Множественное наследование

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

Спецификатор доступа распространяется только на один базовый класс; для других базовых классов начинает действовать принцип умолчания.

Класс не может появляться как непосредственно базовый дважды:

class C : public A, public A { ... }; - ошибка!

но может быть более одного раза непрямым базовым классом:

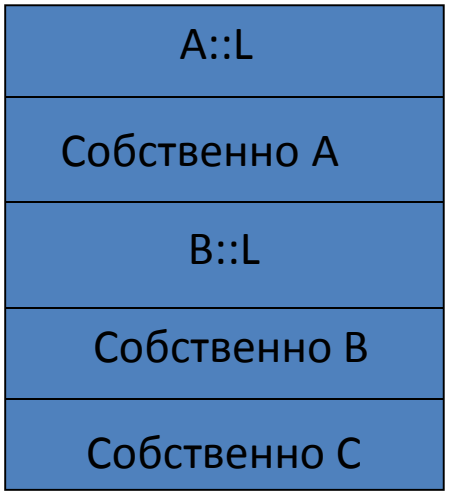
class L { public: int n; ... };

class A : public L { ... };

class B : public L { ... };

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

Здесь **решетка смежности** такая: L <-- A <-- C --> B --> L .



При этом может возникнуть неоднозначность из-за «многократного» базового класса.

О доступе к членам производного класса

void C::f () { ... n = 5; ...} // ошибка! – неясно, чье n, но

void C::f () { ...A::n = 5; ...} O.K.! , либо B::n = 5;

Имя класса в операции разрешения видимости (A или B) – это указание, в каком классе в решетке смежности искать заданное имя.

О преобразовании указателей

Указатель на объект производного класса может быть неявно преобразован к указателю на объект базового класса, только если этот базовый класс является **однозначным** и **доступным**

Продолжение предыдущего примера:

```
void g () {
        C* pc = new C;
        L* pl = pc;          // ошибка! – L  не является однозначным,
        pl = (L*) pc;        // ошибка! – явное преобразование не помогает,
                                    // но возможно:
        pl = (L*)(A*) pc;    // либо pl = (L*)(B*) pc;   -- O.K.!
```

Базовый класс считается **доступным** в некоторой области видимости, если доступны его public-члены.

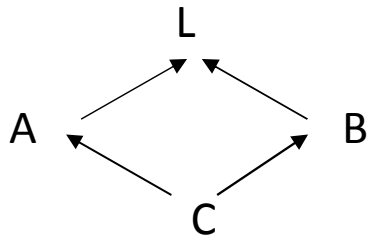
```
class B   { public: int a; ... };
class D : private B { ... };

void g () {
        D* pd = new D;
        B* pb = pd;          // ошибка! – в g() public-члены B, унаследованные
                                    // D,  недоступны, такое преобразование
                             // может осуществлять только
                             // функция-член D, либо друзья D.
}
```

Виртуальные базовые классы

```
class L   { public:  int  n ;  ... };
class A :  virtual public  L  { ... };
class B :  virtual public  L  { ... };
class C : public A, public B  { ... void f (); ... };
```

Теперь решетка смежности будет такой:



и теперь допустимо:

```
void C :: f () { ... n = 5; ...} // О.К.! – n в одном экземпляре

void g () {
        C* pc = new C;
        L* pl = pc;             // О.К.! – появилась однозначность.
}
```

Неоднозначность из-за совпадающих имен в различных базовых классах.

```
class A {
   public:
        int a;
        void (*b) ( );
        void f ( );
        void g ( );  ...
};
```

```
class B {
        int a;
        void b ( );
void h (char);
public:
        void f ( );
        int g;
        void h ( );
        void h (int);  ...
};
```

```
class C : public A, public B  { ... }; //какие  имена
                                       //неоднозначны ?
```

Правила выбора имен в производном классе

1 шаг: контроль **однозначности** (т.е. проверяется, определено ли анализируемое имя в одном базовом классе или в нескольких); при этом контекст не привлекается, совместное использование (в одном из базовых классов) допускается.

2 шаг: если однозначно определенное имя есть имя перегруженной функции, то пытаются **разрешить** анализируемый вызов (т.е. найти best-maching).

3 шаг: если предыдущие шаги завершились успешно, то проводится контроль **доступа**.

