

- механизм работает только для полиморфных классов.

Оператор `typeid` и класс `type_info`

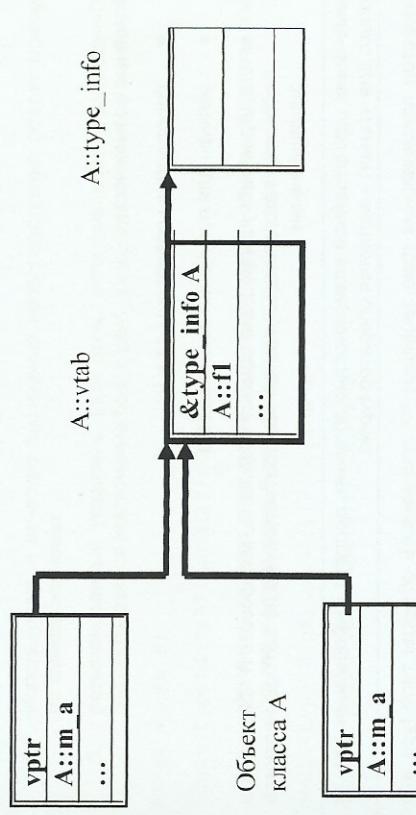
Идентификация типов позволяет получить информацию об объектах во время выполнения. Это означает, что кто-то и где-то должен эту информацию сформировать, а именно:

1. отвести память (где?)
2. сформировать данные (какие?)
3. обеспечить механизм использования этой информации (как?).

Логично:

1. сколько бы объектов класса `X` мы не создавали, достаточно хранить информацию собственно о типе `X` в единственном экземпляре в каждом классе => для каждого класса компилятор создает объект типа `type_info`
2. должен существовать способ получения этой информации для каждого объекта =>
 - а) доступ к объекту `type_info` можно получить при помощи оператора `typeid`
 - б) неявно – посредством оператора `dynamic_cast`.
3. Что существует в единственном экземпляре для всех объектов? – статические переменные или таблицы виртуальных функций. В спецификации языка сказано, что получение этой информации гарантируется только для полиморфных типов (то есть если есть хотя бы одна виртуальная функция). Данные RTTI выполняют примерно ту же задачу, что и таблицы виртуальных функций => для поддержки RTTI была использована именно таблицы виртуальных функций класса. Например индекс 0 в таблице может содержать указатель на объект `type_info`.

Объект
класса A



- При такой реализации память будет тратиться только на добавление еще одной ячейки в каждую таблицу виртуальных функций + выделение памяти для хранения объекта `type_info` для каждого класса.

1.1.25. Формат `type_info`

Заголовочный файл `typeinfo.h`:

```

class type_info {
public:
    virtual ~type_info();
    int operator==(const type_info& rhs) const;
    int operator!=(const type_info& rhs) const;
    int before(const type_info& rhs) const; // не только
                                              // непосредственного
                                              // предка, но и
                                              // любого в иерархии!
    const char* name() const; // имя, которое дал программист
    const char* raw_name() const; // декорированное имя
private:

```

```
type_info (const type_info&);

type_info& operator=(const type_info&);

};


```

Основные возможности:

- перегружены операторы `==` и `!=`.
- методы для получения имени класса.
- программисту запрещено самому создавать, копировать и присваивать объекты такого класса, так как все соответствующие методы являются защищенным

Замечание: единственной возможностью получить доступ к информации явно (и только для чтения) – является оператор `typeid`.

Оператор присваивания тоже защищен => присваивать объекты типа `type_info` компилятор Вам тоже не позволит!

1.1.26. Формы оператора typeid:

```
const type_info& typeid(type);

const type_info& typeid(expression)
```

Так как определение типа имеет смысл только при наследовании, примеры приводку на простенской иерархии:

```
class A {
public:
    virtual void f() {}
```

```
};

class B : public A {};
```

В качестве выражения может использоваться:

1. ссылка на класс

```
B b;
A& ra = b;
```

<http://www.avalon.ru>

Школа Практического Программирования

ФПС СПбГУ

```
cout<<typeid(ra).name() ; // "class B"
При этом даже если в качестве выражения фигурирует ссылка на базовый класс, результат typeid-оператора -const type_info& целевого класса.
```

2. Использовать в качестве выражения непосредственно объект можно, но смысла не имеет
3. разыменованный указатель (при этом, если значение указателя==0, вырабатывается исключение `bad_typeid_excepiton`)

```
A* pa = new B;
cout<<typeid(*pa).name() ; // "class B"
4. Не имеет смысла в качестве выражения использовать указатель, так как оператором typeid будет возвращено typeinfo типа указателя
cout<<typeid(pa).name() ; // "class A"
cout<<typeid(pb).name() ; // "class B"
```

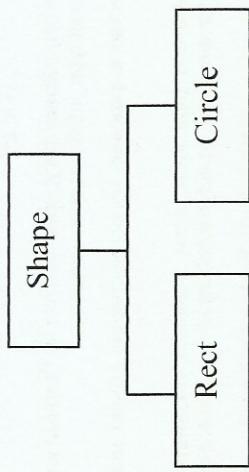
Замечания:

1.
 - если используется недействительный указатель
 - или код не был скомпилирован с ключом /GR
 2. для совместимости с базовыми типами компилятор для каждого базового типа создает объект `type_info`, поэтому для базовых типов тоже можно использовать оператор typeid:
- ```
if (typeid(n)==typeid(int)) ...
```
3. оператор typeid может быть использован для получения объекта `type_info` для параметра шаблона (глава "Обобщенное программирование")

### Пример использования оператора typeid

Дана иерархия классов:

+7(812)703-0202



case 0:

return new Rect;

case 1:

return new Circle;

}

Требуется посчитать, сколько прямоугольников и сколько кружков создала функция и при создании любого объекта вывести тип созданной фигуры.

```

class Shape{
public:
 virtual void F()=0;
};

class Rect: public Shape{
 ...
public:
 virtual void F(){cout<<"I'm Rect"<<endl;}
};

class Circle: public Shape{
 ...
public:
 virtual void F(){cout<<"I'm Circle"<<endl;}
};

Shape* Make()
{
 switch(rand()%2)
 {
 ...
 }
}

int main()
{
 const int n = 10;
 Shape* ar[n];
 int nRects=0, nCircles=0; // Здесь подсчитаем - сколько чего как создалось
 for(int i = 0; i<n; i++)
 {
 ar[i] = Make(); //создание очередного объекта
 cout<<typeid(*ar[i]).name()<<endl; //для диагностики
 //что создали
 if(typeid(*ar[i]) == typeid(Rect)) nRects++; //подсчет
 else nCircles++;
 }
 for(int i = 0; i<n; i++)
 {
 delete ar[i];
 }
}

Функция - фабрика фигур, которая случайным образом генерирует динамический объект одного из производных типов и возвращает указатель базового типа:
Shape* Make()
{
 ...
}

```

**Использование других возможностей класса `type_info`** – определение порядка следования классов в метархии наследования. Задана иерархия классов:

```
class A{... virtual...};
class B: public A{...};
class C: public B{...};

int main()
{
 B* pB = new C;
 A* pA = pB;

 if (typeid(A).before(typeid(*pA))) // ???
 if (typeid(B).before(typeid(*pA))) // ???
 if (typeid(C).before(typeid(*pA))) // ???
 if (typeid(B).before(typeid(*pB))) // ???
 ...
}
```

### Оператор `dynamic_cast`

Необходимость RTTI обусловлена тем, что при компиляции не всегда есть возможность выяснить: на какой целевой объект указывает указатель базового типа (или ссылается ссылка базового типа).

Восстановление "потерянного" типа объекта требует, чтобы была возможность "спросить" объект о его типе, а также должна быть операция преобразования типа, которая возвращает базовый указатель, если объект действительно имеет целевой тип, или «сообщала» о невозможности преобразования.

