

void f()

(268)

{
 A a
 try {

 B b;

 g();

 h();

} catch (UserException & ex) — perfect

{ cout << ex.what(); }

Handler

↳ whenever handler terminates it
↳ Recover exception is not returned
↳ return; it will destroy that
↳ object

}

Ex-2

class MyException : public exception {}

class myclass {};

void h() {} myclass a;

// throws 1; → when hit in terms of control
flow we are going out
// throw 2.5;

// throws MyException; → a ~ myclass

// throw exception();

// throws myclass;

↳ a ~ myclass // abnormal execution
↳ destructor will be called here

void g()

{ myclass a;

try { h(); }

}

g();

↳ if not written in try block

→ it will immediately
raise and rethrow the
exception.

(268)

~~catch (int) & cout << "int"; }~~

~~catch (double) & cout << "double" << endl;~~

~~catch (...) & throw; } }~~ destructor will be called for local ellipsis of catch and clause obj

↓
it knows the objects to call

void f()

L myname a:

try

{

g();

}

~~catch (myException) & cout << "myException" << endl; }~~

~~catch (exception) & cout << "exception" << endl; }~~

~~catch (...) & throw; }~~

Y

int main()

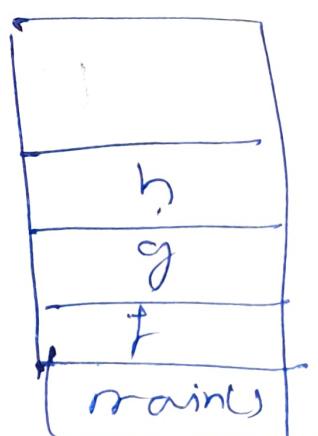
{

try

L f();

}

catch (...) & cout << "Unknown"; }



Y ~~if not catch clause from . main can not know it's not case from another handler will be called.~~

~~(209)~~ catch clause do not catch by implicit conversion.

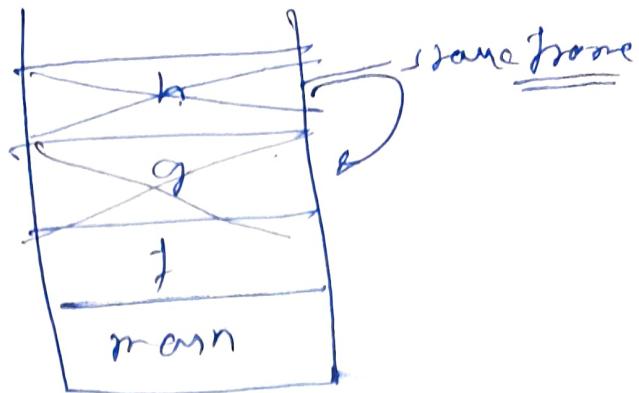
~~(210)~~ Overload resoln allowing implicit conversion. This is not allowed in catch clause

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

- * This not necessary that the caller will handle the exception. caller might not it will return 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 then ~~it can catch it by default~~ as no catch clause it by default returns it (except for main since there is no caller it will call abort)

catch (A*) { handle --
throw }

(270)



- * ~~Destructor of all Object which are live at throw point, their destructor will get called and only after that destruction will be removed).~~

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

try block! Exception Scope:

- * ~~try block~~ : done area first night throw exception
- * function try block
 - * Area for detection in the entire fn body.

```
void f()
```

```
try {
```

throws

```
}
```

```
catch (Exception e) {
```

```
}
```

Nested try block (28) \rightarrow semantically equivalent to nested
blocks

try {

 try L throw E1; }
 catch (E2) L {

}
 catch (E3) L {

 }

}

Catch block: Exception arguments

Catch block

~~① None for exception handler~~
~~→ Catching an exception is like~~
~~invoking a fn~~

② Immediately follow the try block

③ Unique formal parameter to each Handler

④ Can be simply a type name to distinguish its handler from others

How should we pass arguments to catch block

Usually as reference

* wr

~~* exception objects are special kind of~~

automatic objects whose life time is dependent on run-time but not on compile time

(27) And it is always be executed on free space
Heap.

try-catch: Exception matching

Exception matching

- (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
class (B &) child class
class (RA) and then
for Base class

try-catch: exception matching

- 1 In the order of appearance with the matching
 - * If base class catch block ~~protects~~ precedes derived class catch block then ~~(#)~~ compiler issues a warning and continues

(23) ~~⑪~~ unreachable code (derived class)
ignored

- * catch(...) block must be the last catch block because 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 blocks

⑫ static unknown

- * If eventually no handlers found, terminate is called

throws Expression: Exception raise

- ⑬ Expression is treated the same way as
 - * A function argument in a call or the operand of a return stmt.

