Глава 3

Подробное изучение классов

В этой главе вы продолжите изучение классов. Вы узнаете о том, как присвоить один объект другому, как объекты передаются функциям в качестве аргументов, как сделать объект возвращаемым значением функций. Вы также узнаете о новом важном типе функций: дружественных (friend) функциях.

3.1. Присваивание объектов

Если тип двух объектов одинаков, то один объект можно присвоить другому. По умолчанию, когда один объект присваивается другому, делается поразрядная копия всех данных-членов копируемого объекта. Например, когда объект **o1** присваивается объекту **o2,** то содержимое всех данных объекта **o1** копируется в соответствующие члены объекта **o2.** Это иллюстрируется следующей программой:

```
// Пример присваивания объекта
#include <iostream>
using namespace std;
class myclass {
        int a, b;
public:
       void set (int i, int j)  a = i; b = j; )
       void show() { cout << a <<' ' << b <<"\n"; }
 };
 int main()
 {
       myclass o1, o2;
        ol.set(10, 4);
        // o1 присваиваетсяо2 o2 = o1; '
         01.show();
         02.show();
        return 0;
 }
```

В этом примере переменным **a** и **b** объекта **o1** присваиваются соответственно значения 10 и 4. Далее объект o1 присваивается объекту **o2**. Это приводит к тому, что текущее значение переменной **ol.a** присваивается переменной **o2.a**, а текущее значение переменной **ol.b** — **o2.b**. Таким образом, в процессе выполнения программа выведет на экран следующее;

10 4

10 4

Запомните, что присваивание двух объектов просто делает одинаковыми данные этих объектов. Два объекта остаются по-прежнему совершенно независимыми. Например, после выполнения присваивания вызов функции-члена **ol.set()** для задания значения **o1.a** не влияет ни на объект **o2**, ни на значение его переменной **a**.

Примеры

1. Важно понимать, что в инструкции присваивания можно использовать только объекты одинакового типа. Если тип объектов разный, то при компиляции появится сообщение об ошибке. Более того, важно не только чтобы типы объектов были физически

одинаковыми, а чтобы одинаковыми были также имена типов. Например, следующая программа неправильна:

```
// Эта программа содержит ошибку
#include <iostream>
using namespace std;

class myclass {
   int a, b;
public:
   void set(int i, int j) { a = i; b = j; }
   void show() { cout << a << ' ' << b << "\n"; }
};

/* Этот класс похож на класс myclass, но из-за другого имени класса для компилятора он считается другим типом.
*/
class yourclass {
   int a, b;
public:
   void set(int i, int j) { a = i; b = j; }
   void show() { cout << a << ' ' << b << "\n"; }
};

int main()
{
   myclass o1;
   yourclass o2;

   o1.set(10, 4);

   o2 = o1; // ОШИБКА, присваивание объектов разных типов

   o1.show();
   o2.show();

   return 0;
}
```

Несмотря на то что классы **myclass и yourclass** физически одинаковы, они, трактуются компилятором как разные, поскольку имеют разные имена типов.

2. Важно понимать, что все данные-члены одного объекта при выполнении присваивания присваиваются данным-членам другого объекта. Это относится и к сложным данным, таким как массивы. Например, в следующей версии уже знакомого нам класса **stack** символы реально помещаются только в стек **sl,** но после выполнения присваивания массив **stck** объекта **s2** также содержит символы **a**, **b** и **c**.