Формат:

```
dynamic_cast<T*>(pointer) //возвращает указатель типа T*, если
преобразование корректно, или 0
dynamic_cast<T&>(reference) //возвращает ссылку типа T& если
преобразование корректно, или вырабатывает исключение std::bad_cast
Модифицируем предыдущий пример:
int main()
```

<http://www.avalon.ru>

Школа Практического Программирования

ФПС СПбГПУ

Оператор `dynamic_cast` используется для проверки возможности приведения, поэтому:

```
class A{... virtual...};
class B: public A{...};
class C: public B{...};

int main()
{
 A* pA = new C;
 B* pB = dynamic_cast<pA>; // != 0
 C* pC = dynamic_cast<pA>; // != 0
 delete pA;
```

```

}
}

Замечание: dynamic_cast не работает с void* - указателями

```

### Операторы typeid и dynamic\_cast и наследование

Продемонстрируем разницу в использовании операторов typeid и dynamic\_cast. Напоминаю: оператор typeid используется для определения точного типа, а dynamic\_cast для проверки возможности приведения.

Задана иерархия классов:

```

class A{... virtual ...};
class B : public A {...};
class C : public B{...};

int main()
{
 A* pA = new C;
 C* pC = dynamic_cast<C*>(pA); // !=0
 B* pB = dynamic_cast<B*>(pA); // !=0
 // Но!!!
 bool b = typeid(*pA) == typeid(C); // true
 b = typeid(*pA) == typeid(B); // false
}

```

При множественном наследовании механизм RTTI тоже работает, так как в объекте typeid есть вся информация как о производном типе, так и обо всех предках:

Задана иерархия классов:

```

class A{... virtual ...};
class B {... virtual ...};
class C : public A, public B{...};

int main()

```

## Тема XII. Исключительные ситуации (exceptions)

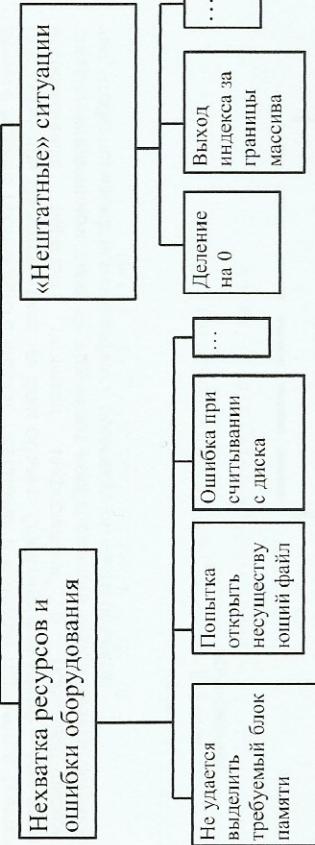
### Что такое исключительные ситуации.

При написании сколько-нибудь серьезных приложений большую роль играет обработка всевозможных ошибочных ситуаций, которые могут возникнуть во время работы приложения. Ошибочные ситуации можно разделить на две большие категории:

От программиста требуется только отследить результат

Программист не предусмотрел обработку

### Исключительные ситуации



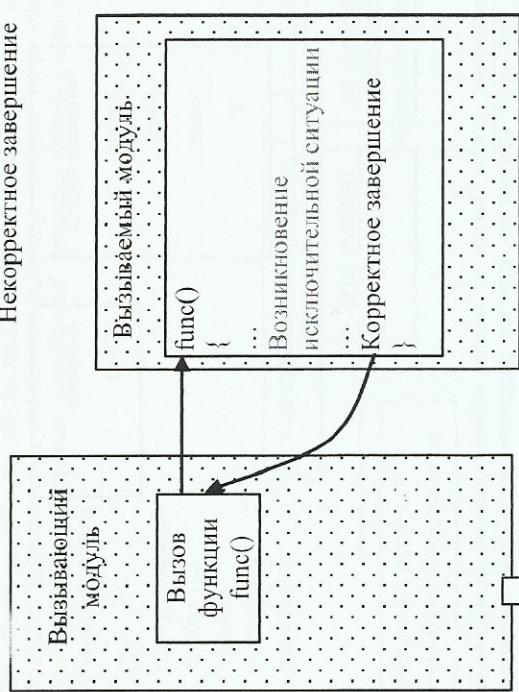
Если Ваше приложение не отреагирует на такую ситуацию, то, скорее всего Вы увидите сообщение "Unhandled exception", Ваше приложение будет аварийно завершено, ... а Вашу программу вряд ли кто-нибудь купит.

Проблема возникает тогда, когда в месте возникновения ошибочной ситуации у Вас «нет контекста» для ее разрешения. Обычно ситуация «неподконтрольных» проблем появляется в случае разработки большого проекта несколькими программистами или использования независимо разработанных библиотек. При этом может возникнуть следующая ситуация:

?

Вызывающий модуль знает –  
какая ошибка, но не знает, что  
с ней делать

Некорректное завершение



При некорректном завершении  
автора вызывающего модуля  
должен сгенерировать  
информацию об ошибке, так  
как «знает», что за ошибка,  
но не знает – что с ней делать

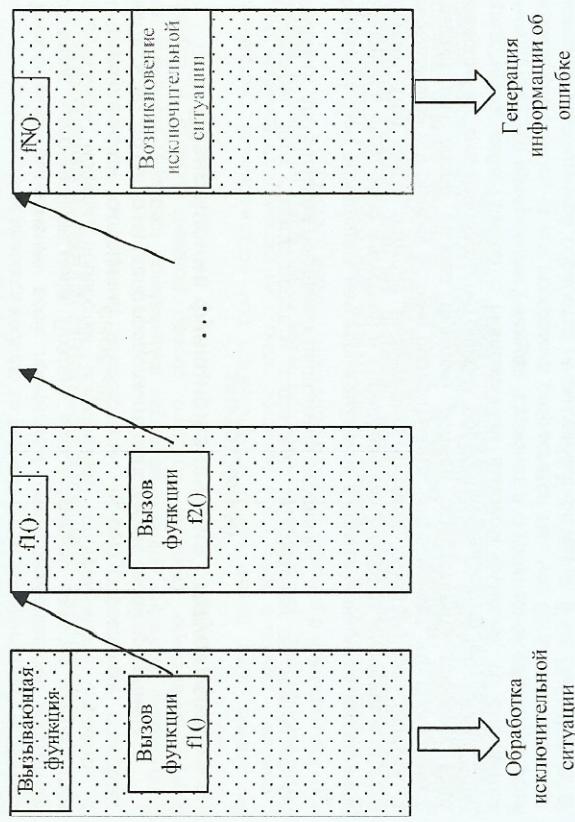
**ЭТОМ СЛУЧАЕ НУЖНО  
сделать**

То есть автор вызывающего модуля может обнаружить «неприятности» во время  
выполнения, но, как правило, не имеет представления о том, что делать в этом  
случае. Автор вызывающего модуля может знать, как поступить в случае  
возникновения ошибок, но не в состоянии их обнаружить.

<http://www.avalon.ru>

Школа Практического Программирования

Если же «вложенность» вызовов функций значительна, то возникновение исключительной ситуации может возникнуть на нижнем уровне, а обработка ошибки должна быть произведена на самом верхнем уровне.



Замечание 1: Но! Механизм обработки исключений дорогой => Следует отметить, что целью обработки исключений является решение «неподальных» по своей природе проблем. Поэтому, если проблема может быть решена локально, почти всегда так ее и следует решать.

Обработку исключений имеет смысл использовать тогда, когда можно восстановить (починить) программу для продолжения работы.

Замечание 2: существуют два уровня обработки исключений:

- структурная обработка исключений (SEH) – используется в основном для обработки hardware-ных исключительных ситуаций (деление на 0, обращение по недействительному адресу...);
- обработка исключений C++

## Способы отслеживания «аварийных» ситуаций

### Прекращение выполнения программы

Прекращение выполнения всей программы при возникновении нештатной ситуации – это крайняя мера. Поэтому еще в древние времена (до развития физической модели обработки исключений) программисты боролись с непредвиденными ситуациями множеством неструктурированных способов, пытаясь, во что бы то ни стало продолжить выполнение своей программы.

### Возвращение функцией значения статуса (кода завершения)

Замечание: этот подход прост и достаточно эффективен => никто его не отменяет для решения локальных нештатных ситуаций.

