Шаблоны

Лекция 10

Шаблоны функций

- Язык C++ является языком со строгой типизацией.
- Но нам хотелось бы видеть привычные функции, которые можно было бы использовать вне зависимости от типа входных параметров:

```
min(int, int)
min(double, double)
min(rational, rational)
```

Шаблоны

1. Механизм шаблонов реализует в C++ **параметрический полиморфизм.**

2. Шаблон представляет собой предварительное описание функции или класса, конкретное представление которых зависит от параметров шаблона.

3. Для описания шаблонов используется ключевое слово **template**, вслед за которым указываются аргументы (параметры шаблона), заключенные в угловые скобки.

4. Параметры шаблона перечисляются через запятую, и могут быть:
   а) объектами следующих типов:
      - целочисленного,
      - перечислимого,
      - указательного (в том числе указатели на члены класса),
      - ссылочного;
   б) именами типов (перед именем типа надо указывать ключевое слово **class** или **typename**).

5. Параметры-объекты являются **константами**, их нельзя изменять внутри шаблона.

Шаблоны функций.

template < список_параметров_шаблона >
тип_рез-та имя_функции (список_аргументов_функции) { /*...*/ }

Обращение к функции-шаблону: имя_функции < список_фактич._пар._шаблона >
(список_фактич_аргументов_функции);

Пример:

```
template < class T > // функция суммирования элементов массива
T  sum ( T  array[ ],  int size ) {
        T  res = 0;
        for ( int i = 0; i < size; i++ )    res += array[ i ];
        return res;
}
```

Использование шаблона для массивов типа **int** [10] :

```
int  iarray [10];
int  i_sum;
//...
i_sum = sum < int > ( iarray, 10 );
```

Можно задать аргумент **size** в виде параметра шаблона:

```
template < class  T, int  size >
T  sum ( T  array [ ] ) { /* ... */ }
```

Тогда вызов sum будет таким: i_sum = sum < **int**, 10 > (iarray);

Неявное определение параметра-типа шаблона

Пример 1.

```
class complex
{...  public:
            complex ( double r = 0, double i = 0 );
            operator double ();        .....
};
template < class T >
T  f ( T& x, T& y )  {
            return x > y ? x : y;
}
double f ( double x, double y ){
            return x > y ? -x : -y;
}
int main ( ) {
            complex a ( 2 , 5 ), b ( 2 , 7 ), c;
            double x = 3.5, y = 1.1;
            int  i,  j = 8,  k = 10;

            c = f ( a , b );        // f < complex > (a , b)
            x = f ( a , y );        // f (a , y)
            i = f ( j , k );        // f  < int > (j , k)
            return 0;
}
```

Пример 2.

```
template < class T >
T max (T & x, T & y) {
    return x > y ? x : y;
}

int main ( ) {
    double x = 1.5,  y = 2.8,  z;
    int  i = 5, j = 12,  k;
    char * s1 = "abft";
    char * s2 = "abxde", * s3;

    z = max ( x, y );                  // max <double>
    k = max < int > (i, j);            // max <int>
    //z = max (x, i);                  // Err! -  неоднозначный выбор параметров
    z = max < double > ( y, j );
    s3 = max  (s2, s1);                // max < char * >,
                                       // но происходит сравнение адресов
    return 0;
}
```

Пример 3.

```
template <class T>  T  m1 (T a, T b) {
   cout << "m1_1\n";
   return a < b ? b : a;
}

template <class T, class B> T m1 (T a, T b, B c) {
  cout << "m1_2\n";
  c = 0;  return a < b ? b : a;
}
template <class T, class Z> T m1 (T a, Z b) {
  cout << "m1_3\n";
  return a < b ? b : a;
}

int m1 (int a, int b) {
        cout << "m1_4\n";
        return a < b ? b : a;
}

int m1 (int a, double b) {
        cout << "m1_5\n";
        return a;
}

int main () {
  int i;
  m1 <int> (2, 3);              // m1_1        // m1_3
  m1 <int, int> (2, 3);         // m1_3        // m1_3
  m1 <int> (2, 3, i);           // m1_2        // m1_2
  m1 (1, 1);                    // m1_4        // m1_4
  m1 (1.3, 1);                  // m1_3        // m1_3
  m1 (1.3, 1.3);                // m1_1        // m1_3
  return 0;
}
```

// Если убрать первый шаблон:

Видимость

- Если в глобальной области видимости объявлен объект, функция или тип с тем же именем, что у параметра шаблона, то глобальное имя оказывается скрытым.

```
typedef double Type;
template <class Type>
   Type min( Type a, Type b ) {
      Type tmp = a < b ? a : b;
      return tmp;
   }
```

Алгоритм выбора оптимально отождествляемой функции с учетом шаблонов

- Для каждого шаблона, подходящего по набору формальных параметров, осуществляется формирование специализации, соответствующей списку фактических параметров.