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

#define SIZE 10

// Объявление класса stack для символов
class stack {
   char stck[SIZE]; // содержит стек
   int tos; // индекс вершины стека
public:
   stack(); // конструктор
   void push(char ch); // помещает в стек символ
```

```
  char pop(); // выталкивает символ из стека
};

// Инициализация стека
stack::stack()
{
  cout << "Работа конструктора стека\n";
  tos = 0;
}

// Помещение символа в стек
void stack::push(char ch)
{
  if (tos==SIZE) {
    cout << "Стек полон \n";
    return;
  }
  stck[tos] = ch;
  tos++;
}

// Выталкивание символа из стека
char stack::pop()
{
  if (tos==0) {
    cout << "Стек пуст \n";
    return 0; // возврат нуля при пустом стеке
  }
  tos--;
  return stck[tos];
}

int main()
{
  // образование двух, автоматически инициализируемых стеков
  stack s1, s2;
  int i;

  s1.push('a');
  s1.push('b');
  s1.push('c');

  // копирование s1 в s2
  s2 = s1; // теперь s2 и s1 идентичны

  for(i=0; i<3; i++) cout << "символ из s1:" << s1.pop() << "\n";
  for(i=0; i<3; i++) cout << "символ из s2:" << s2.pop() << "\n";

  return 0;
}
```

3.При присваивании одного объекта другому необходимо быть очень внимательным. Например, рассмотрим несколько измененный класс **strtypt,** который мы изучали в главе 2. Попытайтесь найти ошибку в этой программе.

```
// Эта программа содержит ошибку
#include <iostream>
#include <cstring>
#include <cstdlib>
using namespace std;

class strtype {
```

```
    char *p;
    int len;
public:
    strtype(char *ptr);
    ~strtype();
    void show();
};

strtype::strtype(char *ptr)
{
    len=strlen(ptr);
    p=(char *) malloc(len+1);
    if(!p) {
        cout << "Ошибка выделения памяти\n";
        exit(1);
    }
    strcpy(p, ptr);
}

strtype::~strtype()
{
    cout << "Освобождение памяти по адресу p\n";
    free(p);
}

void strtype::show()
{
    cout << p << " - длина: " << len;
    cout << "\n";
}

int main()
{
    strtype s1("Это проверка"), s2("Мне нравится C++");

    s1.show();
    s2.show();

    // s1 присваивается s2 - это ведет к ошибке
    s2 = s1;

    s1.show();
    s2.show();

    return 0;
}
```

Ошибка в этой программе весьма коварна. Когда создаются объекты **sl** и **s2**, то для хранения в этих объектах строк выделяется память. Указатель на выделенную каждому объекту память хранится в переменной **p**. Когда объект **strtype** удаляется, эта память освобождается. Однако когда объект **sl** присваивается объекту s2, то указатель **p** объекта **s2** начинает указывать на ту же самую область памяти, что и указатель **p** объекта **sl.** Таким образом, когда эти объекты удаляются, то память, на которую указывает указатель p объекта **sl,** освобождается *дважды,* а память, на которую до присваивания указывал указатель p объекта **s2**, не освобождается *вообще.*

Хотя в данном случае эта ошибка и не опасна, в реальных программах динамическим распределением памяти она может вызвать крах программы. Как показано в этом примере, при присваивании одного объекта другому вы должны быть уверены в том, что не удаляете нужную информацию, которая может понадобиться в дальнейшем.

3.2. Передача объектов функциям

Объекты можно передавать функциям в качестве аргументов точно так же, как передаются данные других типов. Просто объявите параметр функции, как имеющий тип класса, и затем используйте объект этого класса в качестве аргумента при вызове функции. Как и для данных других типов, по умолчанию объекты передаются в функции по значению.

Примеры

1. В этом коротком примере объект передается функции:

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

class samp {
   int i;
public:
   samp(int n) { i = n; }
   int get_i() { return i; }
};

// Возвращает квадрат o.i.
int sqr_it(samp o)
{
   return o.get_i() * o.get_i();
}

int main()
{
   samp a(10), b(2);

   cout << sqr_it(a) << "\n";
   cout << sqr_it(b) << "\n";

   return 0;
}
```

В этой программе создается класс **samp**, который содержит одну целую переменную **i**. Функция **sqr_it()** получает аргумент типа **samp**, а возвращаемым значением является квадрат переменной **i** этого объекта. Результат работы программы — это значения 100 и 4.

