функции преобразования
#include <iostream>
using namespace std;

class coord {
   int x, y;
public:
   coord(int i, int j) { x = i; y = j; }
   operator int() { return x*y; } // функция преобразования
};

int main()
{
   coord o1(2, 3), o2(4, 3);
   int i;
   i = o1; // объект o1 автоматически преобразуется в целое
   cout << i << '\n';
   i = 100 + o2; // объект o2 преобразуется в целое
   cout << i << '\n';
   return 0;
}
```

В результате выполнения этой программы на экран будут выведены значения 6 и 112.

Обратите внимание, что функция преобразования вызывается тогда, когда объект **o1** присваивается целому, а также когда объект **o2** используется, как часть большого выражения, оперирующего с целыми. Как уже установлено, с помощью функции преобразования вы разрешаете вставлять объекты созданных вами классов в «обычные» выражения, без образования сложных цепочек перегруженных оператор-функций.

2. Следующим представлен другой пример функции преобразования. Здесь преобразуется строка типа **strtype** в символьный указатель **str**.

```
#include <iostream>
#include <cstring>
```

```
using namespace std;

class strtype {
   char str[80];
   int len;
public:
   strtype(char *s) { strcpy(str, s); len = strlen(s); }
   operator char *() { return str; } // преобразование в тип char *
};

int main()
{
   strtype s("Это проверка\n");
   char *p, s2[80];
   p = s; // преобразование в тип char *
   cout << "Это строка: " << p << '\n';
   // при вызове функции преобразуется в тип char *
   strcpy(s2, s);
   cout << "Это копия строки: " << s2 << '\n';
   return 0;
}
```

Эта программа выводит на экран следующее:

```
Это строка: Это проверка
Это копия строки: Это проверка
```

Как можно заметить, функция преобразования вызывается не только при присваивании объекта **s** объекту **p** (который имеет тип **char ***), но она также используется как параметр для функции **strcpy()**. Вспомните, функция **strcpy()** имеет следующий прототип:

char *strcpy(char **s1*, const char **s2*);

Поскольку прототип определяет, что объект **s2** имеет тип **char ***, функция преобразования объекта в тип **char *** вызывается автоматически. Этот пример показывает, как функция преобразования может помочь интегрировать ваши классы в библиотеку стандартных функций C++.

13.3. Статические члены класса.

Переменные – члены класса можно объявлять как статические (**static**). Используя статические переменные-члены, можно решить несколько непростых проблем, Если вы объявляете переменную статической, то может существовать только одна копия этой переменной – независимо от того, сколько объектов данного класса создаётся. Каждый объект просто использует(совместно с другими) эту одну переменную. Запомните, для обычных переменных-членов при создании каждого объекта создаётся их новая копия, и доступ к каждой копии возможен только через этот объект. (Таким образом, для обычных переменных каждый объект обладает собственными копиями переменных.) С другой стороны, имеется только одна копия статической переменной – члена класса, и все объекты класса используют её совместно. Кроме этого, одна и та же статическая переменная будет использоваться всеми классами, производными от класс, в котором эта статическая переменная содержится.

Может показаться необычным то, что статическая переменная – член класса создаётся ещё *до* того, как создан объект этого класса. По сути, статический член класса – это просто глобальная переменная, область видимости которой ограничена классом, в котором она объявлена. В результате, как вы узнаете из следующих примеров, доступ к статической переменной-члену возможен без всякой связи с каким бы то ни было объектом.

Когда вы объявляете статические данные-члены внутри класса, вы не определяете их. Определить их вы должны где-нибудь вне класса. Для того, чтобы указать, к какому классу статическая переменная принадлежит, вам необходимо переобъявить (redeclare) её, используя операцию расширения области видимости.

Все статические переменные-члены по умолчанию инициализируются нулём. Однако, если это необходимо, статической переменной класса можно дать любое выбранное начальное значение.

Запомните, что основной смысл поддержки C++ статических переменных-членов состоит в том, что благодаря им отпадает необходимость в использовании глобальных переменных. Как можно предположить, если при работе с классами задавать глобальные переменные, то это почти всегда нарушает принцип инкапсуляции, являющийся фундаментальным принципом ООП и C++.

Помимо переменных-членов статическими можно объявлять и функции-члены, но обычно это не делается. Доступ к объявленной статической функции - члену классу возможен только для других статических членов этого класса. (Естественно, что доступ к объявленной статической функции – члену класса возможен также и для глобальных данных и функций.) У статической функции-члена нет указателя **this**. Статические функции-члены не могут быть виртуальными. Статические функции-члены не могут объявляться с идентификаторами **const** (постоянный) и **volatile** (переменный). И наконец, статические функции – члены класса могут вызываться любым объектом этого класса, а также через имя класса и оператор расширения области видимости без всякой связи с каким бы то ни было объектом этого класса.

Примеры

1. Простой пример использования статической переменной-члена.

