КОНСТРУИРОВАНИЕ ПРОГРАММ

Лекция № 19 — Наследование и полиморфизм. Шаблоны класса.

Преподаватель: Поденок Леонид Петрович, 505а-5

+375 17 293 8039 (505а-5)

+375 17 320 7402 (ОИПИ НАНБ)

prep@lsi.bas-net.by

ftp://student:2ok*uK2@Rwox@lsi.bas-net.by/

Кафедра ЭВМ, 2021

2021.12.08

Оглавление

Дружественность и наследование

Дружественные функции

Закрытые (private) и защищенные (protected) члены класса в принципе не могут быть доступны снаружи того же класса, в котором они объявлены. Однако это правило не распространяется на «друзей».

Друзья — это функции или классы, объявленные с ключевым словом **`friend`**.

Функция, не являющаяся членом, может получить доступ к закрытым и защищенным членам класса, если она объявлена другом этого класса. Это делается путем включения объявления этой внешней функции в класс и добавления к нему ключевого слова **`friend`**:

```
class Rectangle {
    int width, height;
public:
    Rectangle() {}
    Rectangle(int x, int y) : width(x), height(y) {}
    int area() {return width * height;}
    friend Rectangle duplicate(const Rectangle&); // это не член класса, а друг
};

Rectangle duplicate(const Rectangle& rectangle) {

  Rectangle res;
  res.width  = 2 * rectangle.width;
  res.height = 2 * rectangle.height;
  return res;
}
```

```
#include <iostream>

int main () {

  Rectangle foo;
  Rectangle bar(2, 3);
  foo = duplicate(bar);                 // присваивание перемещением
  std::cout << foo.area() << '\n';
  return 0;
}
```

После компиляции и запуска получаем:

```
24
```

Функция **duplicate** является другом класса **Rectangle**. Следовательно, функция **duplicate** может получить доступ к элементам **width** и **height** (которые являются приватными) различных объектов типа **Rectangle**.

Ни в объявлении **duplicate**, ни в последующем его использовании в **main** функция **duplicate** не считается членом класса **Rectangle**. Она просто имеет доступ к закрытым и защищенным членам класса **Rectangle**.

Типичными случаями использования дружественных функций являются операции, которые проводятся между двумя различными классами, требующие доступа к закрытым и/или защищенным членам обоих классов.

Дружественные классы

Подобно дружественным функциям, дружественный класс — это класс, члены которого имеют доступ к закрытым или защищенным членам другого класса:

```
// дружественный class
#include <iostream>

class Square;           // объявление

class Rectangle {       // определение
    int width, height;
public:
    int area() {return (width * height);}
    void convert(Square a);
};

class Square {
friend class Rectangle;
private:
    int side;
public:
    Square(int a) : side(a) {}
};
```

```
void Rectangle::convert(Square a) {
  width  = a.side;
  height = a.side;
}

int main () {

  Rectangle rect;       // по умолч.
  Square square(4);     // объект
  rect.convert(square);
  std::cout << rect.area();
  return 0;
}
```

После компиляции и запуска:

```
16
```

В этом примере класс **Rectangle** является другом класса **Square**, позволяющим функциям-членам **Rectangle** получать доступ к закрытым и защищенным членам **Square**. Более конкретно, **Rectangle** обращается к переменной-члену **Square::side**, которая описывает сторону квадрата.

В этом примере в начале программы есть пустое объявление класса **Square**. Это необходимо потому что класс **Rectangle** использует класс **Square** (как параметр в преобразовании членов), а **Square** использует **Rectangle** (объявляя его дружественным).

Объявление дружественности не симметрично

Объявление дружественности не симметрично, если не указано иное — в данном примере **Rectangle** считается дружественным по отношению к **Square**, но **Square** не считается дружественным по отношению к **Rectangle**. Следовательно, функции-члены класса **Rectangle** могут получать доступ к защищенным и закрытым членам **Square**, но не наоборот.

Если потребуется, предоставить такой доступ, класс **Square** также может быть объявлен другом **Rectangle**.

Объявление дружественности не транзитивно

Другим свойством «дружеских» отношений является то, что они не транзитивны — друг друга не считается другом, если это не указано явно

Наследование между классами

Классы в C++ могут быть расширены с помощью создания новых классов, которые сохраняют все или часть характеристик и возможностей существующего класса.

Этот процесс известен как наследование — он имеет дело с базовым классом и производным классом. Производный класс наследует члены базового класса, дополнительно к которым он может добавлять свои собственные члены.

Наследование — особенность ЯП, позволяющая описать новый класс на основе уже существующего с частично или полностью заимствующейся функциональностью.