Функция может возвращать некоторое значение в случае успешного завершения (например, ненулевое значение) и определенное (нулевое) значение в случае ошибки. При таком подходе вызываемая функция обнаруживает ошибку, а вызывающая эту ошибку должна обрабатывать. Это означает, что программист каждый раз после вызова функции должен осуществлять проверку кода завершения, анализировать тип ошибки, обрабатывать данный тип ошибки. Если представить себе пример с N-вложенным вызовом функций, то становится очевидным, что **программист-зануда** погрязнет в скучнелезной обработке всех возможных ошибок (а его программа разрастается так, что большая часть кода окажется посвященной обработке неожиданных ситуаций, а не ожидаемых). В той же ситуации **среднеответственный программист** пренебрежет обработкой большей части ошибок, расчитывая только на штатные ситуации...

Кроме того, если для обработки ошибки не обойтись одним возвращаемым значением (то есть важен не только сам факт возникновения ошибки, но и ее тип)? Для запоминания типа ошибки обычно использовалась глобальная переменная (например, это содержит код ошибки).

### Статус завершения системных вызовов

В стандартной библиотеке есть глобальная переменная `errno`, которая предназначена для формирования кода системных ошибок. В принципе, любой системный вызов это значение формирует, но так как переменная глобальная, код ошибки можно получить только непосредственно после вызова системной функции.

```
void (*old_handler)(void) = std::set_new_handler(
 my_handler); // Установили свой обработчик
```

## Выявление ошибок в Debug-версии

Диагностический макрос `assert` – хорош для выявления тех ошибок, которые зависят от программиста, но не спасет от ошибок времени выполнения.

### Обработка ошибок с помощью функций обратного вызова

Вызывающая функция (клиент) передает вызываемой (серверу) указатель на другую клиентскую функцию, которую сервер должен вызвать в случае возникновения ошибки.

Замечание: этот прием часто используется при использовании системных сервисов – смысл в том, что в качестве одного из параметров функции или каким-нибудь еще способом (в качестве поля структуры) передается адрес пользовательской функции – функции обратного вызова (callback).

В C++ эта технология, например, используется для обработки отказов при динамическом распределении памяти. Функция `set_new_handler()` позволяет Вашей программе установить Ваш специализированный обработчик ситуации, который будет вызываться системой в случае нехватки памяти вместо стандартного. Причем, «подменять» стандартный обработчик своим можно только на некоторое (потенциально опасное) время. Например:

```
#include <new>
size_t BIG_NUMBER = 0xffffffff;
int* p;
void my_handler(void)
{
 // Ваш обработчик
 BIG_NUMBER /= 2;
}
```

```
int main(void)
{
 //
```

```

p = new int [BIG_NUMBER]; //если памяти не хватает, будет
 вызван ваш обработчик, где вы
 постараитесь справляться с
 ситуацией
}

std::set_new_handler(old_handler); //восстановили
 прежний обработчик
}

```

Замечание: по умолчанию указатель, возвращаемый `set_new_handler`, равен нулю.

### Выполнение глобального перехода при возникновении ошибки

Функция `setjmp()` позволяет программисту запомнить точку возврата и состояние программы на момент выполнения `setjmp()` в структуре предопределенного вида. Функция `longjmp()` позволяет вернуть управление посредством сохранных данных в структуре `jmp_buf`.

```

//client.cpp

#include <csetjmp>
jmp_buf buf;
int main()
{
 ...
 int ret = setjmp (buf);
}

```

При этом она корректно восстанавливает стек. Неудобство – буферов может быть много...

Будучи перенесена в C++, функция `longjmp()` приобрела один существенный недостаток – она не вызывает деструкторы локальных объектов.

### Малоизвестная и малоиспользуемая часть старой стандартной библиотеки C

Функции

`signal()` – для определения – что произошло  
`raise()` – для генерации события

<http://www.avalon.ru>

Школа Практического Программирования

Поведение по умолчанию `signal` означает аварийное завершение программы с кодом 3. Проблема – та же самая + номера ошибок быть уникальными.

### Встроенные средства C++ для обработки исключений (C++ exception handling)

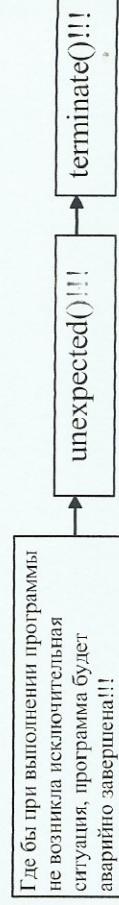
Все перечисленные выше способы обработки предстают собой набор разнородных средств, предназначенных для решения разных аварийных ситуаций.

Встроенная поддержка обработки исключений в C++ появилась сравнительно недавно. Достоинствами механизма обработки исключительных ситуаций являются:

- обработку нештатных ситуаций все равно предусматривать нужно.
- ситуации разные, а способ борьбы с ними **унифицирован**.
- **структура** текста программы улучшается. Программист может отдельить написание рабочего кода (основная работа) от написания обработки нештатных ситуаций (и отложить это рутинное удовольствие «на потом», то есть предполагается, что к этому можно вернуться и позже)

Идея – нужно передать управление и информацию об ошибке из того места, где она возникла, туда, где она может быть обработана (мы будем все промежуточные инстанции) – Похоже на посылку сообщения

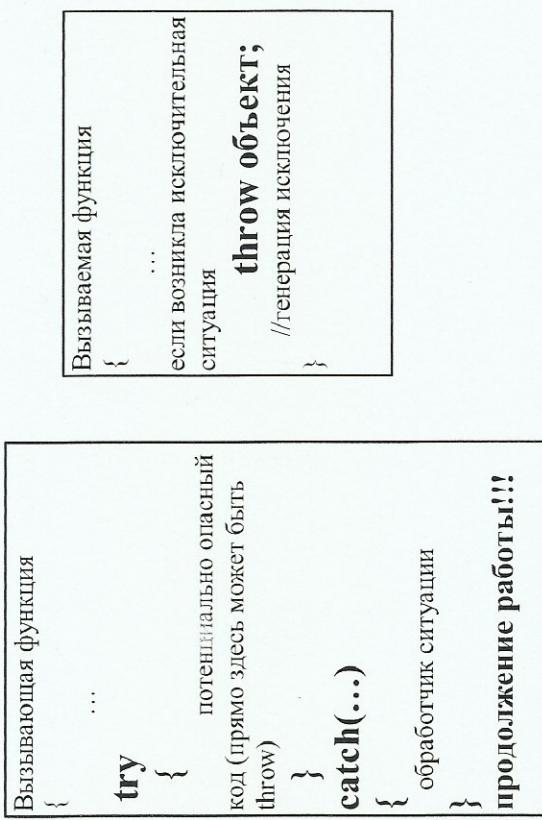
Если обработка исключений не предусмотрена, при возникновении любого исключения программа будет завершена аварийно!!!



Язык C++ включает встроенные средства для обработки аномальных ситуаций (exceptions), возникающих во время выполнения программы – операторы `throw`, `catch`, `try`. Оператор `try` откапывает блок кода, в котором может произойти ошибка. Если ошибка произошла, то оператор `throw` вызывает исключение. Обработчик исключения представляет собой блок кода, который начинается оператором `catch`.

