ППОИС

Часть 1

Обобщенное программирование

Назначение шаблонов в C++

- обобщенное программирование
- параметрический полиморфизм

Обобщенное программирование

Это парадигма программирования, заключающаяся в написании алгоритмов, которые можно применять к различным типам данных.

```
int max(int a, int b) {
    return (a > b) ? a : b;
}

double max(double a, double b){
    return (a > b) ? a : b;
}
```

Параметрический полиморфизм

Параметрический полиморфизм позволяет определять функцию или тип данных обобщенно, так что значения обрабатываются идентично вне зависимости от их типа.

Параметрически полиморфная функция использует аргументы на основе поведения, а не значения, апеллируя лишь к необходимым ей свойствам аргументов, что делает её применимой в любом контексте, где тип объекта удовлетворяет заданным требованиям поведения.

Синтаксис шаблонной функции

```
template <typename T>
T minimum(const T& lhs, const T& rhs)
{
    return lhs < rhs ? lhs : rhs;
}


int i = minimum<int>(a, b);

int i = minimum(a, b);
```

Синтаксис шаблонного класса

```
template <typename T>
class Account {
private:
    T id;
public:
    Account(T id) : id(id) { }
    T getId() {
        return id;
    }
};
```

Параметры шаблона

- class или typename - в качестве параметра передается тип
- не-типы, например, int size
- вывод типа параметра
- параметры по-умолчанию
- не фиксированное количество параметров, оператор (...)
- шаблон как параметр шаблона

```
template< class T1,                          // параметр-тип
          typename T2,                       // параметр-тип
          int I,                             // параметр обычного типа
          T1 DefaultValue,                   // параметр обычного типа
          template< class > class T3,        // параметр-шаблон
          class Character = char             // параметр по умолчанию
        >
```

Примеры

```
template <typename T, typename U, typename V> class Foo{};

template <class T, class U, class V> class Foo{};

template<typename... Arguments> class vtclass;

template <class T, class Allocator = allocator<T> > class vector;
```

Параметры не-типы

- целочисленное значение или перечисление;
- указатель или ссылка на объект класса;
- указатель или ссылка на функцию;
- указатель или ссылка на метод класса;
- std::nullptr_t.

Пример шаблонного параметра шаблона (1)

```
#include <iostream>
#include <functional>
#include <algorithm>

template < typename BinPred , typename T >
void foo (  T & obj1 , T & obj2 , BinPred pred )
{
    if ( pred(obj1,obj2) )
        std::swap (obj1,obj2) ;
}

int main ()
{
    int x = 10 ;
    int y = 30 ;
    //foo<std::less> ( x , y ) ;
    foo ( x , y , std::less<int>() ) ;
    std::cout << x << ' ' << y << std::endl ;
}
```

Пример шаблонного параметра шаблона (2)

```
template < template <typename> class BinPred , typename T >
void foo (  T & obj1 , T & obj2 )
{
    if ( BinPred<T>()(obj1,obj2) )
        std::swap(obj1,obj2) ;
}
```

```
int main ()
{
    int x = 10 ;
    int y = 30 ;
    foo<std::less> ( x , y ) ; //Для std::less не нужно указывать тип параметра, т.к. мы передаем сам шаблон
    std::cout << x << ' ' << y << std::endl ;
}
```

Типы как члены классов

```
class Container
{
public:
  int array[ 15 ];
  typedef int* iterator;
  /* ... */
  iterator begin() { return array; }
};

template< class C >
void f( C& vector )
{
  C::iterator i = vector.begin();          // ошибка
  typename C::iterator i = vector.begin();
}
```

Шаблоны как члены классов

```
class A {
  /* ... */
  public:
    template< class T > T& ConvertTo();
    template< class T > void ConvertFrom( const T& data );
};

template< class T >
void f( T Container )
{
int i1 = Container.template ConvertTo<int>() + 1;
Container.ConvertFrom( i1 ); // квалификатор не нужен
}
```

Определение методов шаблонного класс вне класса

```
template< class T >
class A
{
  void f( T data );
  void g( void );
public:
  A();
};

template< class T >
void A<T>::f( T data ) {}

template< class T >
void A<T>::g( void ) {}
```

Специализация шаблонов

```
template<class T> void f(T t) {
};

template<> void f<char>(char c) {
}

template<> void f(double d) {
}
```

Специализация методов класса

```
template <class T>
class Repository
{
private:
    T m_value;
public:
    Repository(T value)
    {
        m_value = value;
    }

    ~Repository()
    {
    }

    void print()
    {
        std::cout << m_value << '\n';
    }
};
```

```
template <>
void Repository<double>::print()
{
    std::cout << std::scientific << m_value << '\n';
}

int main()
{
    // Инициализируем объекты класса
    Repository<int> nValue(7);
    Repository<double> dValue(8.4);

    // Выводим значения объектов класса
    nValue.print();
    dValue.print();
}
```

Специализация шаблона класса

```
template <class T>
class Repository8
{
private:
    T m_array[8];

public:
    void set(int index, const T &value)
    {
        m_array[index] = value;
    }

    const T& get(int index)
    {
        return m_array[index];
    }
};
```

```
template <>
class Repository8<bool>
{
private:
    unsigned char m_data;

public:
    Repository8() : m_data(0) { }

    void set(int index, bool value)
    {
        // ...
    }

    bool get(int index)
    {
        // ...
    }
};
```

Вопросы

Могут ли в обычном классе (нешаблонном) быть шаблонные методы?

Могут ли в шаблонном классе быть шаблонный метод с другими параметрами?

Могут ли друзья быть шаблонными?

Может ли виртуальный метод быть шаблонным?

Можно ли шаблонный метод переопределять?

Можно ли шаблонный метод перегружать?

Статические методы могут быть шаблонными?

Конструкторы и деструкторы могут быть шаблонными?