Лекция

Функции — члены структуры

При описании параметров некоторых объектов, например, человека, достаточно занести только конкретное значение, например, имя, фамилия, рост и так далее. Но некоторые параметры требуют выполнения процедур на момент их запроса, как-то: давление, температура и так далее.

Поэтому, как правило, описав новую структуру, приходится создавать набор функций, работающих с элементами этой структурой. Например, можно описать структуру трехмерной точки в декартовой системе или вектор.

Пример.

```
struct _3d {
  double x, y, z;
 } vect;
```

Было бы логично написать функцию, которая будет вычислять модуль вектора (длину).

Пример.

```
double mod (_3d &V);
```

Это традиционный путь для программ как на языке С так и других. Описывается функция, вычисляющая модуль вектора, который передается ей через формальные параметры. Между тем, модуль — характеристика, присущая каждому вектору. Поэтому, будет логично, совместить описание вектора и его длину, то есть вектор будет, как бы, сам возвращать свою длину.

Для того чтобы функция стала членом структуры, достаточно поместить ее описание внутрь описания структуры.

Пример.

```
struct _3d {
  double x, y, z;
  double mod ();
 } vect;
```

Здесь функция `mod()` выступает как *член структуры*.

При описании реализации функции надо после типа возвращаемого значения указать *имя шаблона структуры*, членом которого является данная функция, и имя функции, отделив их знаком “**::**” (два двоеточия).

Пример.

```
double _3d::mod ()
 {
  return sqrt (x * x + y * y + z * z);
 }
```

Если текст функции короткий, то реализацию функции можно поместить непосредственно внутрь описания структуры.

Пример.

```
struct _3d {
  double x, y, z;
  double mod () { return sqrt (x * x + y * y + z * z); }
} vect;
```

В этом случае можно опустить имя структуры, а сама функция будет считаться **inline**.

В качестве преимуществ такой структуры отметим, что такая *функция обращается* к *своим* переменным (членам структуры) запросто по имени. То есть, если функция не использует посторонние данные (не члены структуры) для достижения результата и не возвращает результат через список параметров, то список параметров может быть пустым.

Помимо этого необходимо отметить, что каждая функция шаблона структуры представлена в *единственном* экземпляре и получает один скрытый параметр — указатель на ту

переменную–структуру, для которой она вызвана. К этому указателю можно обратиться непосредственно по его имени — this.

Обращаться к функциям–членам классам следует также как и к переменным: указать имя структуры и функции через точку; указать имя указателя на структуру и функции через знак ‘–>’.
Пример.

```
_3d a, *b;
double lenA, lenB, lenAB;
…
b = new _3d;
…
a.x = 5.; …
b->x = 6.; …
lenA = a.mod();
lenB = b->mod();
lenAB = a.mod() + b->mod();
```

Здесь показано, как использовать поля и функции переменной типа структуры и указателя на структуру.

Понятие класса

Класс сложный тип данных, представляющий собой группу данных и функций для работы с этими данными. Ближайшим родственником (прародителем) является тривиальная структура. Упрощенно можно считать, что структура, наделенная *механизмом наследования*, становится *классом*. Но класс обладает рядом свойств, которые позволяют их использовать в рамках концепции *объектно-ориентированного программирования* (ООП).

Механизм наследования позволяет вновь создаваемым классам данных наследовать свойства уже существующих классов. Именно способность передавать и получать свои свойства по наследству отличает класс от структуры.

Синтаксически класс описывается так же, как и структура, только имеет ключевое слово class. Все что сказано о структуре, справедливо и для классов. Ранее созданную структуру легко преобразовать в класс.
Пример.

```
class _3d
 {
  public :
    double x, y, z;
    double mod ();
 };
double _3d::mod ()
 {
  return sqrt (x * x + y * y + z * z);
 }
```

Здесь описан класс `_3d` и описана реализация функции `mod()`, входящей в состав класса.

Для классов принято специальное название переменных и функций, входящих в состав класса. Переменные являющиеся членами класса называют *полями*, а функции — *методами*.

Поля и методы класса размещают в трех разделах, различающихся возможностью доступа к ним. В раздел `private:` входят поля и методы доступные только методам самого этого класса. В раздел `protected:` входят поля и методы доступные наследникам, а в раздел `public:` общедоступные поля и методы.

Поля и методы могут быть объявлены вне разделов (до разделов). Тогда эти поля и методы компилятор воспринимает как ???

Переменные программы, тип которых есть класс, называются *объектами*.

В простейшем случае описание объекта, переменной типа класс, аналогично описанию структуры; обращение к полям и методам класса, такое же, как и к переменным и функциям, являющихся членами структуры, то есть имя объекта и поля (или метода) указываются через точку.

Пример.

```
void main (void)
{
 _3d point;
 double len;
 point.x = 5.0; point.y = 6.0; point.z = 7.0;
 len = point.mod ();
 cout << len;
}
```

Объектно-ориентированное программирование. Базовые понятия

Базовые понятия объектно-ориентированного программирования это три краеугольных камня ООП:

- *Инкапсуляция* (encapsulation) — механизм ООП позволяющий скрыть описание реализации объекта от использующих его модулей.