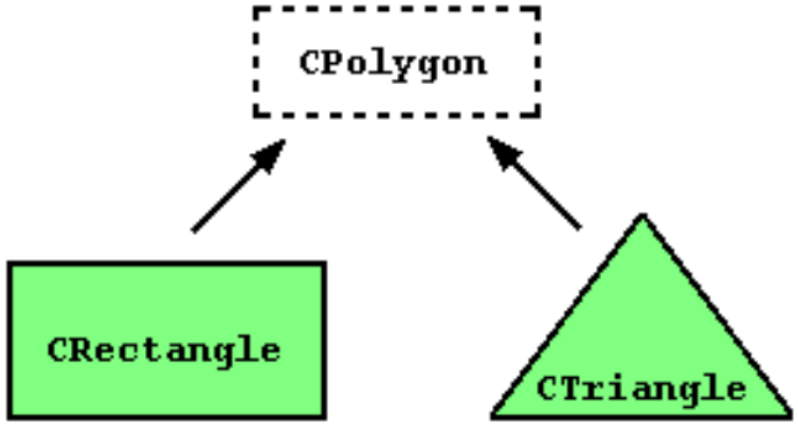
Класс, от которого производится наследование, называется базовым, родительским или суперклассом.

Новый класс называется потомком, наследником, дочерним или производным классом.

Например, создадим набор классов для описания двух видов многоугольников — прямоугольников и треугольников.

Эти два многоугольника имеют определенные общие свойства, такие как значения, необходимые для вычисления их площадей — они оба могут быть описаны высотой и шириной (или основанием).

Данная ситуация может быть представлена классом **Polygon**, из которого мы получаем два других — **Rectangle** и **Triangle**:



Класс **Polygon** будет содержать члены, которые являются общими для обоих типов многоугольников. В нашем случае это *ширина* и *высота*.

Классы **Rectangle** и **Triangle** будут его производными классами со специфическими особенностями, которые отличают один тип многоугольника от другого.

Классы, производные от других, наследуют все *доступные* члены базового класса. Это означает, что если базовый класс содержит член **A**, и мы выводим из него класс, содержащий другой член с именем **B**, производный класс будет содержать как член **A**, так и член **B**.

Отношения наследования двух классов объявляются в производном классе.

Определение производных классов используют следующий синтаксис:

```
class derived_class_name : public base_class_name  {
    /*...*/
};
```

derived_class_name — это имя производного класса;
base_class_name — это имя базового класса (на котором он основан).

Спецификатор доступа **piblic** может быть заменен любым другим спецификатором доступа (**private** или **protect**).

Данный спецификатор доступа ограничивает самый доступный уровень для членов, унаследованных от базового класса — члены с более доступным уровнем наследуются с этим уровнем, а члены с равным или более ограничительным уровнем доступа сохраняют свой ограничительный уровень в производном классе.

```
class Polygon {
protected:                  // разрешает доступ из производного класса
    int width, height;
public:
    void set_values(int a, int b) { width = a; height = b;}
};

class Rectangle : public Polygon {
public:
    int area() { return width * height; }
};

class Triangle : public Polygon {
public:
    int area() { return width * height / 2; }
};

int main () {

  Rectangle rect;
  Triangle  trgl;
  rect.set_values(4, 5);
  trgl.set_values(4, 5);
  std::cout << rect.area() << '\n'; // 20
  std::cout << trgl.area() << '\n'; // 10
  return 0;
}
```

Каждый из объектов классов **Rectangle** и **Triangle** содержит члены, унаследованные от **Polygon** — это **width**, **height** и **set_values**.

Спецификатор защищенного доступа **protected**, используемый в классе **Polygon**, аналогичен **private**. Единственное его отличие заключается в наследовании — члены производного класса могут получить доступ к защищенным (protected) членам, унаследованным от базового класса, но не могут к его закрытым (private) членам.

Объявление членов **width** и **height** защищенными, а не закрытыми, делает эти члены доступными из производных классов **Rectangle** и **Triangle**, а не только из членов **Polygon**.

Если бы они были открытыми, к ним можно было бы получить доступ откуда угодно.

Мы можем суммировать различные типы доступа, в соответствии с которыми функции могут получать доступ следующим образом:

Доступ	public	protected	private
Члены того же класса	да	да	да
Члены производного класса	да	да	нет
Не члены	да	нет	нет

Здесь «не члены» представляют любой доступ извне класса, например, из **main()** или из другого класса или функции.

