```
void f()
{   A a
    try {
        B b;
        g();
        h()
    }
    catch(UserExcep &e)
    {  cout<<e.what();
       return;
    }
}
```

Handler → whenever handler terminate it exception is not rethrown it will destroy that Object

Ex.2

```
class myException: public exception { };
class myclass { };
void h() { myclass a;
    // throw 1;
    // throw 2.5;
    // throw myException();
    // throw exception();
    // throw myclass;
} a.~myclass() // for normal execution destructor will be called here
void g()
{   myclass a;
    try {
        h();
    }
```

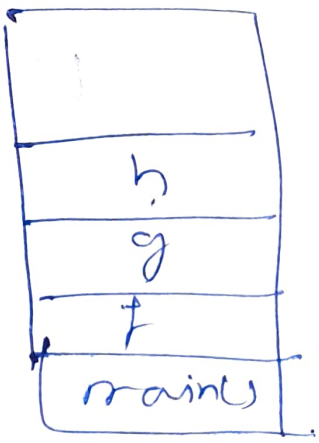
g()
n=5
if not handle in try block
→ it will immediately raise and rethrow the exception.

```
catch(int) { cout<<"int"<<endl; }
catch(double) { cout<<"double"<<endl; }
catch(...) { throw; }
```

ellipsis or catch all clause obj → destructor will be called for local obj

→ re throw the object to caller

```
}
void f()
{ myclass a;
  try
  { g(); }
  catch(MyException) { cout<<"MyException"<<endl; }
  catch(exception) { cout<<"exception"<<endl; }
  catch(...){throw;}
}
int main()
{  try
   { f(); }
   catch(...) { cout<<"Unknown"; }
}
```



// not catch clause from main can not throw it's not case termination handler will be called.

(28) catch clause do not oraten by implicit conversion.

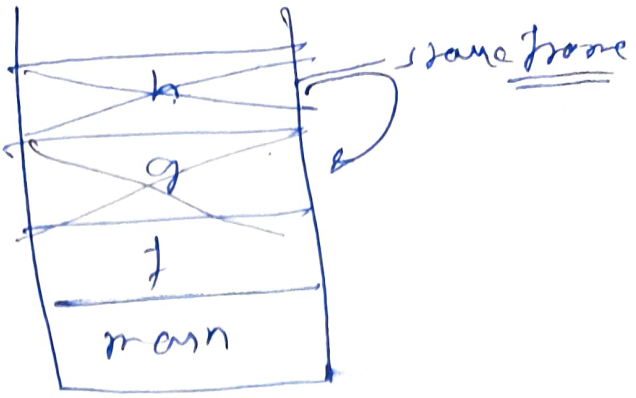
(i) Overload resoln allows implicit conversion. This is not allowed in catch clause

(ii) Overload resolution happens over the set of fn as a whole But in catch clause it happens according to order.

* This not necessary that the caller will handle the exception. caller might not it will rethrow or propagate to callee to caller

* Default behaviour of exception mechanism is if the caller has received the exception which it is not being able to handle bcoz there was no catch clause it by default rethrow it (except for main since there is no caller it will call abort)

```
catch (A a) { // handle ...
            throw }
```



* Destructor of all Object which are live at throw point stmt, their destructor will get called and only after that the stack frame will be removed.

Stack unwinding → destroying the local environment objects and removing the stack frame as we throw object from one fn to its parent.

try block: Exception Scope.

* try block
  * code area that might throw exception

* function try block
  * Area for detection is the entire fn body.

```
void fn()
try {
    throw E();
}
catch(E &e) {
}
```

Nested try block

* semantically equivalent to nested fn calls

```
try {
    try { throw E1(); }
    catch (E2 e) { }
}
catch (E1 e) {
}
```

Catch block / Exception argument

Catch block

(i) Name for exception handler

* Catching an exception is like an invoking a fn

(ii) Immediately follow the try block

(iv) Unique formal parameter to each Handler

(v) Can be simply a type name to distinguish its Handler from others

How should we pass argument to catch block

Usually as reference

* We

* Exception object are special kind of argument object which lie in between dynamic or runtime but not on compile time.