- *Наследование* (inheritance) — механизм ООП позволяющий объявить новый тип данных, который является расширением существующего. Новый объект можно создать на базе существующего, при этом сохраняются все свойства и методы предка.
- *Полиморфизм* (polymorphism) — механизм ООП позволяющий вносить изменения в выполнение одноименных методов объектов. Объекты могут иметь методы с одинаковыми названиями, но для различных объектов они могут обрабатываться по-разному.

Инкапсуляция. Под инкапсуляцией понимается хранение в одной структуре как данных (констант и переменных), так и функций их обработки (методов). Доступ к отдельным частям класса регулируется с помощью специальных ключевых слов `public:` (открытая часть), `private:` (закрытая часть) и `protected:` (защищенная часть).

Если спецификатор доступа не указан, то секция, непосредственно следующая за открывающей скобкой, по умолчанию считается `private`.

Методы, расположенные в открытой части, формируют интерфейс класса и могут свободно вызываться всеми пользователями класса. Считается, что переменные-члены класса не должны находиться в секции public, но могут существовать интерфейсные методы, позволяющие читать и модифицировать значение каждой переменной. Доступ к закрытой секции класса возможен только из его собственных методов, а к защищенной — также из методов классов-потомков.

Несмотря на непривычность слова *инкапсуляция*, это просто связывание полей и методов в одну структуру (складывание их в одну "капсулу"). Это удобно, хотя и без остальных двух принципов никакого нового *качества* программирования не дает. Действительно, если объединить данные хотя бы с алгоритмами доступа к ним, то программист окажется независимым от представления данных в объекте: объект становится абстракцией представления своих собственных данных. В более общем случае объекту можно приписать свойства (методы), абстрагирующие не только представление, но и придающие объекту другие свойства, к примеру, способность отображаться на экране монитора или другом устройстве. Теоретически принцип инкапсуляции применим как к отдельным объектам, так и к классам. В случае классов с методами объединяются не сами данные, а *структуры данных*, и объединение с конкретными данными происходит в момент создания объектов данного класса. На практике же большинство языков ООП

просто не позволяют создавать объекты, не создав предварительно класса.

На современном этапе развития ООП под *инкапсуляцией* принято понимать сокрытие данных за методами класса. Современный стиль программирования классов подразумевает, что доступ к полям класса, как для записи в них, так и для их чтения, должен осуществляться через специальные методы.

Иерархия классов. В C++ класс выступает в качестве шаблона, на основе которого создаются объекты. От любого класса можно породить один или несколько подклассов, в результате чего сформируется иерархия классов. Родительские классы обычно содержат методы более общего характера, тогда как решение специфических задач поручается производным классам.

Наследование. Под наследованием понимают передачу данных и методов от родительских классов производным. Если класс наследует свои атрибуты (то есть поля и методы) от одного родительского класса, то такое наследование называется одиночным. Если же атрибуты наследуются от нескольких классов, то говорится о множественном наследовании. *Наследование* является важнейшей концепцией программирования, поскольку позволяет многократно использовать одни; и те же классы, не переписывая их, а лишь подстраивая для решения конкретных задач и расширяя их возможности.

Этот принцип относится только к классам объектов. Наследование означает, что каждый *объект* может иметь *наследников*. Каждый наследник (потомок) будет обладать всеми полями и методами своего *предка*. Кроме того, как правило, классы–наследники совместимы по типу со своими предками (к сожалению, это справедливо не для всех ОО языков).

Примечание. Наследование бывает двух видов:

одиночное – когда каждый класс имеет одного и только одного предка;

множественное – когда каждый класс может иметь любое количество предков.

Множественное наследование обладает более мощными возможностями: в одном классе-наследнике объединяются свойства (поля и методы) множества различных классов. К примеру один из предков может рисовать себя, а другой – производить вычисления. Представитель их наследника смогут делать и то, и другое. Из-за сложности множественное

наследование очень редко используется. Но использование классов в качестве полей другого класса позволяет достигнуть тех же результатов, что и при множественном наследовании.

Полиморфизм и виртуальные функции. Другая важная концепция ООП, связанная с иерархией классов, заключается в возможности послать одинаковое сообщение сразу нескольким классам в иерархии, предоставив им право выбрать, кому из них надлежит его обработать. Это называется полиморфизмом. Методы, содержащиеся в разных классах одной иерархии, но имеющие общее имя и объявленные с ключевым словом `virtual`, называются *виртуальными*. Благодаря полиморфизму можно делать в программе запрос к объекту, даже если тип его не известен заранее. В C++ эта возможность реализуется за счет подсистемы позднего связывания, под которым понимается динамическое определение адресов функций во время выполнения программы в противоположность традиционному статическому (раннему) связыванию, осуществляемому во время компиляции. В процессе связывания имена функций заменяются их адресами. При вызове виртуальных функций используется специальная таблица адресов функций, называемая виртуальной таблицей. Она инициализируется в ходе выполнения программы в момент создания объекта конструктором класса. Роль конструктора заключается в том, чтобы связать виртуальную функцию с правильной таблицей адресов. Во время компиляции адрес виртуальной функции не известен, но известна ячейка виртуальной таблицы, где этот адрес будет записан во время выполнения программы.