2. Как уже установлено методом передачи параметров в C++, включая объекты, по умолчанию является передача объекта по значению. Это означает, что внутри функции создается копия аргумента и эта копия, а не сам объект, используется функцией. Поэтому изменение копии объекта внутри функции не влияет на сам объект. Это иллюстрируется следующим примером:

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

class samp {
   int i;
public:
   samp(int n) { i = n; }
   void set_i(int n) { i = n; }
   int get_i() { return i; }
};

/* Заменяет o.i его квадратом. Однако, это не влияет на объект, используемый для вызова sqr_it
*/
void sqr_it(samp o)
{
   o.set_i( o.get_i() * o.get_i());

   cout << "Для копии объекта a значение i равно: " << o.get_i();
   cout << "\n";
}

int main()
{
   samp a(10);

   sqr_it(a); // передача a
по значению

   cout << "но переменная a.i в функции main() не изменилась: ";
   cout << a.get_i(); // выводится 10

   return 0;
}
```

В результате работы программы на экран выводится следующее:

Для копии объекта a значение i равно; 100 но переменная a.i в функции main() не изменилась: 10

3. Как и в случае с переменными других типов, функции может быть передано не значение объекта, а его адрес. В этом случае функция может изменить значение аргумента, используемого в вызове. Например, в рассматриваемом ниже варианте программы из предыдущего примера значение объекта, чей адрес используется при вызове функции **sqr_it(),** действительно меняется.

```
/* Теперь функции sqr_it() передается адрес объекта и функция может изменить значение аргумента, адрес
    которого используется при вызове.
*/
#include <iostream>
using namespace std;

class samp {
   int i;
public:
   samp(int n) { i = n; }
   void set_i(int n) { i = n; }
   int get_i() { return i; }
};

/* Заменяет переменную o.i ее квадратом. Это влияет на объект, используемый при вызове
```

```
*/
void sqr_it(samp *o)
{
   o->set_i(o->get_i() * o->get_i());

   cout << " Для копии объекта a значение i равно: " << o->get_i();
   cout << "\n";
}

int main()
{
   samp a(10);

   sqr_it(&a); // функции sqr_it() передан адрес объекта a

   cout << "Теперь значение объекта a в функции main() изменилось: ";
   cout << a.get_i(); // выводится 100

   return 0;
}
```

Теперь результат работы программы следующий:

Для объекта a значение i равно: 100 Теперь значение объекта a в функции main() изменилось: 100

4. Если при передаче объекта в функцию делается его копия, это означает, что появляется новый объект. Когда работа функции, которой был передан объект, завершается, то копия аргумента удаляется. Возникают два вопроса. Во-первых, вызывается ли конструктор объекта, когда создается его копия? Во-вторых, вызывается ли деструктор объекта, когда эта копия удаляется? Ответ на первый вопрос может показаться неожиданным.

   Когда при вызове функции создается копия объекта, конструктор копии *не* вызывается. Смысл этого понять просто. Поскольку конструктор обычно используется для инициализации некоторых составляющих объекта, он не должен вызываться при создании копии уже существующего объекта- Если бы это было сделано, то изменилось бы содержимое объекта, поскольку при передаче объекта функции необходимо его текущее, а не начальное состояние.

   Однако если работа функции завершается и копия удаляется, то деструктор копии вызывается. Это происходит потому, что иначе оказались бы невыполненными некоторые необходимые операции. Например, для копии может быть выделена память, которую, после завершения работы функции, необходимо освободить.

   Итак, при создании копии объекта, когда он используется в качестве аргумента функции, конструктор копии не вызывается. Однако, когда копия удаляется (обычно это происходит при возвращении функцией своего значения), вызывается ее деструктор.

   Следующая программа иллюстрирует эти положения:

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

class samp {
```

```
  int i;
public:
  samp(int n) {
    i = n;
    cout << "Работа конструктора\n";
   }
  ~samp() { cout << "Работа деструктора\n"; }
  int get_i() { return i; }
};

// Возвращает квадрат переменной o.i
int sqr_it(samp o)
{
  return o.get_i() * o.get_i();
}

int main()
{
  samp a(10);

  cout << sqr_it(a) << "\n";

  return 0;
}
```

Эта программа выводит следующее:

```
Работа конструктора
100
Работа деструктора
Работа деструктора
```

Обратите внимание, что конструктор вызывается только один раз. Это происходит при создании объекта a. Однако деструктор вызывается дважды.