Пример.

```
class A {     public: int a; void (*b) ( );
              void f ( );    void g ( );  ...
};

class B { int a; void b ( ); void h (char);
public:       void f ( ); int g; void h ( );
     void h (int);  ... };

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

void gg (C* pc) {

              pc --> a = 1;              // ошибка! – A::a  или  B::a для однозначности
              pc --> b();   // ошибка! – нет однозначности
              pc --> f ();   // ошибка! – нет однозначности
              pc --> g ();  // ошибка! – нет однозначности,
                                         // контекст не привлекается!
              pc --> g = 1;              // ошибка! – нет однозначности,
                                         // контекст не привлекается!
              pc --> h ();  // O.K.!
              pc --> h (1);              // O.K.!
              pc --> h  ('a');        // ошибка! – доступ в последнюю очередь, не доступно
              pc --> A::a = 1;           // O.K.! – т.е. снимаем неоднозначность
                                                  // с помощью  операции «::»
            pc --> B::a = 1;             // ошибка! – поле a не доступно в B (private)
}
```

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 B 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;
}
```

Константные методы

Если необходимо запретить методу изменять информационные члены объектов класса, то при его описании используется дополнительный модификатор ***const*:**

<тип возвр. значения> <имя функции> **(** <пар-ры> **)** ***const*** { <тело> }

Описанные таким образом методы класса называются **константными.**

- Если объект объявлен с модификатором **const**, то изменение его состояния недопустимо. В таком случае все применяемые к этому объекту методы (кроме конструкторов и деструктора) **должны иметь** модификатор ***const*.**

- Данное требование является обязательным **независимо от наличия или отсутствия** информационных членов в классе.

- Для защиты от изменения передаваемых фактических параметров в теле функции соответствующие формальные параметры также объявляются с модификатором ***const***:

  const <тип параметра> <идентификатор>

Расширим наш класс string следующими методами:

```
class string {
    //...
public:
    //...
    string &  concat ( const string & s ) // конкатенация с другой строкой
    int  length () const {return size;} // возвращает длину строки
};
```

Объявив метод length константным, мы явно разрешаем его вызов для константных объектов типа string. Поэтому реализация метода concat (с константным формальным параметром), использующего функцию length, станет допустимой.

Примеры

```
void f (const int i, const myclass ob) {
 i = 1; // ошибка!
ob.f (); // ошибка!, если  f() неконстантный метод
}
```

```
void f (const int * i, const myclass & ob) {
i = NULL; // O.K.
*i = 3; // ошибка!
ob.f(); // ошибка!, если  f() неконстантный метод
 }
```

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-данное 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 происходи обращение к функции.
- Статические методы класса не могут быть виртуальными и константными (inline - могут).

Средства обработки ошибок. Исключения в C++

Обработка **исключительных ситуаций** в C++ 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()$\rightarrow$f()$\rightarrow$g()$\rightarrow$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 недекларированное 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.

Шаблоны

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

Шаблоны функций.

```
template < список_параметров_шаблона >
тип_рез-та   имя_функции ( список_аргументов_функции ) { /*...*/ }
```

Обращение к функции-шаблону: имя_функции < список_фактич._параметров_шаблона >
(список_фактич_аргументов_функции);

Пример: функция суммирования элементов массива.

```
template < class T >
T  sum ( T  array[ ],  int size ) {
        T  res = 0;
        for ( int i = 0; i < size; i++ )    res += array[ i ];
        return res;
}
```

Использование данного шаблона для массивов типа **int** [10] :

```
int  iarray [10];
int  i_sum;
//...
i_sum = sum < int > ( iarray, 10 );
```

Можно задать аргумент **size** в виде параметра шаблона:

```
template < class  T, int  size >
T  sum ( T  array [ ] ) { /* ... */ }
```

Тогда вызов sum будет таким: i_sum = sum < **int**, 10 > (iarray);

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 );                    // max <double>
      k = max < int > (i, j);              // max <int>
      // z = max (x, i);                   // ошибка! -  неоднозначный выбор параметров
      z = max < double > ( y, j );
      s3 = max  (s2, s1);                  // max < char * >,
                                           // но происходит сравнение адресов
      return 0;
}
```

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;
}
```

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 <int, int> (2, 3);           // m1_3
	m1 <int> (2, 3, i);             // m1_2
	m1 (1, 1);                      // m1_4
	m1 (1.3, 1);                    // m1_3
	m1 (1.3, 1.3);                  // m1_1
	return 0;
}
```

// Если убрать первый шаблон:

```
// m1_3
// m1_3
// m1_2
// m1_4
// m1_3
// m1_3
```

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";
}
```