(27) and it is always be executed on free store Heap.

try-catch: Exception matching

* Exact match

(1) The catch argument type matches the type of the thrown object.
* No implicit conversion is allowed

Generalization / specialization

* The catch argument is a public base class of the thrown class object

pointer

pointer types - convertible by standard conversion

we should first write catch clause for child class and then for Base class

catch (B *)
catch (A *

try catch: exception matching

1. In the order of appearance with the matching
   - If Base class catch block precedes derived class catch block then Compiler issues a warning and continues

(273) (II) Unreachable code (derived class) ignored

* catch(...) block must be the last catch block bcoz it catches all exception

* If no matching handler is found in current scope, the search continues to find a matching handler in a dynamically surrounding try block

(1) Stack unwinds

* If eventually no handlers found, terminate is called

throw Expression: Exception raise

(I) Expression is treated three ways as

* A function argument in a call or the operand of a return stmt.

(II) Exception context.

* class Exception;

(III) The Expression

* constructs an exception object by

throw

eg: throw Exception;

* or copies an existing exception object to throw

- Exception ex
- throw ex // Exception(ex)

✓ Exception Object is created on the Free Store

throw Expression: Restriction

* For an UDT expression
  Copy constructor and destructor should be supported

* The type of expression cannot be

(i) An incomplete type like void
array of unknown size or of element of incomplete type
Declared but not defined (struct/union/enum/class Objects or pointers to such objects

(ii) A pointer to an incomplete type
except void*, const void*, volatile void*, const volatile void*.

Rethrow

- Catch may pass on the exception after Handling.
- rethrow is not same as throwing again.

Throwing again

```
try{ ... }
catch( Exception & ex)
{ //Handle and
  //Raise again
  throw ex
}
// ex copied
// ex destroyed
```

Rethrow

```
try{ ... }
catch(Exception & ex) {
  //Handle and
  ...
  pass on
  throw;
}
// No copy
// No destruction
```

(276) Advantage

(I) destructor savvy

(II) Unobtrusive

(III) precise

(IV) Native & standard

(V) scalable

(VI) Fault tolerant

(272) generic programming concept or meta programming concept

* The core idea of template is, we would be able to write a code which at the time of compilation can generate further new code and that generated code then gets compiled again

*

What is a template.

Templates are specifications of a collections of fn or classes which are parameterized by type

Ans

(1) Function search, min, max etc.

* The basic algo in these fn are these same independent of types

* Yet we need to write diff version of these fn for strong type checking in C++

(11) Classes list, queue etc.

* The data members and the methods are almost the same for list of no or list of objects

(28) * We want to increase code reusability in C++

Function template: code reuse in Algo

max

(I) int

(II) double

(III) char*(C-string)

(IV) Complex (user defined class for complex no)

Overload max fn as

(I) int max(int a, int y)

(II) double max(double x, double y)

(III) char * (char *x, char *y)

(IV) complex max (complex x, complex y)

Issue in max

(I) same algo

(II) Different code version

Ex.

int max(int x, int y) { return x>y ? x : y; }

double max(double x, double y)

{ return x>y ? x : y; }

```
char *max( char *x, char *y)
{
    return strcmp(x,y) > 0 ? x : y;
}
```

```
main()
    int a = 3, b = 5
    double c = 2.1, d = 3.7
    max(a,b)
    max(2.1, 3.7)
```

```
char *s1 = new char[6], *s2 = new char[6];
strcpy(s1, "black"); strcpy(s2, "white");
```

above can be solved using macro but macro has its own pitfalls

other ways

Function template:

(i) describe how a function should be built

(ii) supplies a definition of the fn using some arbitary type (as place holders)
* a parameterized definition (type as parameters)

(iii) can be considered the definition for a set of overloaded versions of a fn

(14) Every template parameter is a built-in type

(280) or class type parameter

```
template < class T > or
template < typename T >
```

Ex:

```
template < class T > T max( T x, T y){
                  ↑ placeholder for type
    return x > y ? x : y; }
```

```
int a = 3, b = 5
double c = 2.1, d = 3.7
int res1 = max<int>(a, b)
                    ↑ type
```

instantiated name

Where code of max or data type T as int or generate the code of max taking T as int and then we you call or to max fn