Принцип *полиморфизма* неразрывно связан с наследованием и гласит, что каждый класс-наследник может обладать не только свойствами, унаследованными от предка, но и своими собственными. В частности, свойства предка могут быть *перекрыты* наследником — на место свойств предка могут быть подставлены свойства наследника. Существование принципа полиморфизма является естественным следствием существования принципа наследования: наследование без изменения набора свойств не имеет смысла. Кроме того, без полиморфизма невозможно реализовать объединение различных объектов (классов) по некоторому набору свойств (невозможно абстрагироваться от части свойств объектов), а без этого теряется весь смысл подхода.

Конструкторы и деструкторы

Создание объекта некоторого класса может быть достаточно сложным процессом. Поэтому в C++ предусмотрены возможности явного описания порядка создания и уничтожения объектов данного класса. Процедуры *создания* объекта — *конструкторы*, уничтожения — *деструкторы*.

Конструкторы автоматически вызываются при описании объекта, *деструкторы* — при выходе из блока, в котором этот объект был описан.

Класс может обладать несколькими конструкторами. *Конструкторы* класса в языке C++ имеют *имена*, совпадающие с *именем класса*. Конструкторы одного класса между собой различаются аргументами. То есть, как и обыкновенные функции, как простые методы класса, так и конструкторы могут перекрывать друг друга. Поэтому они все имеют одно имя (но конструкторы имеют специфическое имя — имя класса) но различный набор формальных параметров.

Деструктор может быть только один и имеет имя, совпадающее с именем класса, которому предшествует символ "~".

Примечание. Ни **конструкторы**, ни **деструкторы** НЕ ИМЕЮТ описания типа!

Пример.

```
class _3d
 {
   double x, y, z;
  public :
   _3d (){x = 0; y = 0; z = 0;};
   _3d (const _3d &a)
                    {x = a.x; y = a.y; z = a.z;};
   _3d (double &X, double &Y, double &Z)
                           {x = X; y = Y; z = Z;};
   _3d (double C);
   ~_3d (){};
   double mod ();
 };
_3d::_3d (double C)
 {
  x = C; y = C; z = C;
 };
double _3d::mod ()
 {
```

```
  return sqrt (x * x + y * y + z * z);
 };
```

В данном примере описан класс `_3d`. В описании приведены четыре конструктора. Реализация трех конструкторов (описание тела функции-метода) описано непосредственно в интерфейсной части описания класса. А реализации одного конструктора и одного метода `mod()` описаны отдельно.

Язык C++ позволяет не описывать ни конструкторы, ни деструкторы класса. В таком случае компилятор генерирует необходимые конструкторы и деструкторы. Но большой вопрос: будет ли конструктор или деструктор, порожденный автоматически, выполнять необходимые действия? Поэтому, как правило, программисту приходится описывать все конструкторы, чтобы гарантировать, что каждый конструктор будет выполнять действия необходимые для порождения нового экземпляра класса, то есть объекта. Более редкий случай, когда необходимо описывать деструктор. Деструктор нужен, например, для освобождения динамической памяти, занятой объектом. Если же таких сложных и специфичных задач не предстоит выполнять при уничтожении объекта, то деструктор можно и не описывать. В данном примере тело функции деструктора пусто; деструктор здесь можно было и не описывать. Но описание приведено для демонстрации правил описания деструктора.

При описании переменных типа класс автоматически вызывается конструктор этого класса. Какой из конструкторов класса будет вызван, зависит от списка фактических параметров указанных при описании переменной-объекта.

При разработке класса программист может создавать произвольное количество конструкторов этого класса с произвольным набором входным параметров. Но язык C++ предусматривает несколько специальных конструкторов, которые он вызывает по умолчанию в определенных ситуациях. Это следующие конструкторы:

- конструктор по умолчанию;
- конструктор копирования.

Конструктор по умолчанию вызывается компилятором автоматически, когда порождается объект, а параметры не указываются. Поэтому такой конструктор так и описывается — без параметров. В примере выше это первый конструктор.

Конструктор копирования вызывается компилятором, если при описании переменной-объекта производится инициализация этой переменной значениями другой переменной, того же типа класса. То есть

при описании переменной простого базового типа, можно ее инициализировать ее значением, хранящимся в другой переменной. Аналогично можно поступать и при описании переменной-объекта. Пример описания такого конструктора приведен вторым. Обратите внимание, что аргументом является адрес переменной типа того же класса, и используется ключевое слово `const`.

Набор конструкторов, разработанных при описании класса, предоставляет гибкость при использовании этого класса в программах.
Пример.

```
_3d p1(22., 33., 44.);
                        // вызывается общий конструктор
_3d p2;
                // вызывается конструктор по умолчанию
_3d p3 = p1;
                // вызывается конструктор копирования
_3d p4(p1);
                // вызывается конструктор копирования
_3d *q1 = new _3d (22., 33., 44.);
                        // вызывается общий конструктор
_3d *q2 = new _3d;
                // вызывается конструктор по умолчанию
_3d *q3 = new _3d (*p3);
                // вызывается конструктор копирования
delete q1; delete q2; delete q3;

            // удаление динамически созданных объектов
```

Все объекты, память которым выделялась динамически, должны уничтожаться явно в программе. То же самое касается и переменных типа класс. При этом вызывается деструктор этого класса. Для переменных описанных статически, деструктор вызывается автоматически, когда программа выходит из блока, где такая переменная была описана.