Если предусмотреть обработку исключений:

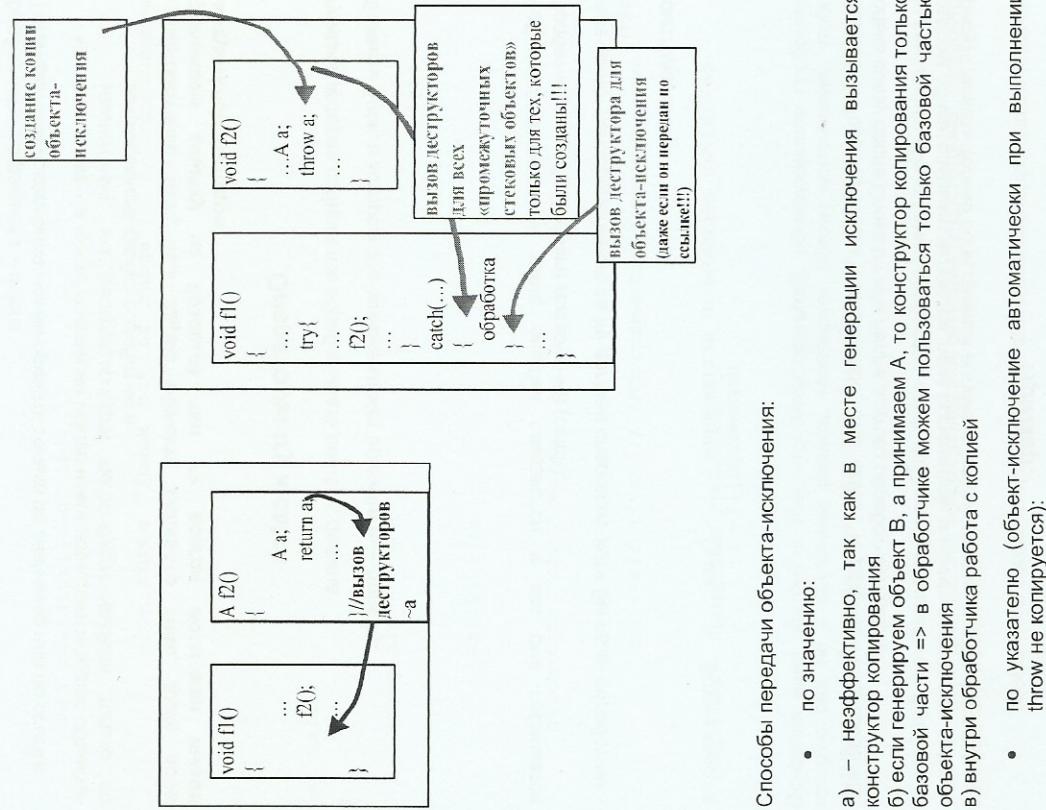
ФПС СПбГУ  
+7(812)703-0202



### Оператор **throw** – сравнение с **return**

Оператор **throw** очень похож на инструкцию **return**, но главные отличия:

- **куда** происходит передача управления!!!
- **где** вызываются деструкторы для всех промежуточных стековых объектов, которые были созданы на момент генерации исключения (в частности локальных объектов вызываемой функции)
- **как** передается объект-исключение и где вызывается для объекта-исключения деструктор



Способы передачи объекта-исключения:

- по значению:
  - неэффективно, так как в месте генерации исключения вызывается конструктор копирования
  - если генерируем объект B, а принимаем A, то конструктор копирования только базовой части => в обработчике можем пользоваться только базовой частью B) внутри обработчика работа с копией
- по указателю (объект-исключение автоматически при выполнении **throw** не копируется):
  - адрес локального объекта – плохо, так как для него будет вызван деструктор
  - адрес динамического объекта плохо, так как **нет** уверенности, что в

- обработчике программист – `delete`
- в) стандартные исключения можно принять только по значению или по ссылке
- по ссылке – в месте генерации исключения передается адрес объекта-исключения, но деструктор для него будет вызван только по закрывающей скобке обработчика
- В обработчике получаем адрес объекта целевого типа, даже если принимаем ссылку на базовый тип => всегда возможен вызов виртуальных методов!**

### Операторы `try` и `catch`

Функция, которая собирается обрабатывать ошибку, должна:

объявить блок, в котором эта ошибка может возникнуть – блок `try`.

```
try{
 ...
}
```

сообщить компилятору какие ошибки ожидаются и как она собирается обрабатывать – обработчики исключений (`catch`).

Замечание: непосредственно за `try` должен следовать хотя бы один обработчик – `catch`

Конструкция

```
catch(тип_переменной_исключения [имя параметра]) {
 ...
}
```

называется обработчиком исключения. Она может и должна использоваться только сразу после блока `try` или после другого обработчика `catch`.

По синтаксису объявление оператора `catch` похоже на определение функции, в котором обявляемыми являются типы аргументов, а имена аргументов должны присутствовать только если данный аргумент задействован в коде функции.

### Пример:

```
class Array{
 int m_ar[10];
}
```

<http://www.avalon.ru>

Школа Практического Программирования

ФПС СПбГУ

+7(812)703-0202

```
public:
 int& operator[](int i) {
 if(i>=0 && i<10) return m_ar[i];
 else throw "Index out of range"; } //Синтаксическая
 оператор throw похож на оператор
 return.
 };
}
```

```
int main()
{
 Array ar;
 try{
 ar[10] = 1;
 }
 catch(const char* error)
 {
 cout<<error<<endl; //программисту предупреждение
 }
 // выполнение программы продолжается
}
```

Основная идея состоит в том, что `Array::operator[]`, обнаружившая проблему, которую она не в состоянии решить, генерирует (`throw`) исключение, чтобы вызывающая функция «не затащила» что-нибудь за пределами выделенной памяти.

Обычно одной диагностикой о произошедшей ситуации недостаточно. Например, в нашем случае не плохо:

- сообщить в вызывающую функцию значение «некорректного» индекса – `throw i;`
  - вырабатывать исключение более сложного типа, чтобы в объекте такого типа можно передать гораздо больше информации
- ```
struct MyArrayError{
```

Обработка нескольких типов исключений

Обычно в программе в процессе выполнения может возникнуть не одна, а **несколько ошибочных ситуаций разного типа**. Хорошой практикой является для каждой такой ситуации определить свой тип исключения. В общем случае структура блока программы, в котором могут возникнуть исключения, выглядит следующим образом:

```

int m_wrongIndex;
int m_High;

MyArrayError(int n, int c){m_wrongIndex=n; m_High=c;};

class Array{
    int m_ar[10];
public:
int& operator[](int i)
{
    if(i>=0 && i<10) return m_ar[i];
    else throw MyArrayError(i, 10);
}
};

int main()
{
    Array ar;
try{
    ar[10] = 1;
}
catch(MyArrayError& error)
{
    cout<<"index out of range"<<error.m_wrongIndex<<"index must
be>=0 or <"<<error.m_High<<endl;
}// вызывается деструктор для объекта -исключения
}

```

```

struct Above{
    int m_wrongIndex;
    int m_High;
    Above(int n, int c){m_wrongIndex=n; m_High=c;};
};

struct Below{
    int m_wrongIndex;
    Below(int n){m_wrongIndex=n;};
};

int& Array::operator[](int i){
    if(i<=0) throw Below(i);
    if(i>10) throw Above(i, 10);
    return m_ar[i];
}

try{
    ...
}
catch(Above& er){
    cout<<...;
}
catch(Below& er){
    cout<<...;
}

```

4. Выполнить любое восстановительное действие и продолжить работу программы

```
int I=0;
```

```
int BIG = 0x7fffffff;
```

```
double** p=0;
```

Замечания:

- каждый обработчик соответствует своему конкретному типу ошибки (то есть при возникновении ошибки данного типа, управление будет передано именно сюда).
- В скобках любого оператора catch указывается тип объекта, который может быть перехвачен данным обработчиком.

Если нет соответствующего обработчика, вызывается `operator new`

Обработчики «просматриваются» в том порядке, в котором их привел в программист, поэтому в том случае, когда типы исключений связаны наследованием, обработчик с объектом исключения производного типа должен предшествовать обработчику с исключением базового типа.

Можно предусмотреть обработку всех явно не указанных в обработчиках catch типов исключений посредством:

```
catch (...) {
```

```
cout<<"Something is wrong!";
```

Замечание: если существует такой обработчик «по умолчанию», он должен быть последним в списке обработчиков

1. Вызвать `abort()`, если ситуацию поправить невозможно

2. Повторно сгенерировать то же самое исключение – то есть передать его выше

```
catch (...) //throw; //гетхигу
```

3. Обработать на своем уровне текущее исключение, а для уровня «выше» сгенерировать другое исключение (запаковать другую информацию)

```
catch (My1& x) { ... throw My2(); ... }
```

http://www.avalon.ru Школа Практического Программирования

- Если исключение сгенерировано, и ни один обработчик не перехватил его, выполнение программы прерывается специальной функцией стандартной библиотеки - `uncaught`.