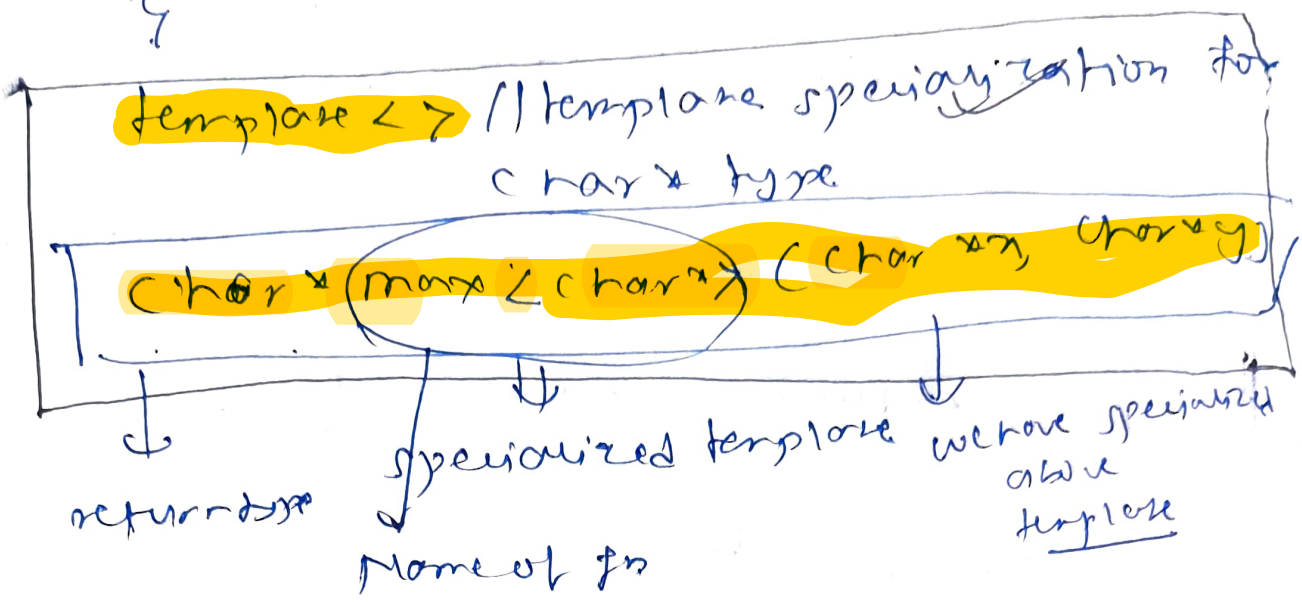
```
double res2 = max<double>( c, d)
```

template specialization → first specialized template get preference

* If we compare the two pointers it will compare the address.

When we have a scenario where our template is working for 2 or 3 data type but not for 3rd type then we want to do

```
template <class T> T max(T x, T y)
{
    return x>y? x:y;
}
```




```
{ return strcmp(x,y)>0 ? x : y; }

char *s1 = new char[6];
char *s2 = new char[6];
strcpy(s1, "black");
strcpy(s2, "white");

max<char*>(s1,s2)
```

Implicit instantiation Here type of parameter will tell us what is the value of T.

```
template <class T> T max(T x, T y)
{
    return x>y? x:y;
}
```

(202) int a=3, b=5

double c=2.1, d=3.7

```
int res1 = max(a, b)
```

int int

both are int then OK.

Implicit instantiation

```
double res2 = max(c, d)
```

double double

Both are double and consistent

```
max(a, c)
```

// Not work

int double

but

```
max<double>(a, c)
```

will

* work as it is explicit instantiation which will convert a into double

Template Argument Deduction: Implicit instantiation

(i) each item in the template parameter list is a template argument.

(ii) When a template fn is invoked, the value of the template arguments are determined by seeing the type of fn arguments

template<class T> T max(T a, T b);

~~template<>~~ char* max<char*> (char*, char*)

template<> char* max<char*> (char* x, char* y)

```
template<class T, int size> Type
max(T x[size]);
```

int a, b, max(a,b) // binds to max<int> (int, int)

double c, d max(c, d)

↓

binds to max<double> (double, double)

char *s1, *s2 max(s1, s2)

↓

binds to max<char*> (char*, char*)

```
int pval[20]; max(pval) // error
```

Three kinds of conversion are allowed in template:

(i) L-value transformation (array to pointer conversion)

(ii) Qualification conversion (const to non const)

(iii) Conversion to a base class instantiation from a class template.