Первый раз он вызывается для копии, созданной, когда объект **a** был передан функции **sqr_it()**, Другой — для самого объекта **a**.

Тот факт, что деструктор объекта, являющегося копией передаваемого функции аргумента, выполняется при завершении работы функции, может стать потенциальным источником проблем. Например, если для объекта, используемого в качестве аргумента, выделена динамическая память, которая освобождается при его удалении, тогда и для копии объекта при вызове деструктора будет освобождаться та же самая память. Это приведет к повреждению исходного объекта. (Для примера см. упражнение 2 данного раздела.) Чтобы избежать такого рода ошибок, важно убедиться в том, что деструктор копии объекта, используемого в качестве аргумента, не вызывает никаких побочных эффектов, которые могли бы повлиять на исходный аргумент,

Как вы, возможно, уже догадались, одним из способов обойти проблему удаления деструктором необходимых данных при вызове функции с объектом в качестве аргумента должна стать передача функции не самого объекта, а его адреса. Если функции передается адрес объекта, то нового объекта не создается и поэтому при возвращении функцией своего значения деструктор не вызывается. (Как вы увидите в следующей главе, в C++ имеется и иное, более элегантное решение этой задачи.) Тем не менее, имеется другое, лучшее решение, о котором вы узнаете, изучив особый тип конструктора, а именно конструктор копий (copy constructor). Конструктор копий позволяет точно определить порядок создания копий объекта. (О конструкторах копий рассказывается в главе 5.)

3.3 Объекты в качестве возвращаемого значения функций

Так же как объект может быть передан функции в качестве аргумента, он может быть и возвращаемым значением функций. Для этого, во-первых, объявите функцию так, чтобы ее возвращаемое значение имело тип класса. Во-вторых, объект этого типа возвратите с помощью обычной инструкции **return.**

Имеется одно важное замечание по поводу объектов в качестве возвращаемого значения функций: если функция возвращает объект, то для хранения возвращаемого значения автоматически создается временный объект. После того как значение возвращено, этот объект удаляется. Удаление этого временного объекта может приводить к неожиданным побочным эффектам, что иллюстрируется в примере 2 этого раздела.

Примеры

1. Пример функции с объектом в качестве возвращаемого значения:

```
// Возвращение объекта из функции
#include <iostream>
#include <cstring>
using namespace std;

class samp {
   char s[80];
public:
   void show() { cout << s << "\n"; }
   void set(char *str) { strcpy(s, str); }
};

// Возвращает объект типа samp
samp input()
{
   char s[80];
   samp str;

   cout << "Введите строку: ";
   cin >> s;

   str.set(s);

   return str;
}

int main()
```

```
{
   samp ob;

   // присваивание возвращаемого значения объекту ob
   ob = input();
   ob.show();

   return 0;
}
```

В этом примере функция **input()** создает локальный объект str и считывает строку с клавиатуры. Эта строка копируется в **str.s,** и затем функция возвращает объект **str.** Внутри функции **main()** при вызове функции **input()** возвращаемый объект присваивается объекту **ob.**