⑭ Exception context

- * class Exception

⑮ The Expression

- * throws an exception object to throw

eg throws Expression

- * or copies an existing exception object to throw

(27)

- 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 can not be

① An incomplete type like void

array of unknown size or of

element of the incomplete type

declared but not defined (struct,

union/enum/Class Objects or

pointers to such objects

② A pointer to an incomplete type

* except void*, const void*,

volatile void*, const volatile

void*.

(re)throw - throwing again (22)

Rethrow

| can may pass ~~the~~ on the exception
after Handling.
* rethrow is not same as throwing
again.

Throwing again

try { ... }

 catch (Exception ex)

 // Handle and

 // Raise again

 throws ex

 // ex copied

 // ex destroyed

}

Rethrow

try { ... }

 catch (Exception ex) { }

 // Handle and

 | pass on

 throws

 // No copy

 // No destruction

(276)

advantage

- (I) ⚫ desnuor survey
- (II) Unobtrusive
- (III) precise
- (IV) Native & standard
- (V) scorable
- ~~(VI)~~ Faulty & distort

(272) generic programming concept or meta programming

concept

- * The core idea of template is, we would be able to write a ~~as~~ code which at the time of compilation can generate further new code and that generated code will get compiled again.

*

what's a template

- * Templates are specifications of a collection of ~~fn~~ or classes which are parameterized by types.

Ex:

- ① Function search, min, max etc.

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

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

- ② Classes list, queue etc.

- * The class members and the methods are almost the same for list of no or list of objects.

* 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 on

- (I) int max(int x, 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 args
- (II) different code versions

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)

(279)

L.

return strcmp(x, y) ? x : y;

Q) return strcmp(x, y) > 0 ? x : y;

manners

int a = 3 b = 5

double c = 2.1, d = 3.2

max(a, b)

min(2.1, 3.2)

char s1 = new char[6], s2 = new char[5]

strcpy(s1, "blank"); strcpy(s2, "white");

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

other soln?

function template:

① describe how a function should be

built

② supplies a defn or rule for using
some arbitrary type (as place holder)
& a parameterized defn type
as parameters

~~③~~ considered the defn for
several overlaoded version other
fn

(1) Every template parameter is a built-in type or class type parameter

(280)

Template < class T > or

Template < typename T >

Ex:

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

int a = 3, b = 5

double c = 2.1, d = 3.7

int res = max< int >(a, b);
↓ type
instantiated name

We can use the code of max and take T as int or a generic type code of max taking T as height and then we can call on to max fn

double res = max< double >(c, d);

Template specialization → first specialized template preferred

If we compare two pointers it will compare addresses.

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

(281)

template < class T > T max(T, T)

{ return $\pi > y ? \pi : y;$

}

template < > // template specialization for
char* typechar* max(char*) (char* π , char* y)

↓
 returns ↓ specialized template we have specialized
 name of fn ↓ above
template

{ return strcmp(π , y) > 0 ? π : y ; }char* π = new char[6];char* y = new char[6];strcpy(π , "black");strcpy(y , "white");max(char*) (π , y)Implicit instantiation Hierotype of parameter
will tell us what is the value of T.

template < class T > T max(T, T)

{ return $\pi > y ? \pi : y;$

}

(202) $\text{int } a=3, b=5$

$\text{double } c=2.1, d=3.7$

$\text{int res} = \max(a, b)$

implicit instantiation

borrow
int from
OK.