Function-level try-блок

Можно все тело функции заключить в блок `try`. При этом последовательность выполнения кода отмечена цифрами 1-2-3

```
void MyErr (void) try
{
    throw "error"; // 1
}
catch (const char* p)
{
    std::cout<<p; // 2
}
int main()
{
    MyErr(); // 3
}
```

Последовательность выполнения кода и выбор исключений.

- Если код в `try`-блоке (или код функции, вызываемой из этого блока) генерирует исключение, будет проверяться обработчики данного блока `try`. Остаток блока ту игнорируется!
- Если сгенерированное исключение имеет тип, указанный в одном из обработчиков, будет выполнен этот обработчик.
- Если исключение не генерируется, то все обработчики игнорируются.

Обработка непредусмотренных исключений

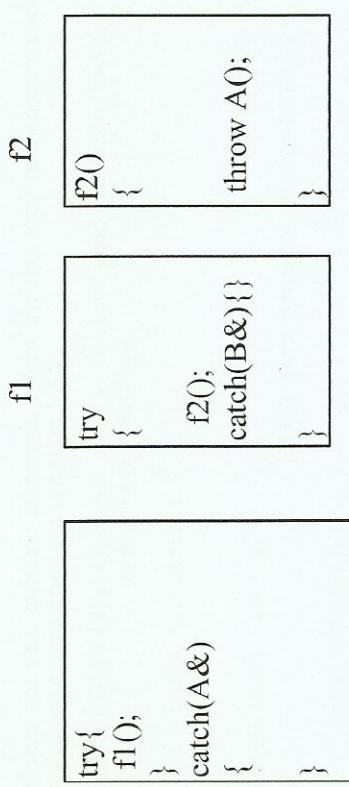
1. Программист может предусмотреть обработчик «по умолчанию»

```
catch (...) {
```

// здесь можно обработать все остальные исключения

}
- При этом такой обработчик должен быть последним!!!

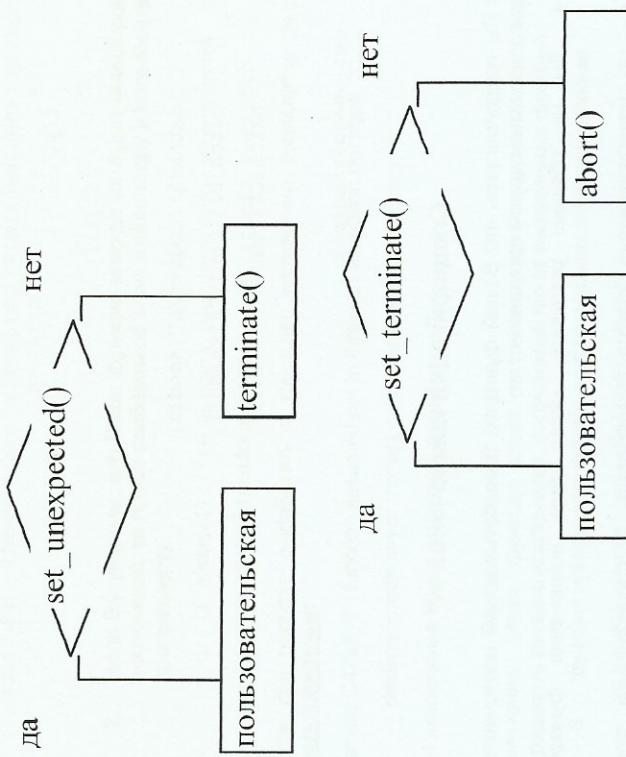
2. Что происходит, если обработка предусмотрена на другом уровне
 Если самый внутренний catch исключение не обработал, оно «передается»
 выше на следующий уровень. Поиск продолжается до самого внешнего
 блока try



3. Что происходит, если программист **нигде не предусмотрел обработку** какого-то конкретного или всех исключений.

Замечание: если возникает неперехваченное исключение, никакого clean_up не происходит (деструкторы не вызываются)

Обработка неперехваченных исключений



Пример:

```

#include <stdexcept>
void my_unexpected()
{
    cout<<"I'll be back!!"<<endl;
    exit(1);
}

int main()
{
  
```

```

void (*pF) () = set_unexpected (my_unexpected) ;
// F2 () ;

for (int i=1; i<=3; i++)
{
    try {
        f1 (i) ;
    }
    catch (A) { cout<<"A"<<endl; }
    catch (B) { cout<<"B"<<endl; }
    catch (C) { cout<<"C"<<endl; }
}
set_unexpected (pF) ;
}

```

Спецификация исключений

Если Вы предполагаете, что Вашей функцией (вырабатывающей исключение)

может пользоваться другой программист, то
 • Вы не обязаны указывать, какого типа исключение функция

вырабатывает,
 • но цивилизованный подход заключается в спецификации этих исключений.

Объявление функций с позывением механизма исключений тоже изменилось => появилось новое понятие – спецификация исключений. Пока можно рассматривать спецификацию исключений в качестве одной из возможностей документирования текста программы.

Замечание 1: спецификация исключений должна быть указана как при объявлении, так и при определении функции!

Замечание 2: спецификация исключений функции на самом деле программиста ни к чему не обязывает, то есть, несмотря на «обещания» функция может выработать исключение любого вида, а компилятор выдаст только предупреждение. Исключение будет сгенерировано и по общим правилам обработано или не обработано!!!

Специфика оператора new

В старых версиях C++ требовалось, чтобы оператор new в случае нехватки памяти возвращал 0. Сейчас new должен генерировать исключение std::bad_alloc, но существует ограничение количества кода, написанного до изменения спецификации => поэтому на самом деле существуют **несколько форм оператора new**.

- 1) void *operator new(size_t n); // если памяти не хватает, вырабатывается исключение
- 2) array new and delete

Пример:

#include <new>

```

{
    A* p;
    try {
        p = new A; // компилятор интерпретирует как p = new
                    // (sizeof(A));
        // генерирует std::bad_alloc
    }
    if (p==0) ... // это условие не выполнится никогда!!!
}

catch (bad_alloc& a) {
    cout << a.what () << endl; // выведет диагностику
    p=0; // обезопасили себя от использования
          // недействительного указателя
}

```

<http://www.avalon.ru>

Школа Практического Программирования

3), 4) продолжают существовать для совместности с предыдущими версиями для одиночного объекта и для массива – **возвращают указатель**

```

void *operator new(size_t n, const notrow_t*); // если памяти не хватает, возвращается нулевой указатель, а исключение не вырабатывается

void operator delete(notrow_t*, void*); // соответствующий delete

```

```

{
    A* p = new(notrow) A[2]; // notrow – это глобальный объект типа notrow_t
    if (p==0) ...

    delete[] (notrow, p);
}

```

Исключения в конструкторах и деструкторах

Объект не считается созданным, пока не завершится выполнение его конструктора => для «недоделанных» объектов деструкторы не вызываются!!

Проблема: другими способами ситуацию разрешить сложно, так как конструкторы и деструкторы в C++ не позволяют возвращать значений => если ошибка все-таки возникает именно в конструкторе, то (чтобы не пользоваться «недоделанным» объектом) нужно как-то об этом сообщить:

- посредством глобальных переменных
- с помощью механизма исключений (основное преимущество – будет вызваны деструкторы для всех объектов, созданных в конструкторе на момент выработки исключения, например, если создавался массив объектов, то будут вызваны деструкторы только для тех элементов массива, которые уже удалось создать)

ФПС СПбГУ

+7(812)703-0202

а) Встроенные объекты (все корректно, так как для всех промежуточных стековых объектов будут вызваны деструкторы)

```
class A{
    int m_a;
public:
    A(int a=0){m_a = a;}
    ~A() {std::cout<<"~A ";}
};

class B{
    int m_b;
public:
    B(int b=0){m_b = b; throw 1;}
    ~B() {std::cout<<"~B ";}
};

class C{
    A m_A;
    B m_B;
public:
    C(int a, int b):m_A(a), m_B(b) {}
    ~C() {std::cout<<"~C ";}
};

int main()
{
    try{
        C c(1,2);
        catch(int)
        {
            std::cout<<"Exception " << i << std::endl;
        }
    }
    catch(int)
    {
        std::cout<<"Exception " << i << std::endl;
    }
}
```

б) Встроенные объекты (все корректно, так как для всех промежуточных стековых объектов будут вызваны деструкторы)

```
class A{
public:
    A(int a=0){m_a = a;}
    ~A() {std::cout<<"~A ";}
};

int main()
{
    A a;
    ...
}
```

6) Модифицируем предыдущий пример: «только» указатели (утечка памяти!!)

```
class C{
    A* m_pA;
    B* m_pB;
public:
    C(int a, int b):m_pA(new A(a)), m_pB(new B(b)) {}
    ~C() {delete[] m_pA; delete[] m_pB; std::cout<<"~C ";}
};

int main()
{
    try{
        C c(1,2);
        catch(int)
        {
            std::cout<<"Exception " << i << std::endl;
        }
    }
    catch(int)
    {
        std::cout<<"Exception " << i << std::endl;
    }
}
```

обратите внимание, что в первом случае деструкторы вызываются для всех стековых объектов, а во втором — для объектов, созданных на момент возникновения исключений.