В приведенном выше примере члены, унаследованные классами **Rectangle** и **Triangle**, имеют те же права доступа, что и в их базовом классе **Polygon**:

```
Polygon::width           // защищенный (protected) доступ
Rectangle::width         // защищенный (protected) доступ
Polygon::set_values()    // открытый (public) доступ
Rectangle::set_values()  // открытый (public) доступ
```

Это связано с тем, что отношение наследования было объявлено с использованием ключевого слова **public** в каждом из производных классов:

```
class Rectangle : public Polygon { /* ... */ }
```

Это ключевое слово **public** после двоеточия (**:**) обозначает наиболее доступный уровень, который члены, унаследованные от базового класса (в данном случае Polygon), будут иметь в производном классе (в данном случае **Rectangle**).

Поскольку **public** является наиболее доступным уровнем, указав это ключевое слово, производный класс унаследует все члены с теми же уровнями, что и в базовом классе.

В защищенном режиме наследования (**protected**) все открытые члены базового класса наследуются в производном классе как защищенные (**protected**)

Если указан наиболее ограничивающий уровень доступа — закрытый (**private**), все доступные члены базового класса наследуются как закрытые (**private**).

Например, если класс **D** является производным от **B**, и мы его выводим как:

```
class D: protected B;
```

то для всех унаследованных от класса **B** членов будет устанавлен ограничительный уровень доступа **protected**, т.е. все члены, которые были **public** в **B**, становятся **protected** в **D**.

Это, тем не менее, не мешает классу **D** объявлять свои открытые члены.

Ограничительный уровень доступа **protected** устанавливается только для унаследованных членов.

Если при наследовании не указан уровень доступа, то для классов, объявленных с ключевым словом **class**, компилятор по умолчанию принимает уровень **private**, а для тех классов, которые объявлены как **struct**, компилятор по умолчанию принимает уровень **public**.

На самом деле, для большинства случаев использования наследования в C++ должно использоваться открытое наследование (**public**).

Когда для базовых классов требуются другие уровни доступа, их обычно лучше представить в виде переменных-членов.

Что наследуется от базового класса?

В принципе, открытый (**public**) производный класс наследует доступ к каждому члену базового класса, за исключением:

- его конструкторов и деструкторов;
- его операторов присваивания (**operator=**);
- его друзей (friends);
- его закрытых (private) членов.

Хотя доступ к конструкторам и деструктору базового класса, как таковой, не наследуется, они автоматически вызываются конструкторами и деструктором производного класса.

Если не указано иное, конструкторы производного класса вызывают конструкторы по умолчанию его базовых классов (то есть конструктор, не имеющий аргументов).

Вызов другого конструктора базового класса также возможен. Для этого используется тот же синтаксис, который используется для инициализации переменных-членов в списке инициализации:

derived_ctor_name **(** *parameters* **) : ***base_ctor_name* **(** *parameters* **) {...}**

```
class Mother {
public:
    Mother()      { std::cout << "Mother: нет параметров\n"; }    // dflt c-tor
    Mother(int a) { std::cout << "Mother: параметр типа int\n"; } // int c-tor
};

class Daughter : public Mother {
public:
    Daughter(int a) { std::cout << "Daughter: параметр типа int\n\n"; }
};

class Son : public Mother {
public:
    Son(int a) : Mother(a) { cout << "Son: параметр типа int\n\n"; }
};

int main () {

  Daughter anny(0);
  Son      pete(0);

  return 0;
}

Mother:  нет параметров
Daughter:  параметр типа int
Mother:  параметр типа int
Son:  параметр типа int
```

Следует обратить внимание на разницу между тем, какой конструктор вызывается для **Mother** при создании нового объекта **Daughter**, и тем, когда создается объект **Son**.

Разница обусловлена различными объявлениями конструктора **Daughter** and **Son**:

```
Daughter(int a)        // ничего не указано: вызов конструктора по умолчанию
Son(int a) : Mother(a) // конструктор указан: вызов этого конструктора
```

Множественное наследование

Класс может наследоваться от более чем одного класса. Это осуществляется указанием нескольких базовых классов, разделенных запятыми, в списке базовых классов (т.е. после двоеточия).

Например, если у программы был специальный класс для печати на экране с именем **`Output`**, и мы бы хотели, чтобы наши классы **`Rectangle`** и **`Triangle`** также наследовали его члены в дополнение к классам **`Polygon`**, мы могли бы написать:

```
class Rectangle : public Polygon, public Output;
class Triangle  : public Polygon, public Output;
```

Далее полный пример:

```
#include <iostream>

class Polygon {
protected:
    int width, height;
public:
    Polygon(int a, int b) : width(a), height(b) {} // конструктор инициализации
};

class Output {
public:
    static void print(int i); // определяется ниже
};
```

```
void Output::print(int i) {
  std::cout << i << '\n';
}

class Rectangle: public Polygon, public Output {
public:
    Rectangle(int a, int b) : Polygon(a, b) {}
    int area() { return width * height; }
};

class Triangle : public Polygon, public Output {
public:
    Triangle(int a, int b) : Polygon(a, b) {}
    int area() { return width * height / 2; }
};

int main () {

  Rectangle rect(4, 5);
  Triangle  trgl(4, 5);

  rect.print(rect.area());      // вызов статической функции через объект
  Triangle::print(trgl.area()); // вызов статической функции через класс
  return 0;
}
```

Полиморфизм

Указатель на базовый класс

Одной из ключевых особенностей наследования классов является то, что указатель на производный класс совместим по типу с указателем на его базовый класс.

