Наследование

продолжение

Модификаторы доступа

В C++ существует три модификатора доступа к членам классов и структур:

public - доступ для всех

protected - доступ только для самого класса и его наследников

private - доступ только для самого класса

Эти же модификаторы можно указывать при наследовании:

```
struct B : public A {...};
struct C : protected A {...};
struct D : private A {...};
```

Если модификатор не указан, то по умолчанию для классов **private**.

public-наследование позволяет установить между классами отношение ЯВЛЯЕТСЯ  если класс **B** открыто унаследован от **A**, то объект класса **B** ЯВЛЯЕТСЯ объектом класса **A**, но не наоборот.

private наследование выражает отношение РЕАЛИЗОВАНО_ПОСРЕДСТВОМ. Если класс **B** закрыто унаследован от **A**, то можно говорить, что объект класса **B** реализован посредством объекта класса **A**. В большинстве случаев закрытое наследование можно заменить агрегацией.

Закрытый (private) виртуальный метод базового класса нельзя вызвать из наследника, но без проблем можно переопределить.

При создании производного класса программа сначала создает объект его базового класса, т.е. объект базового класса должен быть создан до того, как программа дойдет до тела конструктора производного класса.

В C++ для этих целей используется список инициализаторов элементов.

```
manager::manager(char *name, char *position, char
*company_car, float salary, float bonus, int
stock_options) : employee(name, position, salary)
{
}
```

Списки инициализаторов

Конструктор для производного класса может использовать механизм списка инициализаторов для передачи значений конструктору базового класса.

Можно воспользоваться списком инициализаторов и для элементов производного класса – в список вместо имени класса нужно подставить имена элементов.

manager::***manager***(char *name, char *position, char *company_car, float salary, float bonus, int stock_options, const employee & Pt): employee (Pt), …
{
}

Полиморфное открытое наследование

Могут встретиться ситуации, когда поведение метода зависит от вызвавшего его объекта.

Такое поведение называется полиморфным. Его можно реализовать несколькими способами:

- Переопределить методы базового класса в производном классе;
- Использовать виртуальные методы.

Виртуальные методы — один из важнейших приемов реализации ***полиморфизма***.

Они позволяют создавать общий код, который может работать как с объектами базового класса, так и с объектами любого его класса-наследника.

При этом базовый класс определяет способ работы с объектами и любые его наследники могут предоставлять конкретную реализацию этого способа.

Такие методы без реализации называются «чистыми виртуальными» (перевод англ. *pure virtual*) или абстрактными.

Класс, содержащий хотя бы один такой метод, тоже является абстрактным.

- Наследники абстрактного класса должны предоставить реализацию для всех его абстрактных методов, иначе они, в свою очередь, будут абстрактными классами.
- Для каждого класса, имеющего хотя бы один виртуальный метод, создается *таблица виртуальных методов*. Каждый объект хранит указатель на таблицу своего класса.
- Для вызова виртуального метода используется такой механизм:
  - из объекта берется указатель на соответствующую таблицу виртуальных методов,
  - из нее, по фиксированному смещению, — указатель на реализацию метода, используемого для данного класса.
- При использовании множественного наследования или интерфейсов ситуация несколько усложняется за счёт того, что таблица виртуальных методов становится нелинейной.

Пример на C++, иллюстрирующий отличие виртуальных функций от невиртуальных:

```
class Ancestor
{
 public:  virtual void  function1 ()
  { cout << "Ancestor::function1()" << endl; }
void  function2 ()
  { cout << "Ancestor::function2()" << endl; }
};


class Descendant : public Ancestor
{
public:  virtual void  function1 ()
 { cout << "Descendant::function1()" << endl; }
void  function2 ()
{ cout << "Descendant::function2()" << endl; }};
```

```
Descendant *pointer = new Descendant ();
Ancestor *pointer_copy = pointer;
pointer->function1 ();
pointer->function2 ();
pointer_copy->function1 ();
pointer_copy->function2 ();
```

Результаты работы программы :

Descendant::function1()
Descendant::function2()
Descendant::function1()
Ancestor::function2()

При необходимости можно указать конкретную реализацию виртуальной функции, фактически вызывая её невиртуально:
pointer->Ancestor::function1 ();

Выведет Ancestor::function1(), игнорируя тип объекта

- Чтобы сделать класс абстрактным, нужно объявить одну из виртуальных функций чистой.
- Чистая виртуальная функция (pure virtual function) как бы намекает, что она будет реализована в производных классах.
- Чтобы сделать виртуальную функцию чистой (pure), нужно добавить после заголовка функции символы =0 (знак равенства и ноль):

```
class Base
{
public:
  virtual void method () =0;
   virtual ~Base() =0;
};
```

Нельзя создавать объекты класса Base, так как он стал абстрактным!!!