2. Следует быть внимательными при возвращении объектов из функций, если эти объекты содержат деструктор, поскольку возвращаемый объект выходит из области видимости, как только функция возвращает его значение. Например, если функция возвращает объект, имеющий деструктор, который освобождает динамически выделенную память, то эта память будет освобождена независимо от того, использует ли ее объект, которому присваивается возвращаемое значение, или нет. Например, рассмотрим неправильную версию предыдущей программы:

```
// При возвращении объекта генерируется ошибка
#include <iostream>
#include <cstring>
#include <cstdlib>
using namespace std;

class samp {
   char *s;
public:
   samp() { s = '\0'; }
   ~samp() { if(s) free(s);  cout << "Освобождение памяти по адресу s\n";}
   void show() { cout << s << "\n"; }
   void set(char *str);
};

// Загружает строку
void samp::set(char *str)
{
   s = (char *) malloc(strlen(str)+1);
   if(!s) {
      cout << "Ошибка выделения памяти\n";
      exit(1);
   }

   strcpy(s, str);
}

// Возвращает объект типа samp
samp input()
```

```
{
   char s[80];
   samp str;

   cout << "Введите строку: ";
   cin >> s;

   str.set(s);
   return str;
}

int main()
{
   samp ob;

   // возвращаемое значение присваивается объекту ob
   ob = input(); // Это ведет к ошибке
   ob.show();

   return 0;
}
```

Здесь показан результат работы программы:

```
Введите строку: Привет
Освобождение памяти по адресу s
Освобождение памяти по адресу s
Привет
Освобождение памяти по адресу s
Null pointer assignment
```

Обратите внимание, что деструктор класса **samp** вызывается трижды. Первый раз, когда локальный объект **str** выходит из области видимости при возвращении функцией **input()** своего значения. Второй раз **~samp()** вызывается тогда, когда удаляется временный объект, возвращаемый функцией **input().**

Запомните, когда объект возвращается функцией, автоматически генерируется невидимый (для вас) временный объект, который и хранит возвращаемое значение. В этом случае временный объект — это просто копия объекта str, являющегося возвращаемым значением функции- Следовательно, после того как функция возвратила свое значение, выполняется деструктор временного объекта. И наконец, при завершении программы вызывается деструктор объекта **ob** в функции **main().**

Проблема в этой ситуации в том. что при первом выполнении деструктора память, выделенная для хранения вводимой с помощью функции **input()** строки, освобождается. Таким образом, при двух других вызовах деструктора класса **samp** не только делается попытка освободить уже освобожденный блок динамической памяти, но в процессе работы происходит разрушение самой системы динамического распределения памяти и, как доказательство этому, появляется сообщение "Null pointer assignment". (В зависимости от вашего компилятора, модели используемой памяти и тому подобного при попытке выполнить программу

это сообщение может и не появиться.)

Ключевым моментом в понимании проблемы, описанной в этом примере» является то, что при возвращении функцией объекта для временного объекта, который и является возвращаемым значением функции, вызывается деструктор. (Как вы узнаете в главе 5, для решения проблемы в такой ситуации можно воспользоваться конструктором копий.)

3.4. Дружественные функции: обзор

Возможны ситуации, когда для получения доступа к закрытым членам класса вам понадобится функция, не являющаяся членом этого класса. Для достижения этой цели в C++ поддерживаются *дружественные функции (friend functions).* Дружественные функции не являются членами класса, но тем не менее имеют доступ к его закрытым элементам.

В пользу существования дружественных функций имеются два довода, связанные с перегрузкой операторов и созданием специальных функций ввода/вывода. Об этом использовании дружественных функций вы узнаете несколько позднее. Сейчас познакомимся с третьим доводом в пользу существования таких функций. Вам наверняка когда-нибудь понадобится функция, которая имела бы доступ к закрытым членам *двух или более* разных Пассов. Рассмотрим, как создать такую функцию.