Пример о классах **Rectangle** и **Triangle** может быть переписан с использованием указателей с учетом именно этой особенности:

```
#include <iostream>

class Polygon {
protected:
    int width, height;
public:
    void set_values(int a, int b) { width = a; height = b; }
};

class Rectangle : public Polygon {
public:
    int area() { return width * height; }
};

class Triangle: public Polygon {
public:
    int area() { return width * height / 2; }
};
```

```
int main () {

  Rectangle rect;
  Triangle  trig;

  Polygon *prect = &rect;
  Polygon *ptrig = &trig;

  prect->set_values(4, 5);
  ptrig->set_values(4, 5);

  std::cout << rect.area() << '\n';
  std::cout << trig.area() << '\n';

  return 0;
}
```

Функция **main** объявляет два указателя на **Polygon** с именами **prect** и **ptrig**.

Им присваиваются адреса объектов **rect** и **trig**, соответственно, которые являются объектами типа **Rectangle** и **Triangle**.

Такие присвоения являются действительными, поскольку и **Rectangle**, и **Triangle** являются классами, производными от **Polygon**.

Разыменование указателей **prect** и **ptrig** (***prect** и ***ptrig**) также допустимо и позволяет получить доступ к членам этих указанных объектов. Например, следующие два оператора будут эквивалентны представленным в предыдущем примере:

```
prect->set_values(4, 5);
rect.set_values(4, 5);
```

Но поскольку и **prect** и **ptrig** являются указателями на **Polygon**, а не указателями на **Rectangle** или указателями на **Triangle**, оказываются доступны только те члены, которые унаследованы от **Polygon**, но не члены производных классов **Rectangle** и **Triangle**.

Поэтому вышеприведенная программа обращается к функции-члену **area** обоих объектов, используя **rect** и **trgl** напрямую, а не через указатели — указатели на базовый класс не могут получить доступ к функции-члену **area**.

```
error: 'class Polygon' has no member named 'area'
   35 |   prect->area();
```

Доступ к функции-члену **area** с помощью указателей на **Polygon** можно было бы получить, если бы **area** была членом **Polygon**, а не членом его производных классов, но проблема в том, что **Rectangle** и **Triangle** реализуют разные версии **area**, поэтому не существует единой общей версии, которая могла бы быть реализована в базовом классе.

Виртуальные члены

Виртуальный член — это функция-член базового класса, которую можно *переопределить* в производном классе, сохраняя при этом его свойства в отношении вызова посредством ссылок.

Синтаксис функции, которая должна стать виртуальной, заключается в том, что перед ее объявлением стоит ключевое слово **`virtual`**

```
#include <iostream>

class Polygon {
protected:
    int width, height;
public:
    void set_values(int a, int b) { width = a; height = b; }
    virtual int area() { return 0; }
};

class Rectangle : public Polygon {
public:
    int area() { return width * height; }
};

class Triangle : public Polygon {
  public:
    int area() { return (width * height / 2); }
};
```

```
int main () {

  Rectangle rect;
  Triangle  trig;
  Polygon   poly;

  Polygon  *prect = &rect;
  Polygon  *ptrig = &trgl;
  Polygon  *ppoly = &poly;

  prect->set_values(4,5);
  ptrig->set_values(4,5);
  ppoly->set_values(4,5);

  std::cout << prect->area() << '\n'; // все работает как надо
  std::cout << ptrig->area() << '\n'; // все работает как надо
  std::cout << ppoly->area() << '\n'; // все работает как надо

  return 0;
}
20
10
0
```

— В этом примере все три класса **Polygon**, **Rectangle** и **Triangle** имеют одинаковые члены **width**, **height**, а также функции **set_values()** и **area()**.

Функция-член **area** в базовом классе была объявлена как виртуальная, потому что она *переопределяется* в каждом из производных классов.

Невиртуальные (*non-virtual*) члены также могут быть переопределены в производных классах, однако в этом случае невиртуальные члены производных классов не могут быть доступны через указатель на базовый класс — т.е., если в приведенном выше примере ключевое слово **virtual** удаляется из объявления функции-члена **area**, все три вызова **area** вернут значение ноль, потому что во всех случаях вместо этого будет вызываться версия базового класса.

