

# Статические члены класса.

- Статические члены-данные и члены-функции описываются в классе с квалификатором **static**.
- Статические члены-данные существуют в одном экземпляре и доступны для всех объектов данного класса.
- Статические члены класса существуют независимо от конкретных экземпляров класса, поэтому обращаться к ним можно еще до размещения в памяти первого объекта этого класса.
- Необходимо предусмотреть выделение памяти под каждый статический член-данные класса (т.е. описать его вне класса с возможной инициализацией), т.к. при описании самого класса или его экземпляров память под статические члены-данные не выделяется.
- Доступ к статическим членам класса (наряду с обычным способом) можно осуществлять через имя класса (без указания имени соответствующего экземпляра) и оператор разрешения области видимости «::».

## Пример.

```
class A {  
public:  
    static int x;  
    static void f (char c);  
};
```

int A::x; // **внимание!** – размещение статического объекта в  
памяти

```
void g() {  
    ...  
    A::x = 10;  
    ...  
    A::f ('a');  
    ...  
}
```

## Особенности использования статических методов класса

- Статические методы класса используются, в основном, для работы с глобальными объектами или статическими полями данных соответствующего класса.
- Статические методы класса не могут пользоваться нестатическими членами-данными класса.
- Статические методы класса не могут пользоваться указателем `this`, т.е. использовать объект, от имени которого происходит обращение к функции.
- Статические методы класса не могут быть виртуальными и константными (`inline` - могут).

# Средства обработки ошибок. Исключения в С++

Обработка **исключительных ситуаций** в С++ организуется с помощью ключевых слов

**try, catch и throw.**

Операторы программы, при выполнении которых необходимо обеспечить обработку исключений, выделяются в **try-catch** - блок.

Если ошибка произошла внутри **try**-блока (в частности, в вызываемых из **try**-блока функциях), то соответствующее **исключение** должно генерироваться с помощью оператора **throw**, а перехватываться и обрабатываться в теле одного из обработчиков **catch**, которые располагаются непосредственно за **try**-блоком.

**Исключение** - объект некоторого типа, в частности, встроенного.

Операторы, находящиеся после места генерации ошибки в **try**-блоке, игнорируются, а после обработки исключения управление передается первому оператору, находящемуся за обработчиками исключений. **try-catch**-блоки могут быть вложенными.

Общий синтаксис **try-catch** блока:

```
try {  
..... throw исключение; .....  
}  
catch (type) {---/*throw;*/}  
catch (type arg) {---/*throw;*/}  
...  
catch (...) {---/*throw;*/}
```

## Перехват исключений.

С каждым try-блоком может быть связано несколько операторов **catch**. Они просматриваются по очереди сверху вниз.

Какой именно обработчик **catch** будет использоваться, зависит от типа сгенерированного исключения.

Выбирается первый обработчик с типом параметра, совпадающим с типом исключения. Ловушки с базовым типом (или с указателем или ссылкой на базовый тип) перехватывают все исключения с производным типом (или его адресом), т.е. производные типы должны стоять раньше базовых типов.

Если исключение перехвачено каким-либо обработчиком **catch**, аргумент **arg** получает его значение, которое затем можно использовать в теле обработчика. Если доступ к самому исключению не нужен, то в операторе **catch** можно указывать только его тип.

Существует специальный вид обработчика, перехватывающего любые исключения - **catch (...){---}**. Естественно, он должен находиться в конце последовательности операторов **catch**.

Если для сгенерированного исключения в текущем **try**-блоке нет подходящего обработчика, оно перехватывается объемлющим try-блоком (**main()**→**f()**→**g()**→**h()**).

Если же подходящего обработчика так и не удалось найти, может произойти **ненормальное** завершение программы. При этом вызывается стандартная библиотечная функция **terminate ()**, которая в свою очередь вызывает функцию **abort ()**, чего лучше избегать.

## Перехват исключений.

```
void f ()
{
    try {
        throw E();
    }
    catch (H) {
        // когда мы сюда попадем?
    }
}
```

Обработчик будет вызван:

- (1) Если H того же типа, что и E.
- (2) Если H является однозначной открытой базой E.
- (3) Если H и E являются указателями, и (1) или (2) выполняется для типов, на которые они указывают.
- (4) Если H является ссылкой, и (1) или (2) выполняется для типа, на который H ссылается.