- Символы =0 необязательно добавлять ко всем виртуальным функциям, достаточно добавить к одной.
- **Виртуальные функции** класса позволяют переопределять их реализацию в наследниках, причем выбор конкретной реализации осуществляется динамически, в процессе выполнения программы.

Пример

```
class Employee // Класс “Служащий”
{
   private:
     char name[40]; // Имя служащего
   public:
    Employee(char* n);
//Чистая виртуальная функция
    virtual void* promote()=0;
```

Абстрактный класс не может быть реализован в объекте.

Так, следующая строка:

```
Employee s("My name");
```

вызовет ошибку, о которой сообщит компилятор.

Однако программа может объявить указатель абстрактного класса, так что следующая строка вполне допустима:

```
Employee * sPtr;
```

Компоненты абстрактного класса могут наследоваться.

Если все чистые виртуальные методы класса перегружены правилами, не являющимися чистыми и виртуальными, класс может не являться абстрактным.

В следующем примере класс Secretary может быть реализован объектом, поскольку функция promote была перегружена и имеет теперь другой смысл.

Пример

```
class Secretary: public Employee
{
 private:
   int moreData;
 public:
   Secretary(char* n): Employee(n) { … }
    virtual void * promote();
};
Secretary secr("Another Name");
```

Адрес объекта Secretary может быть передан в функцию, которая ожидает передачи Employee:

```
void fn(Employee*);
Secretary secr("Another Name");
fn(&secr);
```

Конструирование объекта происходит поэтапно и начинается созданием объекта самого первого класса в иерархии наследования.

Во время этого процесса перед вызовом конструктора каждого класса указатель на VFT устанавливается равным указателю на VFT текущего конструируемого класса.

Например, есть 3 класса: **A**, **B**, **C** (**B** наследуется от **A**, **C** наследуется от **B**).

При создании экземпляра **C**, произойдут 3 последовательных вызова конструкторов: сначала **A()**, затем **B()**, и в конце **C()**.

Перед вызовом конструктора **A()** указатель на VFT будет указывать на таблицу класса **A**, перед вызовом **B()** он станет указывать на таблицу класса **B()** и т.д.

Аналогичная ситуация при вызове деструкторов, только указатель будет меняться от таблицы самого младшего класса к самому старшему.

Из этого факта следует правило: в конструкторах и деструкторах нельзя вызывать виртуальные

- Если объявление метода класса в базовом классе начинается с ключевого слова virtual, то это делает функцию виртуальной для базового класса и всех производных из него, включая производные от производных.
- Если виртуальный метод вызывается с помощью ссылки на объект или указателя на объект, программа будет использовать метод, определенный для типа объекта , а не метод, определенный для типа указателя или ссылки – динамическое или позднее связывание.
- При определении класса в качестве базового для наследования в качестве виртуальных функций следует объявлять е методы класса, которые , возможно, придется переопределять в производных классах.

Множественное наследование

2013

C++ позволяет порождать класс из нескольких базовых классов

Когда класс наследует характеристики нескольких классов, используется ***множественное наследование***.

Множественное наследование является мощным инструментом объектно-ориентированного программирования.

Формат определения наследования классом свойств нескольких базовых классов аналогичен формату определения наследования свойств отдельного класса.

class **SubClass**: public *BaseClass1*, private *BaseClass2*
{
// остальная часть определения класса
}

Характеристики:

1. В определении может быть перечислено любое число базовых классов через запятую.
2. Ни один базовый класс не может быть прямо унаследован более одного раза.
3. Каждый базовый класс может быть унаследован как ***public*** или как ***private***; по умолчанию – ***private***.

Порядок вызова конструкторов:

(1) Конструкторы всех виртуальных базовых классов; если их имеется более одного, то конструкторы вызываются в порядке их наследования;

(2) Конструкторы невиртуальных базовых классов в порядке их наследования;

(3) Конструкторы всех компонентных классов.

Последовательность вызова деструкторов будет обратной относительно последовательности вызова конструкторов.