В) Решение проблемы – все опасное заворачивать в объекты!!!

```
class WrapA{
    A* m_pA;
public:
    WrapA(int n){m_pA = new A[n];}
    ~WrapA(){delete[] m_pA;}
};

class C{
    WrapA wa;//вместо A* m_pA;
    B* m_pB;
public:
    C(int a, int b):wa(a), m_pB(new B(b)) {...}
    ~C(){delete m_pB; std::cout<<"~C";}
};

int main()
{
    try{
        C c(3, 2);
    }
    catch(int)
    {
        //на момент вызова обработчика гарантировано будут
        //вызваны деструкторы для всех
        //созданных на момент
        //возникновения исключения стековых
        //объектов. В нашем случае – только
        //для WrapA => все динамически
        //созданные A будут уничтожены!
    }
}
```

Рекомендация – не стоит возбуждать исключения в деструкторе, разве что тут же их и обработать.

Исключения и наследование

```
class A{
    virtual void f(...);
};

class B:public A{
    virtual void f(...);
};

void f()
{
    throw B(); // производного типа!!!
}

int main()
{
    //тип исключения и наследование
    try{
        f();
    }
    catch(A& a){ //Выработано исключение производного типа B, а
        //принимается базового типа A, а
        //управление все рано будет передано
        //сюда
    }
    ...
}

//на момент вызова обработчика гарантировано будут
//вызваны деструкторы для всех
//созданных на момент
//возникновения исключения стековых
//объектов. В нашем случае – только
//для WrapA => все динамически
//созданные A будут уничтожены!
```

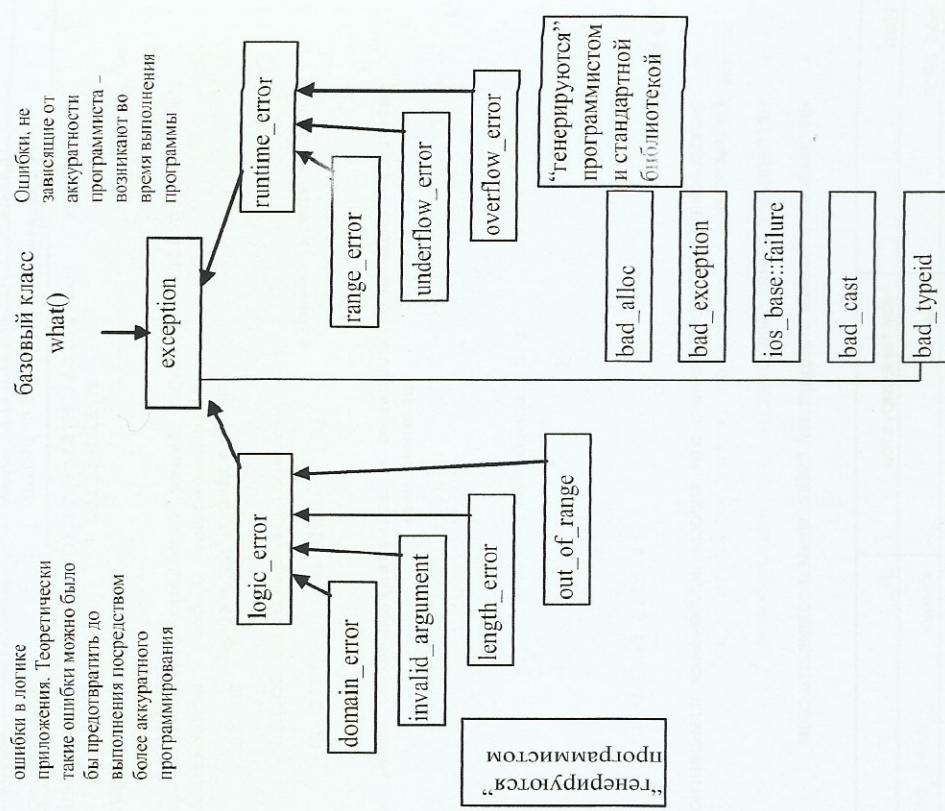
Замечание 1: существенно в случае наследования принимать ссылку, так как по значению Вы получите только копию базовой части объекта => a.f(); - вызов виртуальной функции **базового** класса.

Замечание 2: обработчики исключений просматриваются в том порядке, в котором их указал программист => обработчики производного типа должны предшествовать обработчикам базового типа, иначе последние никогда не будут выполнятся!

Замечание 3: Это справедливо и для указателей!

Стандартные исключения

На самом деле для того, чтобы придать типам исключений некоторую структурность, в стандартной библиотеке определена иерархия классов с базовым классом exception. (все в пространстве имен std). В стандартной библиотеке определены три группы исключений:



```

void F()
{
    try{

```

```
throw exception(); // будет выведено Unknown
// throw exception("MyException"); // будет выведено MyException
```

```
catch (exception& e) // (crash) будет передано управление при возникновении любого исключения, производного от exception

{
    cout<<e.what()<<endl;
}

catch (...) // сюда все остальные – пользовательского типа, не производные
```

```
{ ...
}
```

Вы можете:

- использовать непосредственно эти классы исключений стандартной библиотеки
 - наследовать от них свои классы
 - игнорировать классы стандартной библиотеки и создавать свои
- ```
все остальное <std::except>
```
- ```
operator new
void main()
{
    int N = 0x7fffffff;
    double** p = new double*[N];
    try{
        int i=0;
        for( ; i<N, i++)
        {
            p[i] = new double[N];
        }
    }
    catch(bad_alloc&)
    {
        // i – итерация цикла, на которой память закончилась
        // освобождаем
    }
}
```
- | Имя | Чем генерируется | |
|-------------------|--------------------------------|---------------|
| bad_alloc | new | <new> |
| bad_cast | dynamic_cast | <typeinfo> |
| bad_typeid | typeid | <typeinfo> |
| ios_base::failure | функциями потоков ввода/вывода | <iostream> |
| out_of_range | at() | <std::except> |

```

catch (std::bad_cast b)
{
    ...
}
// не Rect
...
/Rect:
Rect& rRect = static_cast<Rect&>(s);
}

```

Тема XIII. Обобщенное программирование

Идеи обобщенного программирования. Понятия, связанные с шаблонами. Зачем нужны шаблоны.

При создании программ довольно часто приходится писать множество одинаковых фрагментов для обработки **разных** типов данных. Например:

Функция выполняет одни и те же действия, текст на языке высокого уровня будет одинаковым, но низкоуровневый код будет зависеть от типов параметров.

```

T min(T x, T y); // под T можно подразумевать int, double, Point...