## Пример.

```
class A {
public:
    A () {cout << "Constructor of A\n";}
    ~A () {cout << "Destructor of A\n";}
};
class Error {};
class Error_of_A : public Error {};
void f () {
    A a;
    throw 1;
    cout << "This message is never printed" << endl;
}
int main () {
    try {
        f ();
        throw Error_of_A();
    }
    catch (int) { cerr << "Catch of int\n"; }
    catch (Error_of_A) { cerr << "Catch of Error_of_A \n"; }
    catch (Error) { cerr << "Catch of Error\n"; }
    return 0;
}
```

Результат работы программы на предыдущем слайде.

*Constructor of A*

*Destructor of A* // т.к. в f обработчика нет, поиск идет дальше,  
// но при выходе из f вызывается деструктор  
// локальных объектов.

*Catch of int*

Если поменять строки внутри try, получим:

*Catch of Error\_of\_A*

Если закомментировать строку

```
// catch (Error_of_A) { cerr << "Catch of Error_of_A \n"; },
```

получим

*Catch of Error*

## Пример использования классов исключений.

```
class MathEr    {...virtual void ErrProcess();...};  
class Overflow : public Math Er    {... void ErrProcess();...};  
class ZeroDivide : public Math Er  {... void ErrProcess();...};  
...
```

Через параметры конструктора исключения можно передавать любую нужную информацию.

Если использовать виртуальные функции, можно после **try**-блока задать единственный обработчик **catch**, имеющий параметр типа базового класса, но перехватывающий и обрабатывающий любые исключения:

```
try {    ...  
}  
catch (MathEr & m) {... m. ErrProcess(); ...}
```

Организованная таким образом обработка исключений позволяет легко модифицировать программы.

# Исключения, генерируемые в функциях.

В заголовке функции можно указать типы исключений (через запятую), которые может генерировать функция (эту возможность удобно использовать при описании библиотечных функций):

```
тип_рез имя_функции (список_арг) [const] throw (список_типов) { ... }
```

Если список типов **пустой**, то функция не может генерировать **никаких** исключений.

Если же функция все-таки сгенерировала недеklarированное исключение, вызывается библиотечная функция ***unexpected()*** работающая аналогично функции ***terminate()***.

Использование аппарата исключений – единственный безопасный способ нейтрализовать ошибки в конструкторах и деструкторах, поскольку они не возвращают никакого значения, и нет другой возможности отследить результат их работы.

Если деструктор, вызванный во время свертки стека, попытается завершить свою работу при помощи исключения, то система вызовет функцию ***terminate()***, что крайне нежелательно. Отсюда важное требование к деструктору: ни одно из исключений, которое могло бы появиться в процессе работы деструктора, не должно покинуть его пределы.

## Действия, выполняемые с момента генерации исключения до завершения его обработки.

- При генерации исключения (*throw X*) создается временный объект – копия X (работает конструктор копирования). С этой копией будет работать выбранный далее обработчик; она существует до тех пор, пока обработка исключения не будет завершена.
- Для всех других объектов *try*-блока, созданных к этому моменту, перед выходом из *try*-блока освобождается память; при этом для объектов – экземпляров классов вызывается деструктор. То же делается и для уже созданных подобъектов: членов класса – объектов другого класса и баз. Этот процесс называют «раскруткой» («сверткой») стека.
- Если в списке обработчиков *catch* этого *try*-блока найден подходящий, то выполняются его операторы; затем выполнение программы продолжается с оператора, расположенного за последним обработчиком этого *try*-блока.
- Если в списке данного *try*-блока не нашлось подходящего обработчика, то поиск продолжается в динамически объемлющих *try*-блоках (при этом процесс свертки стека продолжается).
- Если подходящего обработчика так и не нашлось, то вызывается функция *terminate()* и выполнение программы прекращается.

# Механизм RTTI (Run-Time Type Identification).

Механизм RTTI состоит из трех частей:

1. операция **dynamic\_cast**  
(в основном предназначена для получения указателя на объект производного класса при наличии указателя на объект базового класса);
2. операция **typeid**  
(служит для идентификации точного типа объекта при наличии указателя на базовый класс);
3. структура **type\_info**  
( позволяет получить дополнительную информацию, ассоциированную с типом).