В отличие от конструкторов, при описании класса описывается *единственный* деструктор. Способов породить объект данного класса должно быть много, чтобы при использовании класса разработчик программы мог воспользоваться более подходящим. Поэтому конструкторов много. Деструктор класса вызывается неявно, и этот вызов порождается компилятором. Поэтому для однозначности такого вызова необходим единственный деструктор. Да и необходимость описывать деструктор необходимо только в тех случаях, когда при уничтожении объекта автоматически созданный деструктор не может

выполнить всех необходимых действий по корректному уничтожению объекта. Как правило, этой причиной является выделение динамической памяти во время жизни объекта. Поэтому в течение жизни объекта его работа должна быть так организована, чтобы была выделена необходимая память и высвобождена. Как правило, полное высвобождение занятой динамической памяти производится уже при уничтожении объекта, то есть деструктором.

Виртуальные методы

Реализация принципа полиморфизма на практике осуществляется посредством виртуальных методов класса. Описание таких методов осуществляется подобно другим методам, но описание такого метода дополняется ключевым словом `virtual`.

Пример.

```
class _3d
 {
   double x, y, z;
  public :
   _3d (){x = 0; y = 0; z = 0;};
   _3d (const _3d &a)
                      {x = a.x; y = a.y; z = a.z;};
   _3d (double &X, double &Y, double &Z)
                             {x = X; y = Y; z = Z;};
   _3d (double C);
   ~_3d (){};
   double mod ();
   virtual void print ();
 };
_3d::_3d (double C)
 {
  x = C; y = C; z = C;
 };
double _3d::mod ()
 {
  return sqrt (x * x + y * y + z * z);
 };
void _3d::print ()
 {
  cout << x << "; " << y << "; " << z << endl;
 };
```

Данный пример содержит описание того же класса, но он теперь дополнен описанием еще одного метода `print`, являющегося виртуальным.

Отличие и преимущество виртуальных методов перед статическими наиболее ярко заметен при создании целого семейства объектов, то есть объекта и набора его потомков и даже их потомков. Стандартным примером является иерархия объектов, представляющих собой описание геометрических фигур: базовый класс — *Полигон*, его наследники *Равносторонний* и *Треугольник*; и наследники *Равностороннего* — *Квадрат* и *Ромб*. Их иерархическое соотношение можно изобразить диаграммой.

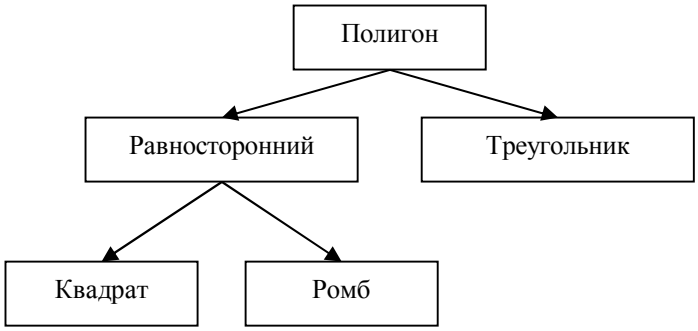

Рис. _. Иерархия классов геометрических фигур

Положим, что среди множества методов описанных в этих классах есть и метод рисования объекта — метод `Draw`. Положим, что этот метод статический.

Положим, что мы создаем некую программу, и эта программа содержит массив разнотипных объектов из этого семейства и треугольники, и квадраты, и ромбы. И в этой программе есть функция, которая выполняет рисование всех этих объектов, то есть просто в цикле вызывает метод `Draw` для очередного элемента массива–объекта.

Решение такой задачи может быть следующим. Массив может содержать только элементы одного типа. Поэтому такой массив может быть массивом указателей на класс *Полигон*. Но при создании каждого из элементов, например `i`-го, мы можем вызывать конструктор соответствующего класса — или треугольника, или квадрата, или ромба.

После такого построения массива он действительно будет содержать указатели на объекты разных классов.

Но если теперь запустить цикл и вызывать метод `Draw` для каждого из объектов, то обнаружим, что работает только единственный метод — метод базового класса *Полигон*.

Картина совершенно меняется, если и в базовом классе, и в его потомках мы метод `Draw` объявим виртуальным. Если теперь запустить цикл и вызывать метод `Draw` для каждого из объектов, то вызываться и работать будет метод, разработанный для каждого из классов. То есть каждый объект будет рисоваться методом `Draw` соответствующего класса.

Наследование

Класс–наследник описывается, когда есть необходимость дополнить поля класса новыми полями и методы класса–предка новыми методами и/или исправить поведение отдельных методов. При описании такого класса указывается имя класса–предка и имена методов, от которых наследуются методы класса–потомка.