$\text{double res} = \max(c, d)$

double → double

Borrow double and

consistency

$\max(a, c)$

// NOT work

int ↗ f double

but,

$\max(\text{double})(a, c)$

will

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

Template Argument Deduction: Implicit instantiation

① each item in the template parameter list is a template argument.

② When a template fn is invoked, the value of the template arguments are determined by seeing the type of the arguments.

template < i > $T \geq T_{\max}(T_a, T_b)$; (283)

template < char* >

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

template < const int > Type
max(T m[2]);

int a, b, max(a, b) // Bonds to max(m)
(const, int)

double c, d max(c, d)

↓

bonds to max(double) (double,
double)

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

↓
Bonds to max(char*) (char*, char).

int pval[9]; max(pval) // error

three kinds of conversion are allowed in

template:

- ① L-value transformation\ array to pointer conversion
- ② Qualification conversion (const to non const)

(84) (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=0.0)
: 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> + max(Tx, Ty)

{ return mx>yx? mx: y; }

}

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

~~char *x, char *y)~~

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

}

285

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

[course max(c1, c2) < Land 1.]

Template Overload → we can put type name
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; }

temp

✓ template < class T (int size) T
max (T x[size]) ↓ only int is
allowed.

{ T t = x[0];

for (int i = 1; i < size; i++)

{ if (x[i] > t)

t = x[i];

possible only

}

return t;

}

if like int
type.

(286)

int arr[] = {2, 5, 6, 3, 9, 0, 4};

Max<int, 7>(arr) ↗

typename keyword.

∅ co

template <class T> f(T, n) {
T *; name *p; }

What does above expression mean.

- ① T *; name is a type and p is a pointer of that type
- ② T *; name and p are variable and this is a multiplication.
to resolve this typename is used

template <class T> f(T, n) < T *;
name *p; } // multiplication

template <class T> f(T, n) {

typename T *; name *p; } // type
↓ we outcome actual typename.

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

~~No~~ typename is also used to tell the compiler that expr expression is a type expr.

Q87

Class template include reusable Data structure

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

A class template

- describes how a class should be built.
- opposite supplies the class description and its defn. Otherwise fn uses some arbitrary type name. (as a place holder)
- is a
 - (a) parameterized type with
 - (b) parameterized mem fn
- is often used for container class

Ex:

template < class T >

class Stack

private: T data[100];

int top;

public:

Stack(): top(-1); { }

~Stack(); { }

void push(const T& item)

{ data[++top] = item; }

```

void push
{
    --top;
}

const T& pop()
const T& top() const
{
    return top
    return data[top];
}

bool empty() const
{
    return top == -1
    ret
    bool empty() const
    {
        return top == -1;
    }
}

```

How to instantiate it

char str[10] = "ABCDE";

Stack char's

int arr[] = {10, 2, 3}

Stacking

Template parameters in class template.

- ① may be of any type (including user defined types)
- ② may be parameterized type (i.e templates)
- ③ must support the methods used

by the template fn. (289)

- what are the required constructor
- what are required operator
- fn
- what are the necessary defining operations?

Class template instantiation

- (i) template<class T> class Stack;
is a forward declaration.
- (ii) Stack<char> { } 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
are everything together if we are instantiating
a ref or pointer to the template class.
then we may not need the whole defn
of the class we could just merge with
forward declaration of template class

(290) partial template instantiation and default template parameter

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

R

if we do not
pass in T1 = int
T2 = string

Ex. version 1

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

class ~~student~~ student

h. T1 roll;
 T2 name;

public: student(T1 r, T2 n):
roll(r), name(n) {}

void print

};

partial template instantiation

perhaps, instantiate
of template

template < class T1 >

class student < T1, char * >

char *

h. T1roll

d

~~char * name~~

char * name;

public:

student(T1 r, char *n): roll(r)
, name(~~string~~ [new char[~~strlen~~]]) {}

(29) name(strepy(new char[n+1]),
n).) ↴ 9

void print()

Q

Cars

student <int, string> s1(2, "Ranu")

student <int> s2(1, "Karan")

student > s3(3, "Ramesh")

student <string> s4("P2", "Shilpa")

✓ student <int, char*> s5(3, "Ramu")

Here T1 = String and T2 = character