Таким образом, ключевое слово **virtual** позволяет соответствующим образом вызывать член производного класса с тем же именем, что и в базовом классе, используя указатель, имеющий тип указателя на базовый класс, но который указывает на объект производного класса.

Класс, который объявляет или наследует виртуальную функцию, называется **полиморфным классом**.

Следует обратить внимание, что, несмотря на виртуальность одного из его членов, **Polygon** является обычным классом, для которого даже был создан объект **poly**, со своим собственным определением члена-функции **area**, которая всегда возвращает 0.

Если нет определения **area( )** для класса **Polygon**, получаем ошибку при сборке:

```
/usr/bin/ld: /tmp/ccGcQq9z.o: in function `Polygon::Polygon()':
4.cpp:(.text._ZN7PolygonC2Ev[_ZN7PolygonC5Ev]+0x9): undefined reference to `vtable for
Polygon'
```

Абстрактные базовые классы

Абстрактные базовые классы очень похожи на класс **Polygon** из предыдущего примера.

Это такие классы, которые могут использоваться только в качестве базовых классов, и поэтому им разрешено иметь виртуальные функции-члены без определения, известные как *чисто виртуальные функции*.

Согласно синтаксису определение таких функций должно быть заменено на **= 0** (знак равенства и ноль).

Абстрактный базовый класс **Polygon** может выглядеть следующим образом:

```
class Polygon {
protected:
    int width, height;
public:
    void set_values(int a, int b) { width = a; height = b; }
    virtual int area() = 0;
};
```

Следует заметить, что **area** не имеет определения — оно заменено на **= 0**, что делает ее чисто виртуальной функцией.

Классы, которые содержат хотя бы одну чисто виртуальную функцию, называются ***абстрактными базовыми классами***.

Абстрактные базовые классы не могут быть использованы для создания объектов — компилятор на сей счет говорит следующее:

```
4.cc: In function 'int main()':
4.cc:26:13: error: cannot declare variable 'poly' to be of abstract type 'Polygon'
   26 |   Polygon   poly;
      |             ^~~~
4.cc:3:7: note:   because the following virtual functions are pure within 'Polygon':
    3 | class Polygon {
      |       ^~~~~~~
4.cc:9:17: note:     'virtual int Polygon::area()'
    9 |     virtual int area() = 0;
      |                 ^~~~
```

То есть, вышеприведенная абстрактная версия базового класса **Polygon** не может использоваться для объявления таких объектов, как:

```
Polygon poly;   // недопустимо если Polygon является абстрактным классом
```

Но абстрактный базовый класс не совсем бесполезен. Его можно использовать для создания указателей на его тип и, таким образом, использовать его полиморфные способности.

Например, следующие объявления указателя будут действительными

```
Polygon *prect;
Polygon *ptrig;
```

и могут быть нормально разыменованы, если они указывают на объекты производных (неабстрактных) классов. Ниже весь пример:

```
#include <iostream>

class Polygon {
protected:
    int width, height;
public:
    void set_values(int a, int b) { width = a; height = b; }
    virtual int area(void) = 0;
};

class Rectangle: public Polygon {
public:
    int area(void);
};

class Triangle: public Polygon {
public:
    int area(void) { return (width * height / 2); }
};

int Rectangle::area(void) {
    return width * height;
}

int area(void) {                  // просто функция
    return width * height / 2;
}
```

```
int main () {

  Rectangle rect;
  Triangle  trig;

  Polygon *prect = &rect;
  Polygon *ptrig = &trig;

  prect->set_values(4, 5);
  ptrig->set_values(4, 5);

  std::cout << prect->area() << '\n';
  std::cout << ptrig->area() << '\n';
  return 0;
}
```

После компиляции и запуска:

```
20
10
```

В этом примере к объектам разных, но родственных типов обращаются с использованием указателя единственного типа **`Polygon *`**, и каждый раз вызывается правильные функции-члены потому, что они объявлены как виртуальные в базовом классе.

Член абстрактного базового класса, например **Polygon**, может использовать специальный указатель **this** для доступа к соответствующим виртуальным членам, даже если сам **Polygon** не имеет реализации для данной функции:

```
#include <iostream>

class Polygon {
protected:
    int width, height;
public:
    void set_values(int a, int b) { width = a; height = b; }
    virtual int area() = 0;
    void printarea() { std::cout << this->area() << '\n'; }
};

class Rectangle : public Polygon {
public:
    int area(void) { return (width * height); }
};

class Triangle : public Polygon {
public:
    int area(void) { return (width * height / 2); }
};
```

```
int main () {
  Rectangle rect;
  Triangle  trgl;

  Polygon *prect = &rect;
  Polygon *ptrig = &trig;

  prect->set_values(4, 5);
  ptrig->set_values(4, 5);

  prect->printarea();
  ptrig->printarea();

  return 0;
}
```

После компиляции и запуска:

```
20
10
```

Виртуальные члены и абстрактные классы предоставляют полиморфные характерные особенности языка C++, которые наиболее полезны для объектно-ориентированных проектов.