Для использования RTTI в программу следует включить заголовок `<typeinfo>`.

(1). Операция ***dynamic\_cast*** реализует приведение типов (указателей или ссылок) полиморфных классов в динамическом режиме.

(операции ***const\_cast***, ***reinterpret\_cast*** и ***static\_cast*** здесь не рассматриваются).

Синтаксис использования операции ***dynamic\_cast*** :

***dynamic\_cast*** < целевой тип > ( выражение )

Если даны **два полиморфных класса В и D** (причем D – производный от B), то ***dynamic\_cast*** всегда может привести D\* к B\*.

Также ***dynamic\_cast*** может привести B\* к D\*, но только в том случае, если объект, на который указывает указатель, действительно является объектом типа D (либо производным от него)!

При неудачной попытке приведения типов результатом выполнения ***dynamic\_cast*** является 0, если в операции использовались указатели. Если же использовались ссылки, генерируется исключение типа ***bad\_cast***.

**Пример:**

```
Base *bp, b_ob;
```

```
Derived *dp, d_ob;
```

```
bp = &d_ob;
```

```
dp = dynamic_cast <Derived *> (bp);
```

```
if (dp)
```

```
    cout << «Приведение типов прошло успешно»;
```

```
bp = &b_ob;
```

```
dp = dynamic_cast <Derived *> (bp);
```

```
if (!dp)
```

```
    cout << «Приведения типов не произошло»;
```

(2)-(3) Информацию о типе объекта можно получить с помощью операции ***typeid***.

Синтаксис использования операции *typeid*:

***typeid* (выражение)**      или  
***typeid* (имя\_типа)**

Операция *typeid* возвращает ссылку на **объект класса *type\_info***, представляющий либо тип объекта, обозначенного заданным выражением, либо непосредственно заданный тип.

В классе *type\_info* определены следующие открытые члены:

```
bool operator == (const type_info & объект); // для сравнения типов  
bool operator != (const type_info & объект); // для сравнения типов  
bool before (const type_info & объект); // для внутреннего использования  
const char * name ( ); // возвращает указатель на имя типа
```

Оператор *typeid* наиболее полезен, если в качестве аргумента задать указатель полиморфного базового класса, т.к. с его помощью во время выполнения программы можно определить тип реального объекта, на который он указывает. То же относится и к ссылкам.

*typeid* может применяться к нулевым указателям (*typeid (\* p)*). Если указатель на полиморфный класс `p == 0`, то будет сгенерировано исключение типа ***bad\_typeid***.

## Пример.

```
class Base {
    virtual void f ( ) {...};
};
class Derived1: public Base { ...
};
class Derived2: public Base { ...
};
int main ( ) {
    int i;
    Base *p, b_ob;
    Derived1 ob1;
    Derived2 ob2;
    cout << «Тип i - » << typeid (i).name ( ) << endl;
    p = &b_ob;
    cout << “p указывает на объект типа ” << typeid (*p).name ( ) << endl;
    p = &ob1;
    cout << “p указывает на объект типа ” << typeid (*p).name ( ) << endl;
    p = &ob2;
    cout << “p указывает на объект типа ” << typeid (*p).name ( ) << endl;
    if ( typeid (ob1) == typeid (ob2) )
        cout << “Тип объектов ob1 и ob2 одинаков\n”;
    else
        cout << “Тип объектов ob1 и ob2 не одинаков\n”;
    return 0;
}
```

# Шаблоны

1. Механизм шаблонов реализует в С++ **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.



# 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;

    int i, j = 8, k = 10;

    c = f ( a , b );    // f < complex > ( a , b )

    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 );
    k = max < int > (i, j);
    // z = max (x, i);
    z = max < double > ( y, j );
    s3 = max (s2, s1);
```

```
    return 0;
```

```
}
```

```
// max <double>
```

```
// max <int>
```

```
// ошибка! - неоднозначный выбор параметров
```

```
// max < char * > ,
```

```
// но происходит сравнение адресов
```

### 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;
}
```

# 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 **Точного** 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 шаблоны 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";
}
```