```

Параметров компилятор резервирует разное количество памяти при создании объекта типа Point и генерировать разные тела методов класса

Данные класса отличаются только типом, а реализация методов на языке высокого уровня выглядит одинаково:

```

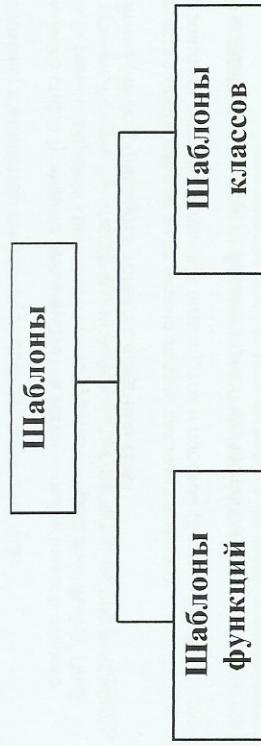
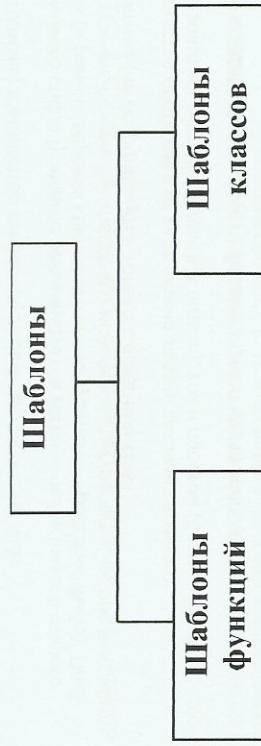
class Point{
    T x, y; // int, double, MyComplex...
}

```

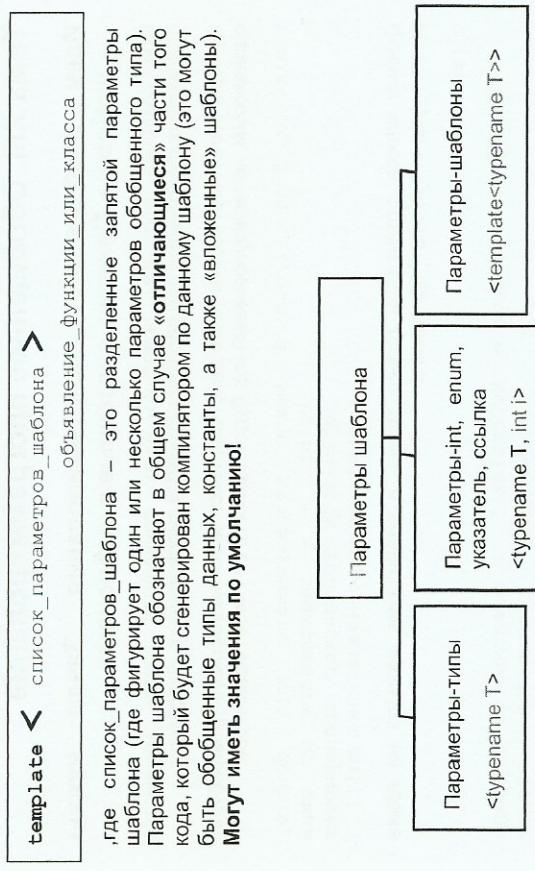
В зависимости от типа параметров компилятор будет резервировать разное количество памяти при создании объекта типа Point и генерировать разные тела методов класса

С помощью ключевого слова **template** на C++ можно задать компилятору образец кода для некоторого обобщенного типа данных - **шаблон**. Используя «скелет» кода, компилятор сам генерирует конечный код для конкретного типа данных.

Шаблоны



Объявление шаблона (общее):



Процесс генерации компилятором как функции (тела), так и объявления класса по шаблону и списку параметров шаблона называется **инстанцирование**. Например, когда компилятор в первый раз встречает вызов функции-шаблона, он создает соответствующий код именно для указанной в качестве параметров шаблона «типов данных» – это неявное инстанцирование. Далее, если компилятор встречает вызов функции с теми же типами параметров, он просто генерирует вызов к уже созданному телу функции. Более того, если даже такие вызовы находятся в разных единицах компиляции, только одна копия тела функции будет включена в исполняемый файл.

Версия шаблона для конкретного набора аргументов называется **специализацией**.

Шаблоны функций

Способы обобщения функций, выполняющих одинаковые действия, но оперирующие различными типами.

Если функция, оперируя различными типами, выполняет одинаковые по смыслу действия, удобно и логично для такой функции иметь одно и то же имя. Такую возможность можно реализовать двумя способами (рассмотрим на примере функции, возвращающей минимальное из двух заданных значений):

- 1) с помощью «перегрузки» функций. При этом программист должен объявить и определить нужное количество функций с одним и тем же именем, которые в нашем примере отличаются только типом параметров:

```

// min for ints
int min( int a, int b ) { return ( a < b ) ? a : b; }

// min for doubles
double min( double a, double b ) { return ( a < b ) ? a : b; }

/etc...
  
```

При этом компилятор генерирует в точке вызова функциями `min()` вызов одной из определенных функций в зависимости от типа параметров.

- 2) без явного (explicit) указания при объявлении шаблона компилятор не создает никакого кода, шаблон является просто заготовкой для компилятора. Только встретив обращение к данному шаблону (вызов функции или создание экземпляра класса и вызов его методов) в тексте программы, компилятор сгенерирует соответствующий код.
- 3) в качестве общего типа можно задать как имя сложного пользовательского типа данных (класс), так и простой (базовый) тип
- 4) можно задать параметры шаблона по умолчанию. Например:

```

template< типарамете A, типарамете B=A > ...
  
```

```

int iX=1000, iY=500;
int iResult = min(iX, iY); // вызов min(int, int)
double dX=1.0, dY = 3;
double dResult = min(dX, dY); // вызов min(double, double)
}

2) с помощью макроподстановки

#define min(a,b) ( a < b ) ? a : b

```

Что гораздо «оласнее», так как макроподстановка – это всего лишь подстановка текста препроцессором

а) типы параметров `a` и `b` могут быть разными, => компилятор может неявно привести типы так, как посчитает нужным, или не сможет привести и выдаст ошибку, но покажет на ошибку в теле макроса (и разобраться в чем дело будет достаточно трудно)

б) некоторых «неожиданностей» при использовании макросов можно избежать с помощью скобок `(a) < (b)` ? `(a) : (b)` – например: без скобок вызов макро `min(x&y&z)` препроцессор превратит в `x&y<b`, а задумано было `(x&y)<b`

в) некоторых побочных эффектов даже при использовании скобок избежать не удается:

```

int r = min(a++, b++); // превращается в (a++) < (b++) ? (a++) :

```

Замечания:

1) как и обычную функцию, шаблон такой короткой функции Вы можете объявить вставляемым:

```

template <class T> inline T min( T a, T b ) { return (a < b) ? a : b; }

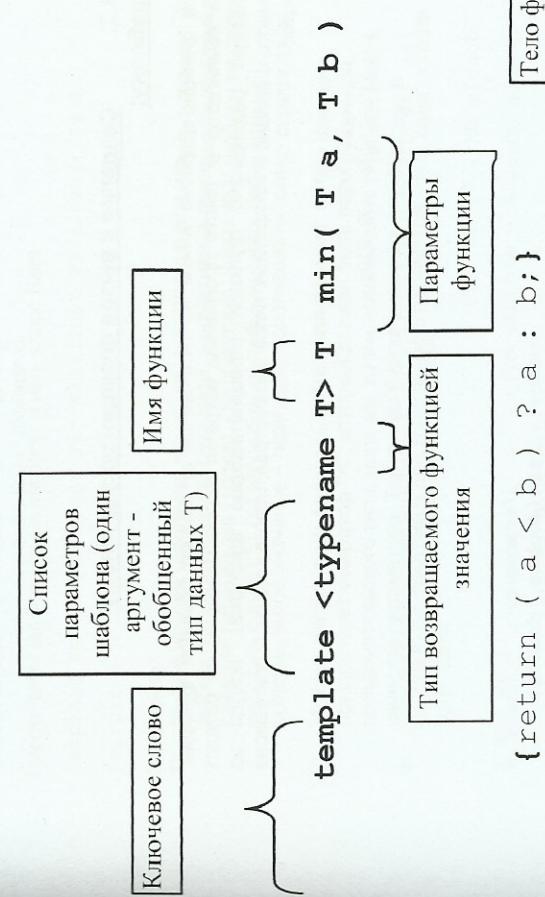
```

2) параметры шаблона (также как и параметры функции) могут иметь значения по умолчанию: `func(T a=T())`

3) так как тело шаблона – это только заготовка компилятору (шаблон), по которой он будет генерировать код для конкретного типа данных, всю заготовку целиком компилятор должен видеть в месте вызова функции => ее тело также должно быть в **заголовковом файле**. Обычно объявление совмещают с определением.