Пример.

```
class _3d_Chield : public _3d
 {
   double coeff;
  public :
   _3d_Chield (double C);
};
_3d_Chield::_3d_Chield (double C) : _3d (C)
 {
  coeff = C * C;
 };
```

В данном примере описан класс `_3d_Chield`. В описании данного класса указано, что он является наследником класса `_3d`, и этот класс дополнен полем `coeff`, а один из конструкторов перекрыт новым. Ключевое слово `public` в заголовке класса[1] используется для указания, что все методы класса–предка должны быть доступны и для класса потомка. В реализации конструктора класса–потомка указано, что должен быть вызван конструктор предка с подходящими параметрами, а потом выполнены те операторы, которые перечислены в теле нового конструктора. То есть тело нового конструктора, по сути, представляет собой комбинацию тела соответствующего конструктора предка и новых операторов. То есть входной параметр `C` будет передан в конструктор предка, он выполнится, а затем выполнится оператор присваивания.

[1] Помимо ключевого слова `public` в заголовке класса могут указываться и другие типы доступа, но тогда поля и методы класса-предка будут недоступны.

Если спецификатор доступа не указан, то секция, непосредственно следующая за открывающей скобкой, по умолчанию считается `private`.

В классе-потомке обязательно описывается конструктор. При его работе первым вызывается конструктор предка.

Конструкторы вызываются в том же самом порядке, в каком классы следуют один за другим в иерархии классов. Поскольку базовый класс ничего не знает про свои производные классы, то его инициализация может быть отделена от инициализации производных классов и производится до их создания, так что конструктор базового класса вызывается перед вызовом конструктора производного класса.

В классе-потомке, при описании конструктора, можно указывать явно, а какой из конструкторов предка именно вызывается. Этот вызов также является первым оператором в теле конструктора-наследника.

В деструкторе от класса-наследника деструктор-предок явно не указывается. Нельзя перекрывать деструктор. Но он обязательно вызывается, как последний оператор в деструкторе-наследнике.

В противоположность конструкторам, деструктор производного класса вызывается перед деструктором базового класса. Причину этого также легко понять. Поскольку уничтожение объекта базового класса влечет за собой уничтожение и объекта производного класса, то деструктор производного объекта должен выполняться перед деструктором базового объекта.

Наследование полей и методов

В производных классах (классах-потомках) можно определять новые поля и методы с тем же именем, что и у базовых (у классов-предков). Тогда для того, чтобы явно указать к какому элементу обращаются, необходимо указывать или имя предка или указатель `this`.

Указатель `this` — это константный указатель на текущий объект (т. е. на объект, к которому применяется вызываемый метод). Этот указатель существует всюду внутри класса, его не надо как-то явно объявлять, к нему можно обращаться внутри любого метода класса.

Пример.

```
void Base::getData()
        //метод класса, который выводит данные на экран
{
cout << "\n" << this->str  << this->number << endl;
}
```

Другой пример.

```
class Child: public Base
{
public:
virtual int count(int val)
  {
  // вызов метода базового класса в методе наследника
    return Base::count(val) * val;
  }
};
```

Наследование виртуальных функций

Виртуальная функция — это метод (функция-член), которая, как предполагается, будет переопределена в производных классах.

При переопределении в наследнике, такая функция может вызывать как другие методы, в том числе и методы предка. Тогда указывается имя предка перед именем функции.

Предположим, что базовый класс содержит функцию, объявленную как *Виртуальный*, и производный класс определяет ту же функцию. Функция-член производного класса будет вызвана для объектов этого класса, даже если они объявлены как указатели или ссылки на базовый класс. (технология позднего связывания)

При описании функции (переопределении) в классе потомке, ключевое слово `virtual` может быть опущено.

Другие методы, если имеют такое же имя, должны иметь другой список формальных параметров. Изменения типа возвращаемого значения (через имя функции) *недостаточно*.

Абстрактные методы

Абстрактные классы используются в качестве обобщенных концепций, на основе которых можно создавать более конкретные производные классы. Невозможно создать объект типа абстрактного класса; однако можно использовать указатели и ссылки на типы абстрактного класса.

Класс, содержащий хотя бы одну чисто виртуальную функцию, считается абстрактным. Классы, производные от абстрактного класса, должны реализовать все его чисто виртуальные функции, иначе они также будут абстрактными классами.

```
class Account {
public:
   Account( double d );   // Constructor.
```

```
    virtual double GetBalance();   // Obtain balance.
    virtual void PrintBalance() = 0;
                                   // Pure virtual function.
private:
    double _balance;
};
```

Единственное различие между этим и предыдущим объявлениями состоит в том, что функция `PrintBalance` объявлена со спецификатором чисто виртуальной функции `pure` (= 0).

Методы `inline`

Реализация некоторых методов может быть описана непосредственно в описании интерфейса класса. Тогда это `inline` метод. При вызове такого метода не происходит вызова функции, а код программы компилятором строится так, что код метода пишется непосредственно в текст программы.