Приведенные выше примеры являются очень простыми вариантами использования, но такие функции могут быть применены к массивам объектов или динамически размещаемым объектам.

Ниже пример, который совмещает некоторые функции, такие как динамическая память, инициализаторы конструктора и полиморфизм:

```
#include <iostream>

class Polygon {
protected:
    int width, height;
public:
    Polygon(int a, int b) : width(a), height(b) {}     // init c-tor
    virtual int area(void) = 0;                        // pure virtual
    void printarea( ) { cout << this->area( ) << '\n'; }
};

class Rectangle : public Polygon {
public:
    Rectangle(int a,int b) : Polygon(a, b) {}
    int area( ) { return width * height; }
};

class Triangle : public Polygon {
public:
    Triangle(int a,int b) : Polygon(a, b) {}
    int area( ) { return width * height / 2; }
};
```

```
int main () {

  Polygon *prect = new Rectangle(4, 5);
  Polygon *ptrig = new Triangle(4, 5);

  prect->printarea();
  ptrig->printarea();

  delete prect;
  delete ptrig;

  return 0;
}
```

После компиляции и запуска:

```
20
10
```

Отметим, что указатели **ppoly**

```
Polygon *prect = new Rectangle(4, 5);
Polygon *ptrig = new Triangle(4, 5);
```

объявляются, как имеющие тип «указатель на **Polygon**», а выделенные объекты были объявлены с прямым типом производных классов **Rectangle** и **Triangle**.

Статические члены (повторение)

Класс может содержать статические члены — либо данные, либо функции.

Статический член данных класса также известен как «переменная класса», поскольку существует только одна общая переменная для всех объектов этого же класса, и она имеет одно и то же значение для всех объектов этого же класса.

Например, такой член может использоваться для переменной в классе, которая может содержать счетчик с количеством объектов этого класса, которые выделены в данный момент, как в следующем примере:

```
class Dummy {
  public:
    static int n;      // статический член класса
    Dummy() { n++; }; // конструктор
};

int Dummy::n = 0;      // инициализация статического члена за пределами класса
```

Фактически, статические члены имеют те же свойства, что и переменные, не являющиеся членами класса, но они имеют область видимости класса.

По этой причине и во избежание их объявления несколько раз они не могут быть инициализированы непосредственно в классе, но должны быть инициализированы где-то за его пределами.

Поскольку статический член класса является общей переменной для всех объектов одного и того же класса, на него можно ссылаться как на член любого объекта данного класса и даже напрямую с использованием имени класса (это допустимо только для статических членов):

```
cout << a.n;
cout << Dummy::n;
```

Эти два вызова относятся к одной и той же переменной — статической переменной **n** внутри класса **Dummy**, общей для всех объектов этого класса.

Она похожа на переменную, не являющуюся членом, но имеет имя, которое требует обращения, как к члену класса (или объекта).

Статические функции-члены

Классы также могут иметь статические функции-члены. Они представляют собой тоже самое — члены класса, которые являются общими для всех объектов этого класса, действуют точно как функции, не являющиеся членами, но доступ к ним осуществляется как к членам класса.

Поскольку они похожи на функции, не являющиеся членами, они не могут получить доступ к нестатическим членам класса (ни к переменным-членам, ни к функциям-членам).

Они также не могут использовать ключевое слово **this**.

const функции-члены

Когда объект класса квалифицируется как **const** объект:

```
const MyClass myobject;
```

доступ к его элементам данных *извне* класса ограничен доступом только для чтения, как если бы все его элементы данных были **const** для всех, кто обращается к ним извне класса.

Следует обратить внимание, что по-прежнему вызывается конструктор и он может инициализировать и изменять эти элементы данных:

```
// constructor on const object
#include <iostream>
using namespace std;

class MyClass {
public:
    int x;
    MyClass(int val) : x(val) {}
    int get() {return x;}
};

int main() {

    const MyClass foo(10);
    // foo.x = 20;           // недопустимо: x не может модифицироваться
    cout << foo.x << '\n';  // ok: член данных x можно читать
    return 0;
}
```

Функции-члены объекта, квалифицированого как **const** могут вызываться только в том случае, если они сами указаны как **const** члены.

В приведенном выше примере, член **get**, который не указан как **const**, для объекта **foo** вызван быть не может.