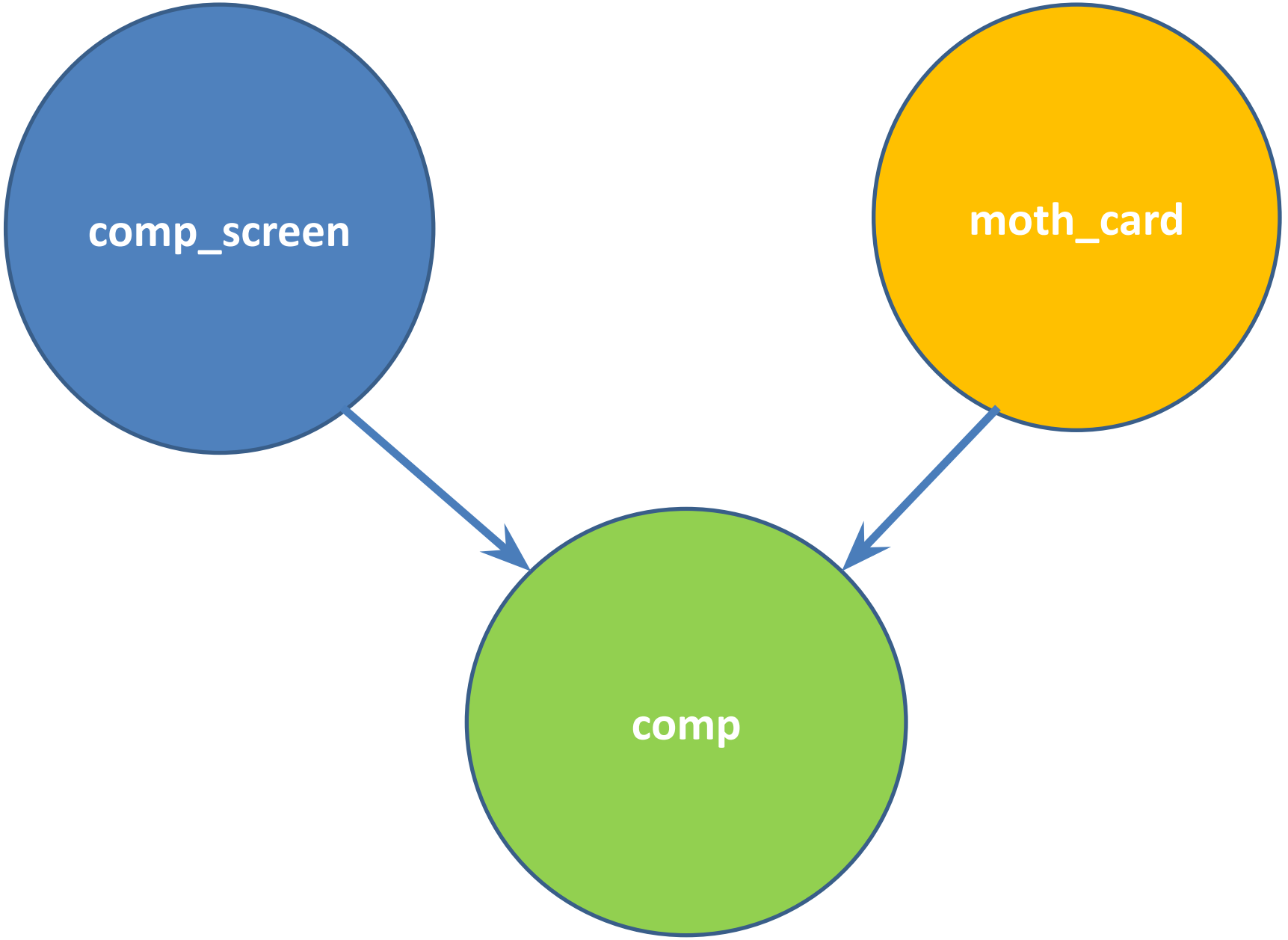
Пример: Есть два класса **comp_screen** и **mother_board**

```
class comp_screen
{
public:
   comp_screen(char *, long, int, int);
   void show_screen(void);
private:
   char type[30] ;
   long colors;
   int x_resolution;
   int y_resolution;
};
```

```
class moth_board
{
```

