

академия  
больших  
данных

mail.ru  
group

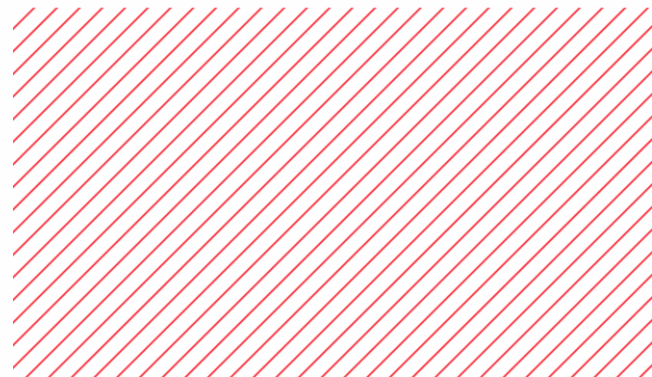


# Templates & Exceptions

## Лекция 5

Гуров Роман

Advanced C++





# Мотивировка

---

```
void swap(int& a, int& b) {  
    int temp = a;  
    a = b;  
    b = temp;  
}
```

```
std::vector<double>
```

Хочется иметь возможность писать универсальный код для разных типов

# Объявление шаблонов

---

```
template <class T>
void swap(T& a, T& b) {
    T temp = a;
    a = b;
    b = temp;
}
```

```
template <typename T>
class Foo {
    // ...
};
```

```
int main() {
    int x = 3, y = 2;
    swap(x, y);
    swap<double>(x, y)
    Foo<int> c;
}
```

Подстановка шаблонных параметров происходит на этапе компиляции

# Специализации шаблонов

---

```
template <class T>  
void f(T x);
```

```
template <>  
void f(int x);
```

```
int main() {  
    f(2.);  
    f(2);  
}
```

Выбирается наиболее узкоспециализированная из подходящих подстановок

# Специализации шаблонов

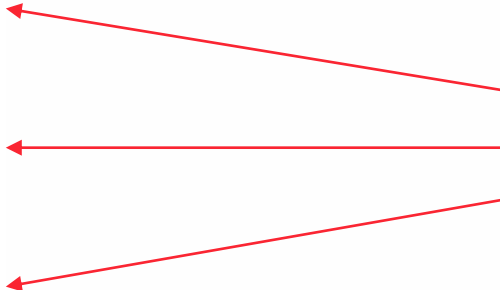
---

```
template <class T1, class T2>  
void g(T1 x, T2 y);
```

```
template <class T>  
void g(int x, T y);
```

```
template <>  
void g(int x, int y);
```

```
int main() {  
    g(2., 2.);  
    g(2, 2.);  
    g(2, 2);  
}
```



# Специализации шаблонов

---

```
template <class T1, class T2>
class A {};
```

```
template <class T>
class A<int, T> {};
```

```
template <class T>
class A<T, int> {};
```

```
int main() {
    A<int, int> a;
}
```

Непонятно, какую специализацию  
вызвать – ошибка компиляции

# Специализации шаблонов

---

```
template <class T1, class T2>
class A {};
```

```
template <class T>
class A<int, T> {};
```

```
template <class T>
class A<T, int> {};
```

```
template <>
class A<int, int> {};
```

```
int main() {
    A<int, int> a;
}
```

Всё хорошо, вызовется `<int, int>`



# Typedef

---

```
typedef LongName<int, std::vector<int>> NewName;
```

```
template <class T>  
using NewName = LongName<int, std::vector<T>>>;
```

Объявляется новое название для существующего типа



# std::remove\_const, std::remove\_reference

---

```
template <class T>
struct remove_const {
    typedef T type;
};

template <class T>
struct remove_const<const T> {
    typedef T type;
};
```

```
int main() {
    remove_const<const int>::type b;
}
```

int b; const





# std::remove\_const, std::remove\_reference

---

```
template <class T>
struct remove_reference {
    typedef T type;
};

template <class T>
struct remove_reference<T&> {
    typedef T type;
};
```

```
int main() {
    remove_reference<int&>::type b;
}
```

# Вывод шаблонных типов

```
template <class T>
void f(T& x);

int main() {
    int x;
    int& y = x;
    const int z = 0;
    const int& t = z;
}
```

T&	T	x
f(x)	<b>int</b>	<b>int&amp;</b>
f(y)	<b>int</b>	<b>int&amp;</b>
f(z)	<b>const int</b>	<b>const int&amp;</b>
f(t)	<b>const int</b>	<b>const int&amp;</b>