```
// Пример статической переменной-члена
#include <iostream>
using namespace std;

class myclass {
   static int i;
public:
   void seti(int n) { i = n; }
   int geti() { return i; }
};
// Определение myclass::i. Переменная i по-прежнему
// остается закрытым членом класса myclass
int myclass::i;

int main()
{
   myclass o1, o2;
   o1.seti(10);
   cout << "o1.i: " << o1.geti() << '\n'; // выводит значение 10
```

```
   cout << "o2.i: " << o2.geti() << '\n'; // также выводит значение 10
   return 0;
}
```

После выполнения программы на экране появится следующее:

```
o1.i: 10
o2.i: 10
```

Глядя на программу, можно заметить, что фактически только для объекта **o1** устанавливается значение статической переменной-члена **i**. l поскольку переменная **i** совместно используется объектами **o1** и **o2** (и, на самом дел, всеми объектами типа **myclass**), оба вызова функции **geti()** дают один и тот же результат.

Обратите внимание, что переменная **i** объявляется внутри класса **myclass**, но определяется вне его. Этот второй шаг гарантирует, что для переменной **i** будет выделена память. С технической точки зрения, определение класса является всего лишь определением типа и не более. Память для статической переменной при этом не выделяется. Поэтому для выделения памяти статической переменной-члену требуется её явное определение.

2. Поскольку статическая переменная – член класса существует ещё до создания объекта этого класса, доступ к ней в программе может быть реализован без всякого объекта. Например, в следующем варианте предыдущей программы для статической переменной **i** устанавливается значение 100 без всякой ссылки на конкретный объект. Обратите внимание на использование оператора расширения области видимости для доступа к переменной **i**.

```
// Для обращения к статической переменной объект не нужен
#include <iostream>
using namespace std;

class myclass {
public:
   static int i;
   void seti(int n) { i = n; }
   int geti() { return i; }
};
int myclass::i;

int main()
{
   myclass o1, o2;
   // непосредственное задание значения переменной i
   myclass::i = 100; // объекты не упоминаются
   cout << "o1.i: " << o1.geti() << '\n'; // выводит 100
   cout << "o2.i: " << o2.geti() << '\n'; // также выводит 100
   return 0;
}
```

Так как статической переменной **i** присвоено значение 100, программа выводит на экран следующее:

```
o1.i: 100
o2.i: 100
```

3. Традиционным использованием статических переменных класса является координация доступа к разделяемым ресурсам, таким как дисковая память, принтер или сетевой сервер. Как вы уже, вероятно, знаете из своего опыта, координация доступа к

разделяемым ресурсам требует некоторых знаний о последовательности событий. Чтобы понять, как с помощью статических переменных-членов можно управлять доступом к разделяемым ресурсам, изучите следующую программу. В ней создаётся класс **output**, который поддерживает общий выходной буфер **outbuf()**. Эта функция посимвольно передает содержимое строки **str**. Сначала функция запрашивает доступ к буферу, а затем передаёт в него все символы **str**. Функция предотвращает доступ в буфер другим объектам до окончания вывода. Понять работу программы можно по комментариям к ней.