Создать класс comp:

class comp : *public* comp_screen, *public* moth_board


comp_screen
moth_card
comp

```
#include <iostream.h>
#include <string.h>
class comp_screen
{ public:
   comp_screen(char *, long, int, int);
   void show_screen(void);
  private:
   char type[30];
   long colors;
   int x_resolution;
   int y_resolution;
};
comp_screen::comp_screen(char *type, long colors, int x_res, int y_res)
{ strcpy(comp_screen::type, type);
   comp_screen::colors = colors;
   comp_screen::x_resolution = x_res;
   comp_screen::y_resolution = y_res;
}
void comp_screen::show_screen(void)
```

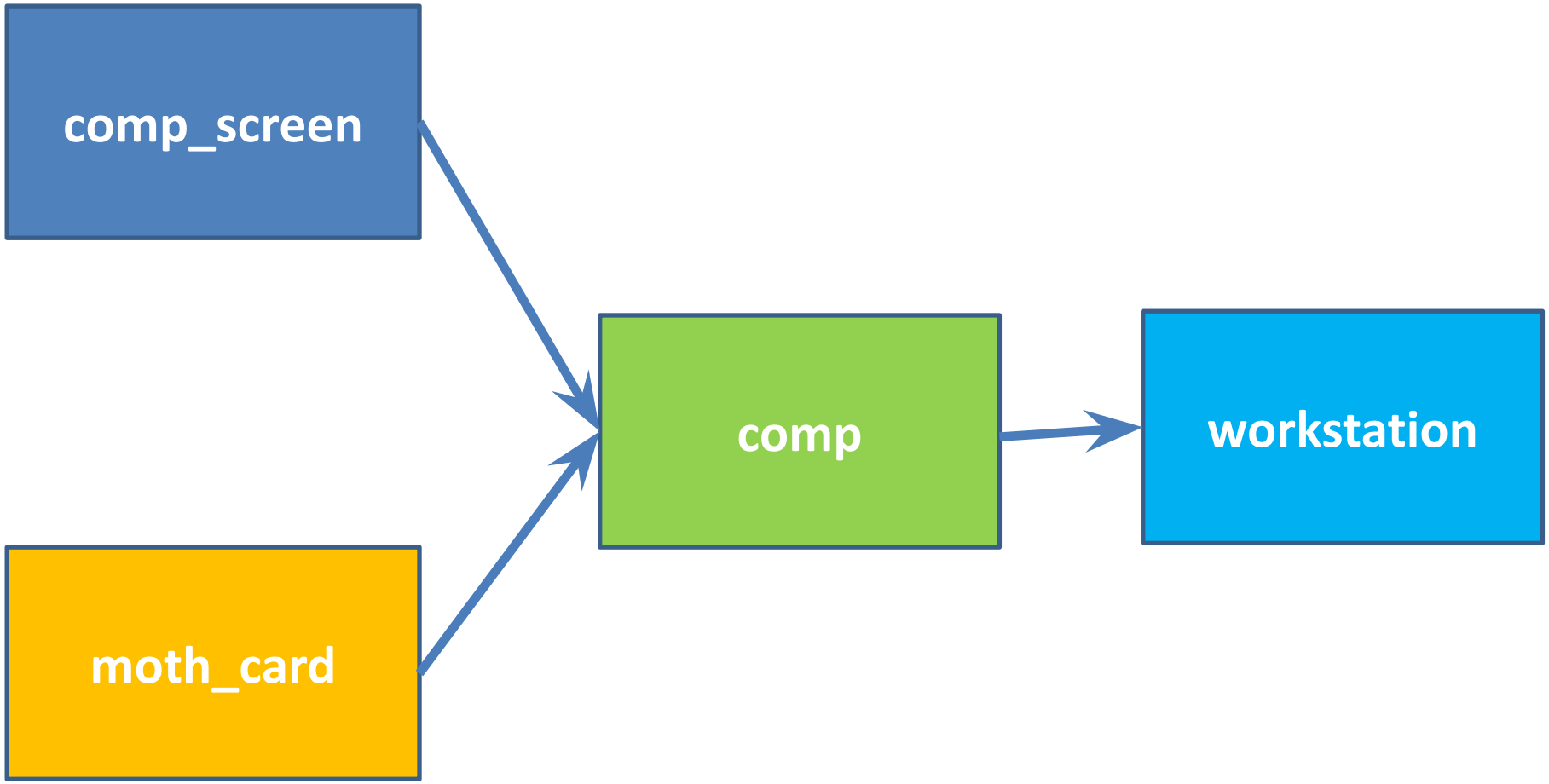
```
class moth_board
{ public:
    moth_board(int, int, int);
    void show_moth_board(void);
  private:
    int processor;
    int speed;
    int RAM;
};
mother_board::moth_board(int processor, int speed, int ram)
{ moth_board::processor = processor;
   moth_board::speed = speed;
   moth_board::RAM = ram;
}


void moth_board::show_moth_board(void)
{
   cout << "Процессор: " << processor << endl;
```

```
void show_comp(void);
private:
  char name [60];
  int hard_disk;
  float flash;
};
comp::comp(char *name, int hard_disk, float flash, char *screen, long colors, int
x_res, int y_res, int processor, int speed, int RAM) : comp_screen(screen, colors,
x_res, y_res), moth_board(processor, speed, ram)
{ strcpy(comp::name, name);
  comp::hard_disk = hard_disk;
  comp::flash = flash; }
void comp::show_comp(void)
{ cout << "Тип: " << name << endl;
  cout << "Жесткий диск: " << hard_disk << "МБайт" << endl;
  cout << "Гибкий диск: " << flash << "МБайт" << endl;
  show_moth_board();
  show_screen(); }
```

```
void main(void)
{
   comp my_pc(“HP", 350, 16.0, “HP-Compaq
BrightView", 16000000, 1280, 800, 2, 2400, 2048);
   my_pc.show_comp();
}
```