Так же подобные методы могут быть реализованы в файле .HPP отдельно (ниже определения интерфейса), но с ключевым словом `inline`.

Свойства

RAD Studio C++ (Builder) использует модификатор `_property` для объявления свойств компонентных классов. Синтаксис описания свойства имеет вид:

`_property <тип свойства> <имя свойства> = {<список атрибутов>} ;`

Список атрибутов содержит перечисление следующих атрибутов свойства:

`write` = < член данных или метод записи > – определяет способ присваивания значения члену данных;

`read` = < член данных или метод чтения > – определяет способ получения значения члена данных;

`default` = < булева константа > – разрешает или запрещает сохранение значения свойства по умолчанию в файле формы *.dfm;

`stored` = < булева константа или функция > – определяет способ сохранения значения свойства в файле формы с расширением *.dfm.

Свойства определяются в разделе `public`.

C++ Builder использует модификатор `__published` для спецификации тех свойств компонентов, которые будут отображаться Инспектором объектов на стадии проектирования приложения. Правила видимости, определяемые этим ключевым словом, не отличаются от правил видимости членов данных, методов и свойств, объявленных как `public`.

Пример.

```
class  PACKAGE TAngles : public TPersistent
{
private:
    int FStartAngle;
    int FEndAngle;
    TNotifyEvent FOnChange;
    void __fastcall SetStart(int Value);
    void __fastcall SetEnd(int Value);
public:
    void __fastcall Assign(TPersistent* Value);
    void __fastcall Changed(void);

__published:
    __property int StartAngle =
        {read=FStartAngle, write=SetStart, nodefault};
    __property int EndAngle =
            {read=FEndAngle, write=SetEnd, nodefault};
    __property TNotifyEvent OnChange =
                    {read=FOnChange, write=FOnChange};
public:
    __fastcall TAngles(void);
    __fastcall virtual ~TAngles(void);
};
```

Пример.

```
class TNeitron: public TRazmer{
 private:
double w1[100][10];
   int w2[10][100];
 public:
  TNeitron();
  void Set_w1(int i, int j, double par)
                                        {w1[i][j]=par;}
  double Get_w1(int i, int j){return w1[i][j];}
  void Set_w2(int i, int j, int par){w2[i][j]=par;}
  int Get_w2(int i, int j){return w2[i][j];}

  __property double Matrix_w1 [int Row][int Col] =
                        { read=Get_w1, write=Set_w1 };
  __property int Matrix_w2 [int Row][int Col] =
                        { read=Get_w2, write=Set_w2 };
};
```

Операторы классов

Иногда хочется проявить творчество и облегчить программный код для себя и для других. Для себя написание, для других понимание. Если, например, в программе часто встречается функция добавления одной строки в конец другой, конечно, можно это реализовать разными, несколькими способами. Например, перегрузить некоторый оператор. Перегружать можно только предопределенные операторы языка C++ (см. табл.).

Таблица. Перегружаемые операторы

```
+     -     *     /     %     ^     &     |     ~
!     ,     =     <     >     <=    >=    ++    --
<<    >>    ==    !=    &&    ||    +=    -=    /=
%=    ^=    &=    |=    *=    <=    >>=   []    ()
```

–> –>* new new[] delete delete[]

Оператор в C++ — это некоторое действие или функция обозначенная специальным символом. Для того что бы распространять эти действия на новые типы данных, при этом сохраняя естественный синтаксис, в C++ была введена возможность перегрузки операторов.

Список операторов:

```
+ - * / % //Арифметические операторы
+= -= *= /= %=
+a -a //Операторы знака
++a a++ --a a-- //Префиксный и постфиксный инкременты
&& || ! //Логические операторы
& | ~ ^
& = |= ^=
<< >> <<= >>= //Битовый сдвиг
= //Оператор присваивания
== != //Операторы сравнения
< > >= <=
```

Специальные операторы:

```
&a *a a-> a->*
() []
(type)
. ,  (a ? b : c)
```

Не все операторы можно переопределять. Операторы "." и "a?b:c"(тернарный оператор) переопределить нельзя.

Так же нужно отметить что, переопределяя операторы "," "&&" "||" теряются их "ленивые" свойства.
Операторы "a->", "[]", "()", "=" и "(type)" можно переопределить только как методы класса.

Примеры, того как переопределять операторы на примере некоего класса.

Бинарный оператор

```
BigInt operator+(BigInt const & a, BigInt const & b)
{
    ...
}
```

Бинарный оператор — это функция от двух параметров, параметрами которой являются левый и правый операнды оператора.

Унарный оператор

```
BigInt & operator++(BigInt const & b)
{
    ...
}
```

Унарный оператор — это оператор от одного параметра. Если он объявлен внутри класса, то этим параметром (неявным) является `this`.

При перегрузке операторов ">>" и "<<", для ввода и вывода через потоки нужно подключить заголовочный файл `iostream`.

```
#include <iostream>
...
std::istream & operator >>(std::istream & is, BigInt
& n)
{
    ...
}
std::ostream & operator <<(std::ostream & os, BigInt
const & n)
{
    ...
}
```

Переопределение префиксных и постфиксных операторов:

```
BigInt & operator --(BigInt & n);//Префиксный
BigInt operator --(BigInt & n, int);//Постфиксный
```

Фактически параметра `size_t size` у операции нет — он фиктивен. Это некий финт для того, чтобы внести различия в сигнатуры операторов.