Чтобы указать, что функция-член является константной, за прототипом функции после закрывающей скобки определения ее параметров должно следовать ключевое слово **const**:

```
int get() const {return x;}
```

Следует обратить внимание, что **const** может использоваться для определения типа, возвращаемого функцией-членом.

Этот **const** не является тем, который указывает член как **const**.

Оба являются независимыми и семантически разными, соответственно, и расположены в разных местах в прототипе функции:

```
int get() const {return x;}         // const функция-член
const int& get() {return x;}        // функция-член возвращает const&
const int& get() const {return x;} // const функция-член возвращает const&
```

Функции-члены, определенные как **const**, не могут изменять нестатические элементы данных, а также не могут вызывать другие нестатические функции-члены.

По сути, **const**-члены не должны изменять состояние объекта.

const-объекты ограничены в отношении доступа только к функциям-членам, помеченным как **const**, но неконстантные объекты не ограничены и, таким образом, могут обращаться как к **const**, так и к неконстантным функциям-членам.

Большинство функций, принимающих в качестве параметров классы, фактически принимают их по константной ссылке, и, таким образом, эти функции могут получить доступ только к своим константным членам:

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

class MyClass {
    int x;
public:
    MyClass(int val) : x(val) {}
    const int& get() const {return x;}
};

void print(const MyClass& arg) {
    cout << arg.get() << '\n';
}

int main() {

    MyClass foo(10);
    print(foo);
    return 0;
}
```

Если бы в этом примере **get** не был указан как **const**-член, вызов функции-члена **arg.get()** в функции **print** был бы невозможен, поскольку **const**-объекты доступны только для **const** функций-членов.

Функции-члены могут быть перегружены по своей константности — т.е. класс может иметь две функции-члена с одинаковыми сигнатурами, за исключением того, что одна является константной, а другая нет.

В этом случае константная версия вызывается только тогда, когда сам объект является **const**, а неконстантная версия вызывается, когда сам объект неконстантный.

```
// overloading members on constness
#include <iostream>
using namespace std;

class MyClass {
    int x;
public:
    MyClass(int val) : x(val) {}
    const int& get() const {return x;}
    int& get() {return x;}
};

int main() {
    MyClass foo (10);
    const MyClass bar (20);
    foo.get() = 15;          // ok: get() returns int&
  // bar.get() = 25;         // not valid: get() returns const int&
    cout << foo.get() << '\n';
    cout << bar.get() << '\n';
    return 0;
}
```

Шаблоны функций (повторение)

Перегруженные функции могут иметь одно и то же определение. Например:

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

int sum(int a, int b) {
  return a + b;
}

double sum(double a, double b) {
  return a + b;
}

int main() {

  cout << sum(10, 20) << '\n';
  cout << sum(1.0, 1.5) << '\n';

  return 0;
}
```

В данном случае функция **sum** перегружена разными типами параметров, но с одинаковым телом.

Функция **sum** может быть перегружена для большого количества типов, и для всех из них может быть смысл иметь одинаковое тело.

В таких случаях C++ предоставляет возможность определять функции с универсальными типами. Такие определения называются *шаблонами функций*.

При определении шаблона функции используется тот же синтаксис, что в случае определения обычной функции, за исключением того, что ему предшествует ключевое слово **template** и ряд параметров шаблона, заключенных в угловые скобки **<>**:

```
template < template-parameters-list > function-declaration
```

Параметры шаблона «*template-parameter-list*» представляют собой последовательность параметров, разделенных запятыми.

Эти параметры могут быть обобщенными шаблонными типами, указанными с помощью ключевого слова **class** или **typename**, за которым следует идентификатор. Этот идентификатор после такого включения можно использовать в объявлении функции, как если бы он был обычным типом.

Например, общая функция **sum** может быть определена как:

```
template <class T>
T sum (T a, T b) {
  return a + b;
}
```

Не имеет значения, указан ли в списке аргументов шаблона универсальный тип с ключевым словом **`class`** или ключевое слово **`typename`**.

```
template <class T>
T sum (T a, T b) {
  return a + b;
}
```

В приведенном коде объявление **T** (универсальный тип в параметрах шаблона, заключенных в угловые скобки) позволяет использовать **T** в любом месте определения функции, как и любой другой тип.

T может использоваться в качестве типа для параметров, в качестве типа возвращаемого значения или для объявления новых переменных этого типа.

Во всех случаях **T** представляет обобщенный тип, который будет определен в момент «воплощения» шаблона (instantiation).

«Воплощение» шаблона — это применение шаблона для создания функции с использованием определенных типов или значений для параметров шаблона.