- Если есть два шаблона функции и один из них более специализирован (т.е. каждый его допустимый набор фактических параметров также соответствует и второй специализации), то далее рассматривается только он.

- Осуществляется поиск оптимально отождествляемой функции из полученного набора функций, включая определения обычных функций, подходящие по количеству параметров. При этом если параметры некоторого шаблона функции были определены путем **выведения** по типам фактических параметров вызова функции, то при дальнейшем поиске оптимально отождествляемой функции к параметрам данной специализации шаблона нельзя применять никаких описанных выше преобразований, кроме преобразований т**очного** отождествления.

- Если обычная функция и специализация подходят одинаково хорошо, то выбирается обычная функция.

- Если полученное множество подходящих вариантов состоит из одной функции, то вызов разрешим. Если множество пусто или содержит более одной функции, то генерируется сообщение об ошибке.

Шаблоны классов.

Шаблоны создаются для классов, имеющих общую логику работы.

Для определения шаблона класса перед ключевым словом **class** помещается **template**-квалификатор.

template <список_параметров_шаблона_типа> **class** имя_класса { /*...*/ };

Конкретный экземпляр шаблона класса (объект класса) можно создать так:

имя_класса <список фактич_парам> объект;

Для шаблонов класса никакие фактические параметры по умолчанию не выводятся.

Функции-члены класса-шаблона автоматически становятся функциями-шаблонами.

Шаблоны методов.

Можно описывать шаблонные методы в классах, не являющихся шаблонами.

Запрещено определять шаблонны для виртуальных методов из-за возникающих больших накладных расходов на возможную перестройку таблиц виртуальных методов при компиляции.

Шаблонный класс stack.

```
template <class T, int max_size >
class stack {
        T s [max_size];
        int top;
public:
        stack ( ) { top = 0;}
        void reset ( ) { top = 0;}
        void push (T i);
        T pop () ;
        bool is_empty ( ) { return top == 0;}
        bool is_full  () { return top == max_size;}
};
```

```
template <class T, int max_size >
void stack <T, max_size > :: push (T i) {
        if ( ! is_full () ) {
                s [top] = i;
                top ++;
        }
        else
                throw "stack_is_full";
}
```

```
template <class T, int max_size >
T stack <T, max_size > :: pop ( ) {
        if ( ! is_empty () ) {
                top --;
                return s [top];
        }
        else
                throw "stack_is_empty";
}
```

Диаграммы UML

Часто при проектировании программ, разрабатываемых в объектно-ориентированном стиле, взаимосвязь используемых в них классов и объектов представляют в виде диаграмм UML.

Классы изображают в виде прямоугольника, состоящего из трех частей:

сверху – имя класса,

в середине – члены данные, возможно, с указанием типов,

внизу – прототипы методов класса.

Имена абстрактных классов и чистых виртуальных функций выделяются *курсивом*.

Перед описанием имени члена класса или метода можно указать спецификатор доступа с помощью значков

+ (public),

- (private),

 # (protected).

Для статических членов класса после спецификатора доступа указывается символ $.

Виды отношений между классами

Большинство ООЯП поддерживают следующие отношения между классами:

- Ассоциация.
- Наследование.
- Агрегация.
- Использование.
- Инстанцирование.

Ассоциация

Ассоциация – отношение, показывающее, что два класса концептуально взаимодействуют друг с другом.

Отношение ассоциации удобно представлять в виде ER-диаграмм (entity – relationships – сущность - связь), в основном используемым при разработке реляционных баз данных. Связи изображаются сплошными линиями без направления.