Переопределение операторов внутри класса:

```
class BigInt
{
    ...
    BigInt * operator ->();
```

```
    char operator [](size_t i) const;
    char & operator [](size_t i);
    ...
};
```

Обратите внимание, что при переопределении "–>" необходимо вернуть *именно указатель* на объект.

Необходимо отметить, оператор "[]" переопределен *как константный* и *не константный*. Во втором случае мы возвращаем *ссылку на объект*, а не сам объект. Это может быть полезно при определении массивов.

Как правильно перегружать оператор "="

Для перегрузки оператора "=" есть специальная идиома, которая облегчает присваивание сложных объектов.

Рассмотрим следующий код:

```
class BigInt
{
    size_t size_;
    char * digits_;

    BigInt(BigInt const & num)
    {
      ...
    }

    void swap(BigInt & b)
    {
        std::swap(size_, b.size_);
        std::swap(digits_, b.digits_);
    }
    BigInt & operator = (BigInt const & num)
    {
        if(this != &n)
        {
            BigInt(num).swap(*this);
        }
        return *this;
    }
    ...
};
```

Функция `std::swap(a,b)` - меняет значение `a` и `b` местами.

Проверить код примера.

Дружественная функция

Дружественная функция — это функция, которая не является членом класса, но имеет доступ к членам класса, объявленным в полях `private` или `protected`.

При описании используется ключевое слово `friend`.

Помимо этого, существуют и *дружественные классы*.

Шаблоны классов

Шабло́ны (англ. *template*) — средство языка C++, предназначенное для кодирования обобщённых алгоритмов, без привязки к некоторым параметрам (например, типам данных, размерам буферов, значениям по умолчанию).

В C++ возможно создание шаблонов функций и классов.

Шаблоны позволяют создавать параметризованные классы и функции. Параметром может быть любой тип или значение одного из допустимых типов (целое число, `enum`, указатель на любой объект с глобально доступным именем, ссылка). Например, нужен какой-то класс:

```
class SomeClass{
     int SomeValue;
     int SomeArray[20];
     ...
};
```

Для одной конкретной цели можно использовать этот класс. Но, вдруг, цель немного изменилась, и нужен еще один класс. Теперь нужно 30 элементов массива `SomeArray` и вещественный тип `SomeValue` элементов `SomeArray`. Тогда мы можем абстрагироваться от конкретных типов и использовать шаблоны с параметрами. Синтаксис: в начале перед объявлением класса напишем слово `template` и укажем параметры в угловых скобках. В нашем примере:

```
template < int ArrayLength, typename SomeValueType >
class SomeClass{
     SomeValueType SomeValue;
     SomeValueType SomeArray[ ArrayLength ];
     ...
};
```

Тогда для первой модели пишем:

```
SomeClass < 20, int > SomeVariable;
```

для второй:

```
SomeClass < 30, double > SomeVariable2;
```

Хотя шаблоны предоставляют краткую форму записи участка кода, на самом деле их использование не сокращает исполняемый код, так как для каждого набора параметров компилятор создаёт отдельный экземпляр функции или класса. Как следствие, исчезает возможность совместного использования скомпилированного кода в рамках разделяемых библиотек.

В предыдущем примере мы описывали **init** для того, чтобы проинициализировать индексы источника и приемника. Эта функция должна была вызываться для каждой вновь создаваемой переменной этого типа. Если теперь преобразовать **C_Buf** в класс, то логично переделать **init** в конструктор. Этот пример, когда конструктор просто необходим при описании класса.

Более редкий случай, когда необходимо применение деструктора. Деструктор нужен, например, для освобождения динамической памяти, занятой объектом. Вот пример описания конструкторов класса **_3d**:

```
class _3d
{_3d (double &X, double &Y, double &Z) {x = X; y = Y; z = Z;};
_3d (_3d &a) {x = a.x; y = a.y; z = a.z;};
}
```

Если необходимые конструкторы или деструктор для класса не описаны. То транслятор создает их сам.

Вызов конкретного конструктора для создаваемого объекта происходит в зависимости от аргументов, которые могут быть указаны в круглых скобках после имени создаваемого объекта. Например :

```
_3d     A(0., 1., 0.), B;
```

Здесь для объекта A будет вызван описанный нами конструктор **_3d (double &X, …)**, а для объекта B — созданный транслятором **_3d ();** .

Существует специальный тип конструктора — *конструктор копирования*, который вызывается при выходе из функции, если та возвращает объект данного класса. Дело в том, что все объекты, описанные внутри функции разрушаются (для них вызывается деструктор) при выходе из нее. Такой конструктор нужен для того, чтобы скопировать результат до того, как он будет разрушен. Это необходимо например, для объектов, использующих динамическую память. В качестве аргумента, в этом конструкторе выступает объект того же класса.