4) функция-шаблон в свою очередь может быть перегружена и в частности наряду с параметрами общего типа может принимать параметры любого типа

3.1. Объявление шаблона функции:



3) с помощью шаблона функции можно

- уменьшить количество «дублируемого» кода, определив только один шаблон, оперирующий с некоторым обобщенным типом данных – «`T`»,
- шаблон лишен недостатков макроподстановок, так как тело функции по шаблону реализуется компилятором, который прекрасно знает семантику C++ (например, параметр шаблона гарантированно вычисляется только один раз).

```
template <class T> T min( T* p, int r, int pim) {...}
    пример - поиск минимального
    элемента в массиве
```

3.2. Создание и вызов функции по заданному шаблону:

Вызов функции-шаблона ничем не отличается от вызова обычной функции. Когда компилятор в тексте программы встретит вызов функции, он создаст конкретную реализацию функции (специализацию шаблона), исходя из заданного шаблона и конкретных типов параметров, использованных при вызове функции:

```
{
    int ix=1000, iy=500;
    int iResult = min(ix, iy); //компилятор создаст и
    //вызовет min(int, int)

    double dx=1.0, dy = 3;
    double dResult = min(dx, dy); //компилятор создаст и
    //вызовет //min(double, double)

//Но!!!
iResult = min(ix, dy); //ошибка компилятора! - параметры
//        разных типов
// => нужно явно указать компилятору какую специализацию нужно
//        вызывать (или генерировать)

iResult = min<int>(ix, dy); //компилятор преобразует
//        <double dx> в <int dx>, и вызовет
min(int, int)
}
```

Таким образом, один раз определив в приведенном примере шаблон функции min(), можно вызывать ее для любых сколь угодно сложных типов данных (для которых определена операция сравнения – <<>). Это существенно снижает объем Вами написанного кода и в то же время повышает его «тибкость» без уменьшения надежности.

Шаблоны функций и объекты пользовательского типа

Рассмотрим шаблон функции min() и наш класс Rect:

```
template <class T> const T& min(const T& a, const T& b )
{
    return ( a < b ) ? a : b;
}
```

Замечание:

- 1) **Модификация шаблона**: так как мы распространяли использование шаблона на пользовательские типы, эффективное передавать **ссылки** в качестве параметров (+const – необходимо, иначе этот шаблон не сможет работать с min(1,5) - a) запретить модифицировать параметры, б) возвращаемое значение – позволить использовать только справа от "=")
- 2) **Модификация пользовательского типа данных**: в теле функции присутствует **оператор <<**, который должен быть перегружен для пользователя типа, если мы хотим использовать объекты Rect в качестве параметров. Так как мы будем вызывать эту функцию в константной функции и для константного параметра – она сама должна быть const.

```
class Rect{
    int m_l, m_t, m_r, m_b;
public:
    Rect( int l, int t, int r, int b ) {m_l=l; m_t=t; m_r=r; m_b=b; }

    bool operator<(const Rect& r) const
    {
        return
            Square() <r.Square();
    }

    double Square() const { return (m_r-m_l)*(m_b-m_t); }

};

int main()
{
    Rect r1(1,1,3,3), r2(0,0,10,10);
    Rect res = min(r1,r2);
}
```

Шаблоны классов.

Шаблоны классов называют также «обобщенными классами» (generic classes). Шаблон класса описывает, как компилятору сгенерировать класс (то есть **выделить память под данные и сгенерировать код методов класса**) по соответствующему набору аргументов шаблона и «скелету» класса. (Объявление шаблона является заготовкой класса, по которой компилятор создает конкретные классы, основываясь на конкретных типах используемых данных.)

Специфика:

1) шаблоны могут участвовать в наследовании

2) шаблоны методов могут быть виртуальными

```
template<typename T> class A{
public:
    T m_a;
    A(const T& a) {m_a = a;}
    virtual void f() {m_a++;}
};
```

```
template<typename T> class B:public A<T>{
public:
    T m_b;
    B(const T& a, const T& b):A<T>(a) {m_b = b;}
    virtual void f() {m_b++;}
};
```

```
public:
```

```
    B<int>* pB = new B<int>(1, 5);
    A<int>* pA = new B<int>(1, 5);
    pA->f();
    pB->f();
}
```

- 3) шаблоны классов могут содержать статические члены

```
MyVector.h
template <typename T> class MyVector {
public:
    static int m_count;
};
```

main.cpp

```
int MyVector<int>::m_count=0;
int main()
{
    int n = MyVector<int>::m_count;
```

4) у шаблонов могут быть friend-функции и классы

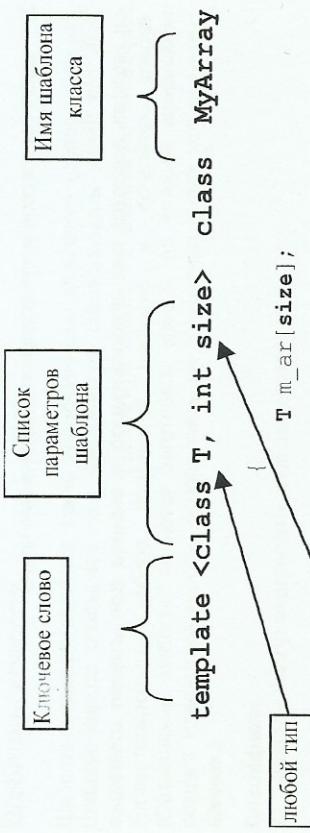
5) классы могут содержать встроенные шаблоны, при этом действуют некоторые ограничения, которые перечислены по мере использования

Для примера рассмотрим шаблон класса, с помощью которого можно реализовать «массив» заданного размера (size) для элементов любого типа (обобщенный тип T):

```
template<typename T> class A<T>{
public:
    T m_a;
    A(T a):m_a(a){}
    void f() {m_a++;}
};
```

```
public:
```

```
    B<int>* pB = new B<int>(1, 5);
    A<int>* pA = new B<int>(1, 5);
    pA->f();
    pB->f();
}
```



Методы шаблона класса вне объявления класса определяются следующим образом:

```
template <class T, int size> T& MyArray< T, size >::operator[](int i)
{
    if( i >= 0 && i < size) return m_ar[i];
    // исключение - out_of_range
}
```

Создание объектов конкретного типа на базе шаблона:

```
int main()
{
    MyArray< int, 5 > ar1; // создает массив для 5 элементов
    int ar1[1] = 1;
    int iTmp = ar1[1];
}
```

```
MyArray< char, 6 > ar2; // создает массив для 6 элементов
char
MyArray< Rect, 7 > ar3; // создает массив для 7 элементов
типа Rect
...
}
```

1) Замечания:

- 1) так как шаблоны являются механизмом времени компиляции, все параметры списка параметров базовых типов шаблона (такие как "int") должны быть константами:
- 2) методы класса являются шаблонами функций => должны быть реализованы тоже в заголовочном файле
- 3) параметры шаблона базового типа могут иметь значения по умолчанию:

```
int N = 5;
MyArray< MyClass, N > ar4; // ошибка компилятора (N не является константой!)
2) методы класса являются шаблонами функций => должны быть реализованы тоже в заголовочном файле

template<typename T, int size> class MyArray{...};

MyArray<int> ar5; // размер массива по умолчанию == 10
3) параметры шаблона базового типа могут иметь значения по умолчанию:
```

- 4) шаблон класса может содержать метод класса, который в свою очередь является шаблоном, базирующимся на другом типе. При этом такой метод-шаблон должен быть встроенным – то есть должен быть «определен» внутри класса.

```
template<typename T> class X
{
public:
    template<typename U> void f(const U &u) { ... } //OK
};
```

Некорректно:

```
template<typename T> class X {  
public:  
    template<typename U> void f(const U &u);  
};  
  
template<typename T> template <typename U>  
void X<T>::f (const U &u)  
{  
}
```