# Вывод шаблонных типов

---

```
template <class T>
void f(T x);

int main() {
    int x;
    int& y = x;
    const int z = 0;
    const int& t = z;
}
```

T&	T	x
f(x)	<b>int</b>	<b>int</b>
f(y)	<b>int</b>	<b>int</b>
f(z)	<b>int</b>	<b>int</b>
f(t)	<b>int</b>	<b>int</b>

# Non-type template parameters

---

```
template <class T, int n>
class array {
    // ...
};
```

```
int main() {
    array<int, 10> a;
    array<int, 20> b;
}
```

Можно заложить какое-то значение прямо  
в тип данных на этапе компиляции

# Template template parameters

---

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

```
template <template <class> class T, class X>
void g(const T<X>& t) {
    T<double> t_double;
    X x;
}
```

```
int main() {
    f<C, int>();
    g(C<int>());
}
```



# Variadic templates

---

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

```
void g() {}
```

```
template <typename Head, typename... Tail>
void g(Head h, Tail... t) {
    std::cout << h << ' ';
    g(t...);
}
```

```
int main() {
    g(1., 2, "three");
}
```



# Функторы

---

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

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

```
int main() {
    int a[100];
    sort(a, a + 100, less<int>());
}
```





# Исключения

---

```
int* do_something(size_t input, int& error);  
// ...  
int error;  
int* result = do_something(10, &error);  
if (error) {  
    // handle error  
}
```

Неудобно писать так для каждого вызова



# Исключения

---

```
throw C ();
```

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

Исключение != ошибка времени выполнения  
Исключение – способ передачи сообщений в программе



# Правила обработки исключений

---

```
try {  
    throw int();  
} catch (double x) {  
  
} catch (int x) {  
  
} catch (...) {  
  
}
```

Выбирается первый подходящий **catch**  
Преобразование типов допускается только от наследника к родителю

# Повторное бросание исключения

---

```
try {  
    try {  
        throw int();  
    } catch (...) {  
        throw;  
    }  
} catch (int x) {  
  
}
```

Внешний **catch** поймает **int**



# Копирование при исключениях

---

```
try {  
    std::string s;  
    throw s;  
} catch (std::string& s) {  
  
}
```

**throw** всегда копирует бросаемый объект

# Копирование при исключениях

```
try {  
    Derived d;  
    Base& b = d;  
    throw b; ← Скопируется как Base  
} catch (Base& b) {  
    dynamic_cast<Derived&>(b);  
}
```

Ошибка

```
try {  
    Derived d;  
    throw d;  
} catch (Base& b) {  
    dynamic_cast<Derived&>(b);  
    throw;  
}
```

**throw** не учитывает полиморфность классов при копировании

# Спецификации исключений до C++11

---

```
void f() throw(double) {  
    throw 1;  
}
```

Обещаем бросать  
только **double**

Компилятор позволяет бросать неожиданные исключения,  
но при выполнении программа крашнется

# Спецификации исключений в C++11

---

```
void f() noexcept {  
    throw 1;  
}
```

Обещаем ничего  
не бросать

Спецификатор **noexcept** не позволяет указать бросаемые типы





# Оператор noexcept

---

```
void f() noexcept {}
```

```
noexcept(f) == true
```

Оператор **noexcept** на этапе компиляции проверяет, объявлена ли данная функция с **noexcept**



# Условный noexcept

---

```
template <class T>
void f(T x) noexcept(noexcept(g(x))) {
    g(x);
}
```

Спецификатору **noexcept** можно передать **constexpr** значение булевского типа

# Исключения в конструкторах

---

```
struct Ugly {  
    int* p;  
    Ugly() {  
        p = new int;  
        throw 1;  
    }  
    ~Ugly() {  
        delete p;  
    }  
};
```

```
int main() {  
    try {  
        Ugly u;  
    } catch (...) {}  
}
```

Деструктор класса не будет вызван,  
что приведёт к утечке памяти

# Исключения в деструкторах

---

```
struct Ugly {  
    ~Ugly() {  
        throw 1;  
    }  
};
```

```
int main() {  
    try {  
        Ugly u;  
        throw 1;  
    } catch (...) {}  
}
```

В процессе бросания исключения можно наткнуться  
на другое исключение в деструкторе



# Гарантии безопасности при исключениях

---

```
vector.push_back(10);
```

Строгая гарантия	Нестрогая гарантия
Состояние программы возвращается к моменту прямо перед вызовом	Состояние программы остается валидным