Виды связей, представляемых ER-диаграммами:

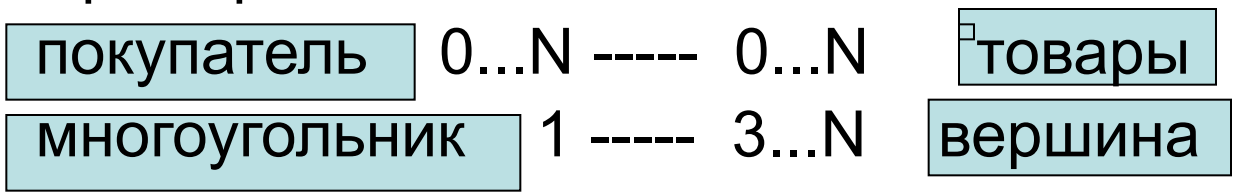
1—1

1—N

N—N

Различают обязательное и необязательное участие сущностей в установленных между ними связях.

Примеры:

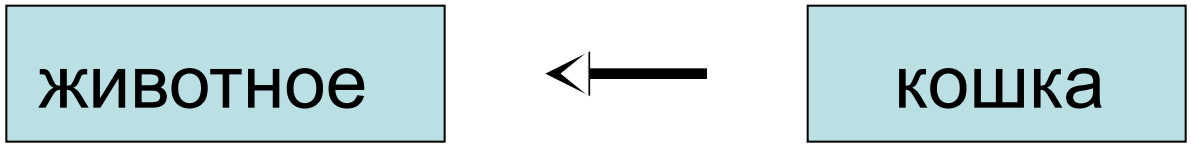


Наследование

Часть – общее (“ is a ”).

Отношение задается в виде стрелки с незакрашенным треугольником на конце, которая указывает на базовый класс.

Пример:



базовый класс
суперкласс

производный класс
подкласс

Агрегация.

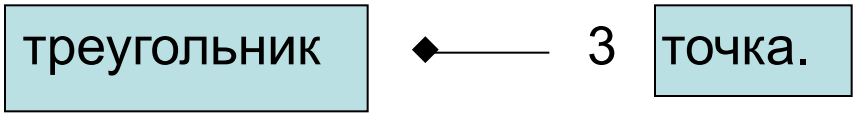
Часть – целое (“ has a ”).

Строгая агрегация – **композиция.**

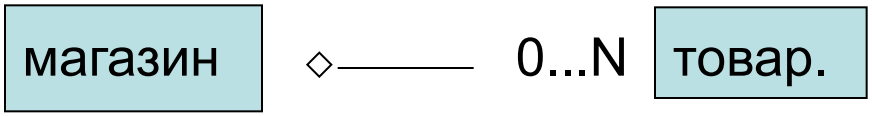
Нестрогая агрегация - **агрегация** (при этом один объект может быть включен в разные объекты одновременно).

Композиция обозначается стрелкой с закрашенным ромбом на конце, направленной на включающий класс, а агрегация – стрелкой с незакрашенным ромбом на конце.

Примеры:




```
class triangle {...
        point p1,p2,p3; ...
}
```




```
class shop {...
        goods * g; ...
}
```

Использование и Инстанциирование

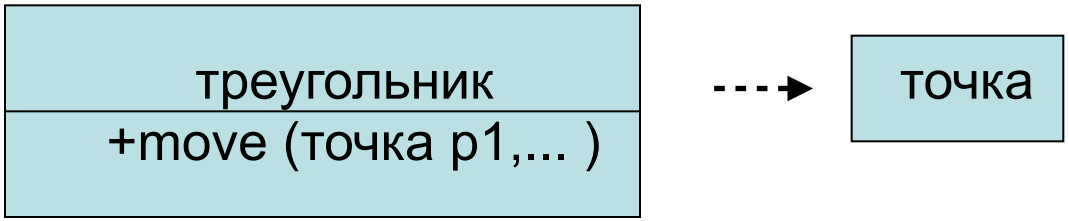
Отношение **использования** возникает, когда

- в прототипе метода одного класса используется имя другого класса;
- в теле метода одного класса - локальный объект другого класса;
- в теле метода одного класса вызывается функция другого класса.

Использующий класс называют client, а используемый supplier.

Отношение использования обозначается пунктирной стрелкой указывающей на класс supplier.

Пример:



Инстанцирование – связь между шаблоном класса и классом - результатом генерации по шаблону.

В UML инстанцирование обозначается стрелкой, идущей от шаблона класса к конкретной его реализации.