Дружественная функция задается так же, как обычная, не являющаяся членом класса, функция. Однако в объявление класса, для которого функция °Удет дружественной, необходимо включить ее прототип, перед которым давится ключевое слово **friend.** Чтобы понять, как работает дружественная ФУНКЦИЯ, рассмотрим следующую короткую программу:

```
// Пример использования дружественной функции #include <iostream>
using namespace std;

class myclass {
int n, d;
public:
      myclass(int i, int j) { n = i; d = j; 1
      // объявление дружественной функции для класса myclass
friend int isfactor(myclass ob);
};
/* Здесь представлено определение дружественной функции. Она возвращает истину, если n делится
без остатка на d. Отметьте, что ключевое слово friend в определении функции isfactor() не
используется.
*/ int isfactor(myclass ob)
```

```
    {
    if(!(ob.n % ob.d)) return 1;
    else return 0;


int main()
    {
    myclass obi(10, 2), ob2(13, 3);

if (isfactor(ob1) cout << "10 без остатка делится на 2\n";
else cout << "10 без остатка не делится на 2\n";

if (isfactor (ob2) cout <<"13 без остатка делится на 3\n";
else cout <<"13 без остатка не делится на 3\n";
return 0;
```

В этом примере в объявлении класса myclass объявляются конструктор и дружественная функция **isfactor().** Поскольку функция **isfactor()** дружественна для класса myclass, то функция **isfactor()** имеет доступ к его закрытой части. Поэтому внутри функции **isfactor()** можно непосредственно ссылаться на объекты **ob.n** и **ob.d.**

Важно понимать, что дружественная функция не является членом класса. для которого она дружественна. Поэтому невозможно вызвать дружественную функцию, используя имя объекта и оператор доступа к члену класса (точку или стрелку). Например, по отношению к предыдущему примеру эта инструкция неправильна:

```
ob1.isfactor(); // неправильно, Isfactor() - это не функция-член
```

На самом деле дружественная функция должна вызываться точно **так же,** как и обычная функция.

Хотя дружественная функция "знает" о закрытых элементах класса, для которого она является дружественной, доступ к ним она может получить только через объект этого класса. Таким образом, в отличие от функции-члена **myclass,** в котором можно непосредственно упоминать переменные **n и d,** дружественная функция может иметь доступ к этим переменным только через объект, который объявлен внутри функции или передан ей.

Примеры

1. Обычно дружественная функция бывает полезна тогда, когда у двух разных классов имеется нечто общее, что необходимо сравнить. Например, рассмотрим следующую программу, в которой создаются классы **car** (легковая машина) и **truck** (грузовик), причем оба содержат в закрытой переменной, скорость

соответствующего транспортного средства:

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

class truck; // предварительное объявление

class car {
    int passengers;
    int speed;
public:
    car(int p, int s) { passengers = p; speed = s; }
    friend int sp_greater(car c, truck t);
};

class truck {
    int weight;
    int speed;
public:
    truck(int w, int s) { weight = w; speed = s; }
    friend int sp_greater(car c, truck t);
};

/* Возвращает положительное число, если легковая машина быстрее грузовика.
Возвращает 0 при одинаковых скоростях.
Возвращает отрицательное число, если грузовик быстрее легковой машины.
*/
int sp_greater(car c, truck t)
{
    return c.speed - t.speed;
}

int main()
{
    int t;
    car c1(6, 55), c2 (2, 120);
    truck t1(10000, 55), t2(20000, 72);

    cout << "Сравнение значений c1 и t1:\n";
    t = sp_greater(c1, t1);
    if(t<0) cout << "Грузовик быстрее. \n";
    else if(t==0) cout << "Скорости машин одинаковы. \n";
    else cout << "Легковая машина быстрее. \n";

    cout << "\nСравнение значений c2 и t2:\n";
    t = sp_greater(c2, t2);
    if(t<0) cout << "Грузовик быстрее. \n";
    else if(t==0) cout << "Скорости машин одинаковы. \n";
    else cout << "Легковая машина быстрее. \n";