Ex-1 For user defined type.

```
class complex { double re, im;
public:
    complex( double a=0, double b=0)
    : re(a), im(b) { };
    friend bool operator > ( const
    complex &c1, const complex &c2)
    { return c1.norm() > c2.norm();
    }
    friend ostream & operator << (
     ostream &os, const complex &c)
    {
       os << c.re << c.im;
    }
};
template<class T> T max(T x, T y)
{ return x>y? x: y;
}
template<> char * max<char *> (
   char * x, char *y)
{ return strcmp(x,y)>0? x: y;
}
```

Ex complex c1(2.1, 3.2) c2.(6.2, 7.2)

cout << max(c1, c2) << endl;

template overload. → we can put typname in place of class

```
template <class T> T max(T x, T y)
{ return x>y? x: y; }

template <> char* max<char*>(
  char* x, char* y)
{ return strcmp(x,y)>0 ? x : y; }
```

non type parameter

```
template <class T, int size> T
  max(T x[size])
{ T t = x[0];
  for(int i=1; i<size; i++)
  { if(t<x[i]) t = x[i]; }
  return t;
}
```

only int is allowed.

possible only int or float type.

(286) int arr[] = {2, 5, 6, 3, 9, 9, 4};

```
maxElement<int, 7>(arr)
```

typename keyword.

eg

```
template<class T> f(T a) {
    T::name *p;
}
```

Where from above expression mean.

(i) T::name is a type and p is a pointer of that type

(ii) T::name and p are variable and this is a multiplication.

to resolve this typename is used

```
template<class T> f(T, a) { T::
name *p; } // multiplication
```

```
template<class T> f(T a) {
    typename T::name *p; } // type
```

↓ we write above typename.

Note: The keyword class and typename have almost the same meaning in a template parameter

(ii) typename is allowed to tell the compiler that an expression is a type exp.

(87) Class template : code reusable Data structure

* Data structure are generic - same interface, same algo
* C++ implementation are diff due to element type

A class template
- describes how a class should be built.
- supplies the class description and the dependent name (as a placeholders)
- is a
  - (a) parameterized type with
  - (b) parameterized member fn
- is often used for containers class

Ex.

```
template <class T>
class Stack
{
    T data[100];
    int top;
public:
    Stack() : top(-1) { }
    ~Stack() { }
    void push(const T& item)
    {
        data[++top] = item;
    }
```

```
void pop()
{
  --top;
}
const T& top()
const T& top() const
{
  return data[top];
}
bool empty() const
{
  return top==-1;
}
```

How to instantiate it

```
char str[10]="ABCDE";
```

Stack<char> s

```
int arr[]={10,2,30};
```

Stack<int> s

Template parameters in class template.

(i) may be of any type (including user defined type)

(ii) may be parameterized type (i.e templates)

(iii) must support the methods used

by the template fn.
- what are the required constructor
- what are required operator fn
- what are the necessary debug operations?

class template instantiation

(i) template<class T> class Stack;
is a forward declaration.

(ii) Stack<char> s; is error without defining

(iii) Stack<char> *sp okay

(iv) void ReverseString(Stack<char> &s, char *str) Okay.

Note: It's not necessary that we have to instantiate everything together if we are instantiating a ref or pointer to the template class. then we may not need the whole definition of the class we could just manage with forward declaration of template class

(190) partial template Instantiation and default template parameter

default parameter

template < class T1 = int, class T2 = string >

↓ if we don't pass T1 = int, T2 = string

ex. ___ version 1

```
template < class T1 = int, class T2 = string >
class student
{
    T1 roll;
    T2 name;
public: student( T1 r, T2 n):
    roll(r), name(n) { }
    void print()
};
```

partial template instantiation

partial instantiation of template

```
template < class T1 >
class student < T1, char * >
{
    char * name;
public:
    student(T1 r, char *n) : roll(r)
    , name(new char[strlen(n)+1]) { }
};
```

T1 → roll

(291)

```
name( strcpy( new char[strlen(n)+1],
                n) ) { }

void print()

{

}
```

Calls

```
student<int, string> s1(2, "Ram");
student<int> s2(1, "kamal");
student<> s3(3, "Ramesh");
student<string> s4("F2", "Shiela");
student<int, char*> s5(3, "Ram");
```

Here T1 = string and T2 = default