```
// Пример разделения ресурсов
#include <iostream>
#include <cstring>
using namespace std;

class output {
   static char outbuf[255]; // это разделяемые ресурсы
   static int inuse; // если переменная inuse равна 0,
                     // буфер доступен; иначе - нет
   static int oindex; // индекс буфера
   char str[80];
   int i; // индекс следующего символа строки
   int who; // идентификатор объекта, должен быть положительным
public:
   output(int w, char *s) {
      strcpy(str, s); i = 0; who = w;
   }

/* Эта функция возвращает -1 при ожидании буфера; 0 - при завершении вывода; who - если буфер все еще используется.
*/
   int putbuf()
   {
      if(!str[i]) { // вывод завершен
         inuse = 0; // освобождение буфера
         return 0; // признак завершения
      }
      if(!inuse) inuse = who; // захват буфера
      if(inuse != who) return -1; // буфер использует кто-то еще
      if(str[i]) { // символы еще остались
         outbuf[oindex] = str[i];
         i++; oindex++;
         outbuf[oindex] = '\0'; // последним всегда идет нуль
         return 1;
      }
      return 0;
   }
   void show() { cout << outbuf << '\n';}
};

char output::outbuf[255]; // это разделяемые ресурсы
int output::inuse = 0; // если переменная inuse равна 0,
                       // буфер доступен; иначе - нет
int output::oindex = 0; // индекс буфера

int main()
{
   output o1(1, "Это проверка "), o2(2, "статических переменных");
   while (o1.putbuf() | o2.putbuf()); // вывод символов
   o1.show();
   return 0;
}
```

4. Статические функции-члены применяются достаточно редко, но для предварительной (до создания реального объекта) инициализации закрытых статических данных-членов они могут оказаться очень удобными. Например, ниже представлена совершенно правильная программа.

```
#include <iostream>
using namespace std;

class static_func_demo {
   static int i;
public:
   static void init(int x) { i = x; }
   void show() { cout << i; }
};

int static_func_demo::i; // определение переменной i

int main()
{
   // инициализация статических данные еще до создания объекта
   static_func_demo::init(100);
   static_func_demo x;
   x.show(); // вывод на экран значения 100
   return 0;
}
```

Здесь значение функции **init()** инициализирует переменную **i** ещё до создания объекта типа **static_func_demo**.

13.4. Постоянные и модифицируемые члены класса.

Функции – члены класса могут объявляться постоянными (с идентификатором **const**). Если функция объявлена постоянной, она не может изменить вызывающий её объект. Кроме этого, постоянный объект не может вызвать непостоянную функцию-член. Тем не менее, постоянная функция-член может вызываться как постоянными, так и непостоянными объектами.

Для задания постоянной функции-члена используйте её форму, представленную в следующем примере:

```
class X {
  int some_var;
public:
  int f1() const; // постоянная функция-член
};
```

Обратите внимание, что ключевое слово **const** указывают следом за списком параметров функции, а не перед именем функции.

Возможна ситуация, когда вам понадобится, чтобы функция-член, оставаясь постоянной, тем не менее была способна изменить один или несколько членов класса. Это достигается заданием модифицируемых членов класса (ключевое слово **mutable**). Модифицируемый член класса можно изменить с помощью постоянной функции-члена.

Примеры

1. Функция-член объявляется постоянной, чтобы предотвратить возможность изменения вызвавшего её объекта. Для примера рассмотрим следующую программу:

```
/* Пример объявления постоянных функций-членов. Данная программа содержит ошибку и компилироваться не будет
*/
#include <iostream>
using namespace std;

class Demo {
   int i;
public:
   int geti() const {
      return i; // здесь все правильно
   }

   void seti(int x) const {
      i = x; // Ошибка!!!
   }
};

int main()
{
   Demo ob;
   ob.seti(1900);
   cout << ob.geti();
   return 0;
}
```

Данная программа не будет компилироваться, поскольку функция-член **seti()** объявлена постоянной, что означает невозможность изменения вызывающего её объекта. Таким образом, попытка изменения функцией переменной **i** ведёт к ошибке. С другой стороны, поскольку функция **geti()** не меняет переменной **i**, она совершенно правильна.

2. Чтобы допустить изменение избранных членов класса постоянной функцией-членом, они задаются модифицируемыми. Ниже представлен пример:

```
// Пример задания модифицируемого члена класса
#include <iostream>
using namespace std;

class Demo {
   mutable int i;
   int j;
public:
   int geti() const {
      return i; // здесь все правильно
   }

   void seti(int x) const {
      i = x; // теперь все правильно
   }

/* Если убрать комментарии вокруг этой функции, то программа компилироваться не будет
   void setj(int x) const {
      j = x; // здесь прежняя ошибка
```