Пример 1. Часто требуется иметь буфер значительного объема. Поэтому нужно изменить описание класса **C_Buf** таким образом, чтобы можно было задавать объем буфера при его объявлении. В состав класса вводится дополнительно длина буфера. Эта переменная подставляется вместо 1024. Кроме того, вводится конструктор, который размещает массив **ptr**, и деструктор, который его освобождает.

```
class C_Buf
{
  public :
  double        *ptr;   // массив буфера
  int     dest, src;        // dest — приемник, src — источник
  int     len;
  C_Buf (int len = 1024);      // void init () {src = dest = 0;}; ???
  ~C_Buf () {if (ptr != NULL) free (ptr);};
  void  add (double &a);
  double        get ();
  int     used ();
  int     free ();
}
C_Buf::C_Buf (int _len)
{
len = _len;
dest = src = 0;
ptr = (double *) malloc (len * sizeof (double));
if (ptr == NULL) len = 0;
};
```

```
void C_Buf::add (double &a)
{
  ptr[dest++] = a;
  if (dest == len) dest = 0;
};
double C_Buf::get ()
{
if (++src != len) return ptr [src - 1];
src = 0;
return ptr [len - 1];
};
double C_Buf::used ()
{
  int     n = dest - src;
  if (n >= 0) return n;
  else return n + len;
};
int C_Buf::free ()
{
  int     n = src - dest;
  if (n > 0) return n;
  else return n + len;
};
```

(конец).

Приложение 1

Если есть **a = R.mod ();** то **this** при этом вызове соответствует адресу **R**, а функция **mod ()** может быть реализована таким образом :

```
double _3d::mod ()
{
   return sqrt (this->x * this->x + this->y * this->y + this->z * this->z);
}
```

Если переменная не описана ни внутри функции, ни как глобальная переменная, то считается, что она является членом структуры и принадлежит рабочей переменной ***this**. Поэтому можно опустить **this** и к членам структуры обращаться просто по имени.

Пример 1 : Написать функцию **double _3d::proection (_3d r);** которая будет возвращать длину проекции вектора на рабочий вектор. Проекция вектора **A** на **B** вычисляется по формуле : $(x_a + x_b * y_a * y_b + z_a * z_b) / mod (B)$.

```
double _3d::proection (_3d r)
{
   return (x * r.x + y * r.y + z * r.z) / sqrt (x * x + y * y + z * z);
}
```

Пример 2 : Опишите структуру **polar**, определяющую вектор в сферической системе координат : **r, fi, l**. Для нее написать функцию **_3dpolar::vect();** возвращающую рабочий вектор в декартовой системе координат.

```
struct polar
{
   double r, fi, l;
   double vect ();
};
_3d polar::vect ()
{
   double cf=cos(fi), sf=sin(fi), cl=cos(l), sl=sin(l);
   _3d     R;
   R.x = r * sf * cl;
   R.y = r * sf * sl;
   R.z = r * cf;
   return R;
}
```

Пример 3 : Приведите пример структуры C_Buffer, описывающей кольцевой буфер емкостью 1024 действительных числа. Для нее описать следующие функции :

```
void init (); //   инициализирующая;
void add (double); //   добавляющая элемент;
double get (); //          взять элемент;
int free (); //    величина свободного пространства
int used (); //   величина занятого пространства;
```

Примечание. Буферизация или организация очереди широко распространенное техническое решение. Оно применяется, например, для согласования двух устройств или процессов, работающих в разном темпе. Один из способов реализации буфера — кольцевой буфер на одномерном ограниченном массиве.

Идея решения заключается в следующем. Имеется одномерный массив и две индексные переменные. Одна — индекс приемника — указывает на элемент массива, куда записывается вновь поступающее значение. При записи индекс приемника увеличивается на 1. Вторая — индекс источника — указывает на элемент массива, из которого извлекается значение. При увеличении индекс источника также увеличивается на 1. Если один из индексов выходит за границу массива, то он устанавливается на его начало. В организованной таким образом очереди могут "стоять" не только действительные числа, но и объекты с более сложной структурой. В соответствие с вышеизложенным опишем кольцевой буфер :

```
struct C_Buf
{
  double        ptr[1024];      // массив буфера
  int     dest, src;        // dest — приемник, src — источник
  void  init () {src = dest = 0;};
  void  add (double &a);
  double        get ();
  int     used ();
  int     free ();
}
void C_Buf::add (double &a)
{
  ptr[dest++] = a;
  if (dest == 1024) dest = 0;
};
double C_Buf::get ()
{
if (++src != 1024) return ptr [src - 1];
src = 0;
return ptr [1023];
};
double C_Buf::used ()
{
  int     n = dest - src;
  if (n >= 0) return n;
  else return n + 1024;
};
int C_Buf::free ()
{
  int     n = src - dest;
  if (n > 0) return n;
  else return n + 1024;
};
```

P.S. Функция **init ()** нужна для инициализации указателей. Обращение к ней обязательно перед использованием буфера.

Данная структура — показательный пример того, к чему нужно стремиться при написании программ. При реализации этой структуры удалось добиться того, что ни массив, ни указатель *в явном виде* в программе *не используются* — все делают функции!