    return 0;
}
```

В этой программе имеется функция **sp_greater(),** которая дружественна для классов **car и truck.** (Как уже установлено, функция может быть дружественной двум и более классам.) Эта функция возвращает положительное число, если объект **car** движется быстрее объекта **truck,** нуль, если их скорости одинаковы, и отрицательное число, если скорость объекта **truck** больше, **чем** скорость объекта **car.**

Эта программа иллюстрирует один важный элемент синтаксиса C++ — *предварительное объявление (forward declaration),* которое еще называют *ссылкой вперед (forward reference).* Поскольку функция **sp_greater()** получает параметры обоих классов car и **truck,** то логически невозможно объявить и тот и другой класс перед включением функции **sp_greater()** в каждый из них. Поэтому необходим иной способ сообщить компилятору имя класса без его фактического объявления. Этот способ и называется предварительным объявлением. В C++, чтобы информировать компилятор о том, что данный идентификатор является именем класса, перед первым использованием имени класса вставляют следующую строку:

class ***имя_класса;***

Например, в предыдущей программе предварительным объявлением является инструкция

class truck;

после которой класс **truck** можно использовать в объявлении дружественной функции **sp_greater()** без опасения вызвать ошибку компилятора.
Функция может быть членом одного класса и дружественной другому. Например, если переписать предыдущий пример так, чтобы функция **sp_greater()** являлась членом класса **car** и дружественной классу **truck,** то получится следующая программа:

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

class truck; // предварительное объявление

class car {
   int passengers;
   int speed;
public:
   car(int p, int s) { passengers = p; speed = s; }
   int sp_greater(truck t);
};

class truck {
   int weight;
   int speed;
public:
   truck(int w, int s) { weight = w; speed = s; }
```

```
    // отметьте новое использование оператора расширения области видимости
    friend int car::sp_greater(truck t);
};

/* Возвращает положительное число, если легковая машина быстрее грузовика. Возвращает 0 при
    одинаковых скоростях. Возвращает отрицательное число, если грузовик быстрее легковой
    машины.
*/
int car::sp_greater(truck t)
{
/* Поскольку функция sp_greater() - это член класса car, ей должен передаваться только объект truck
*/
    return speed - t.speed;
}

int main()
{
    int t;
    car c1(6, 55), c2 (2, 120);
    truck t1(10000, 55), t2(20000, 72);

    cout << "Сравнение значений c1 и t1:\n";
    t = c1.sp_greater(t1); // вызывается как функция-член класса car
    if(t<0) cout << "Грузовик быстрее. \n";
    else if(t==0) cout << "Скорости машин одинаковы. \n";
    else cout << "Легковая машина быстрее. \n";

    cout << "\nСравнение c2 и t2:\n";
    t = c2.sp_greater(t2); // вызывается как функция-член класса car
    if(t<0) cout << "Грузовик быстрее. \n";
    else if(t==0) cout << "Скорости машин одинаковы. \n";
    else cout << "Легковая машина быстрее. \n";

    return 0;
}
```

Обратите внимание на новое использование оператора расширения области видимости, который имеется в объявлении дружественной функции внутри объявления класса **truck.** В данном случае он информирует компилятор о том, что функция **sp_greater()** является членом класса **car.**

Существует простой способ запомнить, где в такой ситуации нужно указывать оператор расширения области видимости: сначала идет имя класса потом — оператор расширения области видимости и последним — имя функции-члена. Таким образом член класса будет полностью задан.

Фактически при упоминании в программе члена класса никогда не помешает полностью (с именем класса и оператором расширения области видимости) задать его имя. Однако при использовании объекта для вызова функции-члена или для доступа к переменной-члену, полное имя обычно излишне и употребляется редко. Например, инструкция

t = cl.sp greater(t1);

может быть написана с указанием (избыточным) оператора расширения области видимости и именем класса **car;**

t = cl.car:**:sp_greater(t1);**

Поскольку объект **c1** является объектом типа **car**, компилятор уже и так знает, что функция **sp_greater()** — это член класса **car,** что делает необязательным полное задание имени класса.