Это делается путем вызова шаблона функции с тем же синтаксисом, что и при вызове обычной функции, но с указанием аргументов шаблона, заключенных в угловые скобки:

```
name<template-arguments>(function-arguments)
```

Например, шаблон функции суммы, определенный выше, может быть вызван с помощью:

```
x = sum<int>(10, 20);
```

Функция **sum<int>** является лишь одним из возможных воплощений шаблона функции **sum**. В этом случае, используя **int** в качестве аргумента шаблона в вызове, компилятор автоматически создает версию суммы, где каждое вхождение **T** заменяется на **int**, как если бы оно было определено как:

```
int sum(int a, int b) {
  return a+b;
}
```

Шаблоны класса

Аналогичным образом тому, как мы можем создавать шаблоны функций, мы также можем создавать шаблоны классов, позволяя классам иметь члены, которые используют параметры шаблона в качестве типов. Например:

```
template <class T>
class mypair {
    T values[2];
public:
    mypair(T first, T second) {
        values[0] = first;
        values[1] = second;
    }
};
```

Вышеуказанный класс служит для хранения двух элементов любого допустимого типа. Например, если мы хотим объявить объект этого класса для хранения двух целочисленных значений типа **int** со значениями 115 и 36, мы должны написать:

```
mypair<int> myobject(115, 36);
```

Этот же класс также можно использовать для создания объекта для хранения любого другого типа, такого как:

```
mypair<double> myfloats (3.0, 2.18);
```

Единственной функцией-членом в предыдущем шаблоне класса является конструктор, и он был определен как встроенный в самом определении класса.

В том случае, если функция-член определена вне определения шаблона класса, ей должен предшествовать префикс шаблона **<...>**:

```
#include <iostream>

template <class T>
class mypair {
    T a, b;
public:
    mypair(T first, T second) {a = first; b = second;}
    T getmax();              // определена вне класса
};

template <class T>           // префикс шаблона функции
T mypair<T>::getmax() {      // определение
    T retval;
    retval = a > b? a : b;
    return retval;
}

int main () {

    mypair<int> myobject(100, 75);
    std::cout << myobject.getmax();
    return 0;
}
```

Синтаксис определения функции-члена **getmax()**:

```
template <class T>
T mypair<T>::getmax()
```

В этом объявлении есть три **T**.

Первый — это параметр шаблона.

Второй **T** относится к типу, возвращаемому функцией.

Третий **T** (тот, что в угловых скобках) также является обязательным — он указывает, что параметр шаблона этой функции также является параметром шаблона класса **mypair**.

Специализация шаблонов

Если в качестве аргумента шаблона передается определенный тип, можно определить другую реализацию для шаблона. Это называется *специализация шаблона*.

Например, предположим, что у нас есть очень простой класс **mycontainer**, который может хранить один элемент любого типа и который имеет только одну функцию-член, называемую **increase**, которая увеличивает его значение.

```
// template specialization
#include <iostream>
using namespace std;

// class template:
template <class T>
class mycontainer {
    T element;
public:
    mycontainer(T arg) {element = arg;}
    T increase() {return ++element;}
};
```

Но мы обнаруживаем, что для элемента типа **char**, было бы удобнее иметь совершенно другую реализацию с функцией-членом, например, преобразующей символ в верхний регистр, поэтому мы объявляем *специализацию* шаблона класса для этого конкретного типа:

```
// class template specialization:
template <>
class mycontainer <char> {
    char element;
public:
    mycontainer(char arg) {element = arg;}
    char uppercase() {
      if ((element >= 'a')&&(element<='z'))
      element += 'A' - 'a';
      return element;
    }
};

int main () {
    mycontainer<int> myint (7);
    mycontainer<char> mychar ('j');
    cout << myint.increase() << endl;
    cout << mychar.uppercase() << endl;
    return 0;
}
```

Синтаксис, используемый для специализации шаблона класса имеет вид:

```
template <> class mycontainer <char> { ... };
```

Мы предваряем имени класса **template <>** с пустым списком параметров.

Это потому, что все типы известны, и для специализации не требуются аргументы шаблона, но, тем не менее, это специализация шаблона класса, и, следовательно, его необходимо отметить как таковой.

Но более важным, чем этот префикс, является параметр специализации **<char>** после имени шаблона класса.

Этот параметр специализации сам определяет тип, для которого специализируется класс шаблона (**char**).

Следует обратить внимание на различия между общим шаблоном класса и специализацией:

```
template <class T> class mycontainer { ... };
template <> class mycontainer <char> { ... };
```

Первая строка — это общий шаблон, а вторая — специализация.

Когда мы объявляем специализации для шаблонного класса, мы также должны определить все его члены, даже те, которые идентичны универсальному классу шаблона, потому что «наследования» членов от универсального шаблона к специализации нет.