Язык C++

Мещерин Илья

Лекция 10

6.8) Variadic templates

```
template<typename ... Types>
void f(Types ... args);
```

Пример использования.

```
1  void print(){}
2
3  template<typename Head, typename ... Tail>
4  void print(Head x, Tail ... t){
5      cout << x << ' ';
6      print(t ...);
7  }
8
9  int main(){
10     print(23, "42", 0.21);
11 }
```

Функция *print()* может принимать переменное число аргументов и выводить их в поток вывода.

```
sizeof ...( t)
```

Оператор, который позволяет узнать количество аргументов в паке аргументов, когда их переменное количество.
Шаблон с переменным числом аргументов будет выступать как наименее предпочтительный.

```
void print(Head x, Tail& ... t){}
```

Позволяет принять аргументы по ссылке.

```
template<class T>
struct St{};

template<typename Head, typename ... Tail>
void print(Head x, St<Tail> ... t){}
```

Так тоже можно

6.8$\frac{1}{2}$) Tuple

```
tuple<int, double, std::string> t = make_tuple(1, 2.0, "3");
get<0>(t); get<1>(t); get<2>(t);
get<int>(t); get<double>(t); get<string>(t);
```

Кортеж, обобщение класса *std::pair*.

6.9) Пример реализации сортировки

```
template<class T, class Cmp>
void sort(T *begin, T *end, Cmp cmp){
    //...
    cmp(*x, *y);
    //...
}
```

Объект *cmp* может быть функцией, а может быть объектом, у которого определен *operator()* от двух объектов типа *T*. По умолчанию *Cmp = std::less<T>*, где *std::less<T>* - класс, в котором определен *operator()*, который вызывает *operator<*.

```
template<class T>
struct less{
    bool operator()(const T &x, const T &y){
        return x < y;
    }
};

int x = 5, y = 10;
less<int> l;
l(x, y);
```

Пример реализации класса *less*. Аналогично существует класс *std::greater*.

6.10) CRTP (Curiously Recurring Template Pattern)

```
1  template<class T>
2  struct Base{
3      void f(){
4          static_cast<T*>(this)->f();
5      }
6  };
7
8  struct Derived : public Base<Derived>{
9      void f(){ cout << "YEAH!"; }
10 };
11
12 int main(){
13     Derived d;                          stdout:
14     Base<Derived> &b = d;                   YEAH!
15     b.f();
16 }
```

Это эффективно эмулирует систему виртуальных функций во время компиляции без необходимости платить цену за динамический полиморфизм (таблицы виртуальных методов, время, затрачиваемое на выбор метода, множественное наследование), но не позволяет делать этого во время выполнения.

Исключения
(Exceptions)

7.1) Общая идея

int *p = **new int**[1000]; std::bad_alloc

Если оператор *new* не смог выделить память, то он может кинуть исключение.

dynamic_cast<Derived&>(b); std::**bad_cast**

Если под *b* на самом деле лежит не *Derived*, то *dynamic_cast* может кинуть исключение.

Концепция исключений: если у функции происходит исключительная ситуация, то она может сгенерировать некоторый объект, который будет сигнализировать о том, что что-то пошло не так. Этот объект создается в специальной области памяти, чтобы он существовал. После чего происходит раскрутка стека вызовов до первого обработчика исключений подходящего типа, и управление передаётся обработчику. Если такой обработчик не был найден, то происходит runtime error (т.к. мы выйдем из функции *main* с **необработанным исключением**).

7.1$\frac{1}{2}$) Try, catch

```
throw x;

try{
    //...
} catch(int x) {
    //...
} catch() {
    //...
}
```

С помощью *throw* можно кидать объект любого типа.

catch(...)

Поймать что угодно.

7.2) Отличие между исключениями и ошибками

Не каждое исключение это runtime error и не каждый runtime error это исключение.

Исключение не обязательно может сигнализировать об ошибке, например, можно выйти из вложенных циклов, бросив исключение и поймав в нужном месте.

7.3) Последовательность и правила обработки исключений

```
try{
    int x;
    throw x;
} catch(double x) {
    cout << 1;
} catch(int x) {
    cout << 2;
} catch(...) {
    cout << 3;
}
```

stdout:
2

Во-первых, когда происходит обработка исключений компилятор не делает приведение типов, кроме приведения типов между родителями и наследниками. Во-вторых, из всех *catch* компилятор выбирает первый, который подходит, а остальные игнорирует.

```
try{
    int* x;
    throw x;
} catch(void *x) {                    stdout:
    cout << 1;                              1
} catch(int *x) {
    cout << 2;
}
```

На *void** поймается любой указатель.

```
struct Base{};
struct Derived : Base{};

try{
    Derived d;
    throw d;
} catch(Base &x) {                    stdout:
    cout << 1;                              1
} catch(Derived &x) {
    cout << 2;
}
```