```
  }
*/
};

int main()
{
  Demo ob;

  ob.seti(1900);
  cout << ob.geti();
  return 0;
}
```

Здесь переменная **i** задана модифицируемой, поэтому её может изменить функция-член **seti()**. Тем не менее, поскольку переменная **j** по-прежнему остаётся не модифицируемой, постоянная функция-член **seti()** не модет изменить её значение.

13.5. Заключительный обзор конструкторов.

Хотя тема конструкторов в этой книге уже обсуждалась, некоторые аспекты их применения остались нераскрытыми. Рассмотрим следующую программу:

```
#include <iostream>
#include <cstdlib>
using namespace std;

class myclass {
  int a;
public:
  myclass(int x) { a = x; }
  myclass(char *str) { a = atoi(str); }
  int geta() { return a; }
};

int main()
{
  // преобразование в вызов конструктора myclass ob(4)
  myclass ob1 = 4;
  // преобразование в вызов конструктора myclass ob("123")
  myclass ob1 = "123";
  cout << "ob1: " << ob1.geta() << endl;
  cout << "ob2: " << ob2.geta() << endl;
  return 0;
}
```

Здесь у конструктора класса **myclass** имеется один параметр. Обратите особое внимание на то, как в функции **main()** объявлен объект **ob**. Значение 4, заданное в скобках сразу за объектом **ob**, - это аргумент, который передаётся параметру **x** конструктора **myclass()** и с помощью которого инициализируется переменная **a**. Именно такая форма инициализации использовалась в примерах программ, начиная с первых глав этой книги. Однако это не единственный способ инициализации. Рассмотрим, к примеру, следующую инструкцию:

myclass ob = 4; // эта инструкция автоматически преобразуется в инструкцию myclass //ob(4);

Как показано в комментариях, эта форма инициализации автоматически преобразуется в вызов конструктора **myclass()** со значением 4 в качестве аргумента. Таким образом, предыдущая инструкция обрабатывается компилятором так, как будто на её месте находится инструкция:

myclass ob(4);

Как правило, всегда, когда у конструктора имеется только один аргумент, можно использовать любую из представленных выше двух форм инициализации объекта. Смысл второй формы инициализации в том, что для конструктора с одним аргументом она позволяет организовать неявное преобразование типа этого аргумента в тип класса, к которому относится конструктор.

Неявное преобразование можно запретить с помощью спецификатора **explicit** (явный). Спецификатор **explicit** применим только к конструкторам. Для конструкторов, заданных со спецификатором **explicit**, допустим только обычный синтаксис. Автоматического преобразования для таких конструкторов не выполняется. Например, если в предыдущем примере конструктор класса **myclass** объявить со спецификатором **explicit**, то для такого конструктора автоматического преобразования поддерживаться не будет. В представленном ниже классе конструктор **myclass()** объявлен со спецификатором **explicit**.

```
#include <iosrtream>
using namespace std;

class myclass {
  int a;
public:
  explicit myclass(int x) {a = x;}
  int geta() {return a;}
};
```

Для такого класса допустима только одна форма конструкторов:

```
myclass ob(4);
```

Примеры

1. В классе может более одного преобразующего конструктора. Например, рассмотрим следующую версию класса **myclass**.

```
#include <iostream>
#include <cstdlib>
using namespace std;

class myclass {
  int a;
public:
  myclass(int x) { a = x; }
  myclass(char *str) { a = atoi(str); }
  int geta() { return a; }
};

int main()
{
  // преобразование в вызов конструктора myclass ob(4)
  myclass ob1 = 4;
```

```
   // преобразование в вызов конструктора myclass ob("123")
   myclass ob1 = "123";

   cout << "ob1: " << ob1.geta() << endl;
   cout << "ob2: " << ob2.geta() << endl;
   return 0;
}
```

Поскольку типы аргументов обоих конструкторов различны (как это и должно быть) каждая инструкция инициализации автоматически преобразуется в соответствующий вызов конструктора.

2. Автоматическое преобразование на основе типа первого аргумента конструктора в вызов самого конструктора имеет интересное применение. Например, для класса **myclass** из примера 1, чтобы присвоить объектам **ob1** и **ob2** новые значения, функция **main()** выполняет преобразования из типа **int** в тип **char ***.