Построение иерархии классов

При использовании наследования в C++ для порождения одного класса из другого возможны ситуации, когда порождается класс из класса, который уже, в свою очередь, является производным от некоторого базового класса.


comp_screen
moth_card
comp
workstation

Например, предположим, что необходимо использовать класс comp базовый для порождения класса workstation:

```
class workstation : public comp
{
public:
   workstation (char *operating_system, char *name, int hard_disk, float floppy, char *screen, long colors, int x_res, int  y_res, int processor, int speed, int ram);
   void show_workstation(void);
private:
   char operating_system[60];
};
```

Конструктор класса workstation вызывает конструктор класса comp, который в свою очередь вызывает конструкторы классов сотр_screen и moth_board:

```
workstation::workstation( char *operating_system, char
*name, int hard_disk, float flash, char *screen, long colors,
int    x_res, int y_res, int processor, int speed, int RAM) :
comp (name, hard_disk, flash, screen, colors, x_res, y_res,
processor, speed, RAM)
{
   strcpy(workstation::operating_system,
operating_system);
}
```

- Для порождения класса из нескольких базовых после имени нового класса и двоеточия указываются имена базовых классов, разделяя их запятыми.
- При определении конструктора производного класса необходимо вызвать конструкторы всех базовых классов, передавая им необходимые параметры.
- При порождении классов может случиться так, что используемый базовый класс реально порожден из других базовых классов. Если так, то программа создает иерархию классов. Если вызывается конструктор производного класса, то вызываются также и конструкторы наследуемых классов (последовательно).

Друзья

- Ключевое слово friend сообщет C++, кто является его другом, т. е. другими словами, что другие классы могут обращаться напрямую к его частным элементам.
- Частные элементы класса защищают данные класса, следовательно, следует ограничить круг классов-друзей только теми классами, которым действительно необходим прямой доступ к частным элементам искомого класса.
- C++ позволяет ограничить дружественный доступ определенным набором функций.

Пример

```
class book
{
public:
   book (char *, char *, char *);
   void show_book(void);
   friend librarian;
private:
   char title [60] ;
   char author[60];
   char catalog[60];
};
```

- Использование в программах на C++ друзей позволяет одному классу обращаться к частным элементам другого класса напрямую, используя оператор точку.
- Для объявления одного класса другом (friend) другого класса необходимо внутри определения этого другого класса указать ключевое слово friend, за которым следует имя первого класса.
- После объявления класса другом по отношению к другому классу, все функции-элементы класса-друга могут обращаться к частным элементам этого другого класса.
- Чтобы ограничить количество дружественных методов, которые могут обращаться к частным данным класса, C++ позволяет указать дружественные функции. Для объявления функции-друга следует указать ключевое слово friend, за которым следует прототип функции, которой, собственно, и необходимо обращаться к частным элементам класса.
- При объявлении дружественных функций можно получить синтаксические ошибки, если неверен порядок определений классов. Если необходимо сообщить компилятору, что идентификатор представляет имя класса, который программа определит позже, можно использовать оператор вида:

class class_name;