```
#include <iostream>
#include <cstdlib>
using namespace std;

class myclass {
   int a;
public:
   myclass(int x) { a = x; }
   myclass(char *str) { a = atoi(str); }
   int geta() { return a; }
};

int main()
{
   // преобразование в вызов конструктора myclass ob(4)
   myclass ob1 = 4;
   // преобразование в вызов конструктора myclass ob("123")
   myclass ob1 = "123";
   cout << "ob1: " << ob1.geta() << endl;
   cout << "ob2: " << ob2.geta() << endl;
   /* использование автоматического преобразования для присваивания новых значений
*/
   // преобразование в вызов конструктора myclass ob("1776")
   myclass ob1 = "1776";
   // преобразование в вызов конструктора myclass ob(2001)
   myclass ob1 = 2001;
   cout << "ob1: " << ob1.geta() << endl;
   cout << "ob2: " << ob2.geta() << endl;
   return 0;
}
```

3. Чтобы запретить показанные в предыдущих примерах преобразования, для конструкторов можно задать спецификатор **explicit**:

```
#include <iostream>
#include <cstdlib>
using namespace std;

class myclass {
   int a;
public:
   explicit myclass(int x) { a = x; }
```

```
    explicit myclass(char *str) { a = atoi(str); }
    int geta() { return a; }
};

int main()
{
    // преобразование в вызов конструктора myclass ob(4)
    myclass ob1 = 4;

    // преобразование в вызов конструктора myclass ob("123")
    myclass ob1 = "123";
    cout << "ob1: " << ob1.geta() << endl;
    cout << "ob2: " << ob2.geta() << endl;
    return 0;
}
```

13.6. Спецификаторы сборки и ключевое слово *asm*.

В C++ поддерживаются два важных механизма для облегчения связи C++ с другими языками программирования. Первым является *спецификатор сборки (linkage specifier)*, который сообщает компилятору, что одна или более функций вашей программы на C++ будет компоноваться с другим языком программирования, который может иметь другие соглашения о передаче параметров процедуре заполнения стека и т.д. Вторым является ключевое слово **asm**, которое позволяет вставлять в исходную программу команды ассемблера. Оба этих механизма рассматриваются в этом разделе.

По умолчанию все функции программы на C++ компилируются и компонуются как функции C++. Однако компилятору C++ можно сообщить, что функция будет компоноваться как функция, написанная на другом языке программирования. Все компиляторы C++ допускают компоновку функций либо как функций C, либо как функций C++. Некоторые также допускают компоновку функций для таких языков, как Pascal, Ada, Fortran. Чтобы компоновать функции для других языков программирования, используется следующая основная форма спецификатора сборки:

extern *“язык” прототип_функции;*

Здесь ***язык*** – это название языка программирования, как функцию которого вы хотите компоновать вашу функцию. Если необходимо использовать спецификатор сборки более чем для одной функции, используется такая его основная форма:

extern *“язык”* {
прототипы_функций;
}

Все спецификаторы сборки должны быть глобальными; их нельзя задавать внутри функций.

Чаще всего в программы на C++ приходится вставлять фрагменты программ на C. Путём задания сборки с «C» вы предотвращаете *искажения (mangling)* имён функций информацией о типе. Поскольку в C++ имеется возможность перегружать функции и создавать функции-члены, то каждому имени функции обычно сопутствует некоторая информация о её типе. С другой стороны, так как язык C не поддерживает ни перегрузки

функций, ни функций-членов, он не может распознавать имена функций, искажённые информацией об их типе. Указание компилятору необходимости сборки с «C» позволяет решить проблему.

Хотя вполне возможно совместно компоновать ассемблерные подпрограммы с программами на C++, часто легче использовать язык ассемблера в процессе написания программ на C++. В C++ поддерживается специальное ключевое слово **asm**, позволяющее вставлять ассемблерные инструкции в функции C++. Преимущество встроенного ассемблера в том, что ваша программа полностью компилируется как программа на C++, и нет необходимости раздельно с ней компилировать, а затем совместно компоновать ассемблерные файлы. Здесь показана основная форма ключевого слова **asm**:

asm ***(«ас_инструкция»);***

где ***ас_инструкция*** – это ассемблерная инструкция, которая будет встроена в вашу программу.

Важно отметить, что в некоторых компиляторах поддерживаются следующие три, несколько иные формы инструкции **asm**:

asm ***ас_инструкция;***

asm ***ас_инструкция физический_конец_строки***

asm {
последовательность ассемблерных инструкций
}

Здесь ассемблерные инструкции не выделяются кавычками. Для правильного встраивания ассемблерных инструкций вам понадобится изучить техническую документацию на ваш компилятор.

Примеры

1. В этой программе функция **func()** компонуется не как функция C++, а как функция C:

```
// Демонстрация спецификатора сборки
#include <iostream>
using namespace std;

extern "C" int func(int x); // компонуется как функция C

// Теперь функция компонуется как функция C.
int func(int x)
{
  return x/3;
}
```

Теперь эта функция может компоноваться с программой, которая компилировалась компилятором C.

2. В представленной ниже программе компилятору сообщается, что функции **f1(), f2(), f3()** должны компоноваться как функции C:

```
extern «C» {
  void f1();
  int f2(int x);
  double f3(double x, int *p);
}
```

3. В этом фрагменте в функцию **func()** вставляется несколько ассемблерных инструкций:

```
// Не пытайтесь выполнить эту функцию!
void func()
{
  asm (“mov bp”, sp);
  asm (“push ax”);
  asm (“mov c1, 4”);
  //…
}
```

13.7. Массивы в качестве объектов ввода/вывода.

Помимо ввода/вывода на экран и в файл, в C++ поддерживается целый ряд функций, в которых в качестве устройств для ввода и вывода используются массивы. Хотя ввод/вывод с использованием массивов *(array-based I/O)* в C++ концептуально перекликается с аналогичным вводом/выводом в C (в особенности это касается функций **sscanf()** и **sprintf()** языка C), ввод/вывод с использованием массивов в C++ более гибкий и полезный, поскольку он позволяет интегрировать в него определённые пользователем типы данных. Хотя охватить все аспекты массивов в качестве объектов ввода/вывода невозможно, здесь рассматриваются их наиболее важные и часто используемые свойства.

Важно понимать, что для реализации ввода/вывода с использованием массивов тоже нужны потоки. Все, что вы узнали о вводе/выводе C++ из глав 9 и 9 применимо и к вводу/выводу с использованием массивов. При этом, чтобы узнать о всех достоинствах массивов в качестве объектов ввода/вывода, вам следует познакомиться с несколькими новыми функциями. Эти функции предназначены для связывания нужного потока с некоторой областью памяти. После того как эта операция выполнена, весь ввод/вывод производится посредством тех самых функций для ввода и вывода, о которых вы уже знаете.

Прежде тем как начать пользоваться массивом в качестве объектов ввода/вывода, необходимо удостовериться в том, что в вашу программу включён заголовок **<strstream>**. В этом заголовке определяются классы **istrstream, ostrstream** и **strstream**. Эти классы образуют, соответственно, основанные на использовании массивов потоки для ввода, вывода и ввода/вывода. Базовым для этих классов является класс **ios**, поэтому все функции и манипуляторы классов **istream, ostream** и **iostream** имеются также и в классах **istrstream, ostrstream** и **strstream**.

Для вывода в символьный массив используйте следующую основную форму конструктора класса **ostrstream**:

ostrstream *поток_вывода* (char **буфер,* streamsize *размер,* openmode *режим* = ios::out);

Здесь ***поток_вывода*** – это поток, который связывается с массивом, заданным через указатель ***буфер***. Параметр ***размер*** определяет размер массива. Параметр ***режим*** по

умолчанию обычно настроен на обычный режим вывода, но вы можете задать любой, определённый в классе **ios**, флаг режиме вывода. (За подробностями обращайтесь к главе 9.)

После того как массив открыт для вывода, символы будут выводиться в массив вплоть до его заполнения. Переполнение массива произойти не может. Любая попытка переполнения массива приведёт к ошибке ввода/вывода. Для определения количества записанных в массив символов используйте приведенную ниже функцию-член **pcount()**:

int pcount();

Функция должна вызываться только в связи с потоком и возвращает она число символов, записанных в массив, включая нулевой символ завершения.

Чтобы открыть массив для ввода из него, используйте следующую форму конструктора класса **istrstream**:

istrstream *поток_вывода* (const char * *буфер*);

Здесь ***буфер*** – это указатель на массив, из которого будут вводиться символы. Поток ввода обозначен через ***поток_вывода***. При считывании входной информации из массива, функция **eof()** возвращает истину при достижении конца массива.

Чтобы открыть массив для ввода/вывода, используйте следующую форму конструктора класса **strstream**:

strstream *поток_ввод_вывод* (char * *буфер*, streamsize *размер*, openmode *режим* = ios::in | ios::out);

Здесь ***поток_ввод_вывод*** – это поток ввода/вывода, для которого в качестве объекта ввода/вывода через указатель ***буфер*** задан массив длиной в ***размер*** символов.

Важно помнить, что все описанные ранее функции ввода/вывода работают и с массивами, включая функции ввода/вывод двоичных файлов и функции произвольного доступа.

Примеры

1. В этом простом примере показано, как открыть массив для вывода и записать в него данные:

```
// Короткий пример вывода в массив
#include <iostream>
#include <strstream>
using namespace std;

int main()
{
  char buf[255]; // буфер для вывода
  ostrstream ostr(buf, sizeof buf); // открытие массива для вывода
  ostr << "ввод/вывод через массивы работает с потоками\n";
  ostr << "точно также, как при обычном вводе/выводе\n " << 100;
  ostr << ' ' << 123.23 << '\n';
  // можно также использовать манипуляторы
  ostr << hex << 100 << ' ';
  // или флаги формата
```

```
   ostr.setf(ios::scientific);
   ostr << 123.23 << '\n';
   ostr << ends;
   // вывод на экран полученной строки
   cout << buf;
   return 0;
}
```

В результате выполнения программы на экран выводится следующее:

```
ввод/вывод через массивы работает с потоками
точно так же, как при обычном вводе/выводе
100 123.23
64 01.2323e+02
```

Как вы могли заметить, перегруженные операторы ввода/вывода, встроенные манипулятора ввода/вывода, функции-члены и флаги формата полностью доступны и для ввода/вывода с использованием массивов. (Это также относится ко всем манипуляторам и перегруженным операторам ввода/вывода, которые вы создаёте для своих классов)

В представленной выше программе с помощью манипулятора **ends** в массив специально добавляется завершающий нулевой символ. Будет ли нулевой символ занесен в массив автоматически или нет, зависит от реализации. Поэтому, если это важно для вашей программы, лучше специально записать в массив завершающий нулевой символ.

2. Пример ввода данных из массива:

```
// Пример ввода из массива
#include <iostream>
#include <strstream>
using namespace std;

int main()
{
   char buf[] = "Привет 100 123.125 a";
   istrstream istr(buf); // открытие массива для ввода из него
   int i;
   char str[80];
   float f;
   char c;
   istr >> str >> i >> f >> c;
   cout << str << ' ' << i << ' ' << f;
   cout << ' ' << c << '\n';
   return 0;
}
```

Эта программа считывает и воспроизводит данные, содержащиеся в массиве, заданном указателем **buf**.

3. Запомните, что массив для ввода, после того как он связан с потоком, становится похожим на файл. Например, в следующей программе для считывания содержимого массива по адресу **buf** используются функции **eof()** и **get()**:

```
/* Демонстрация того факта, что функции eof() и get() работают с вводом/выводом основанным на использовании массивов
*/
#include <iostream>
#include <strstream>
```

```
using namespace std;

int main()
{
   char buf[] = "Привет 100 123.125 a";
   istrstream istr(buf);
   char c;
   while(!istr.eof()) {
      istr.get(c);
      cout << c;
   }
   return 0;
}
```

4. В следующей программе выполняется ввод данных из массива и вывод данных в массив:

```
// Демонстрация ввода/вывода с использованием массивов
#include <iostream>
#include <strstream>
using namespace std;

int main()
{
   char iobuf[255];
   strstream iostr (iobuf, sizeof iobuf);
   iostr << "Это проверка\n";
   iostr << 100 << hex << ' ' << 100 << ends;
   char str[80];
   int i;
   iostr.getline(str, 79); // считывает строку вплоть до \n
   iostr >> dec >> i; // считывает 100
   cout << str << ' ' << i << ' ';
   iostr >> hex >> i;
   cout << hex<< i;
   return 0;
}
```

Эта программа сначала записывает информацию в массив по адресу **iobuf**, а затем считывает её обратно. Сначала с помощью функции **getline()** строка «**Это проверка**» целиком считывается в массив, далее считывается десятичное значение 100, а затем шестнадцатеричное 0x64.