CSC230

Intro to C++ Lecture 16

Outline

- Template in C++

- Exceptions

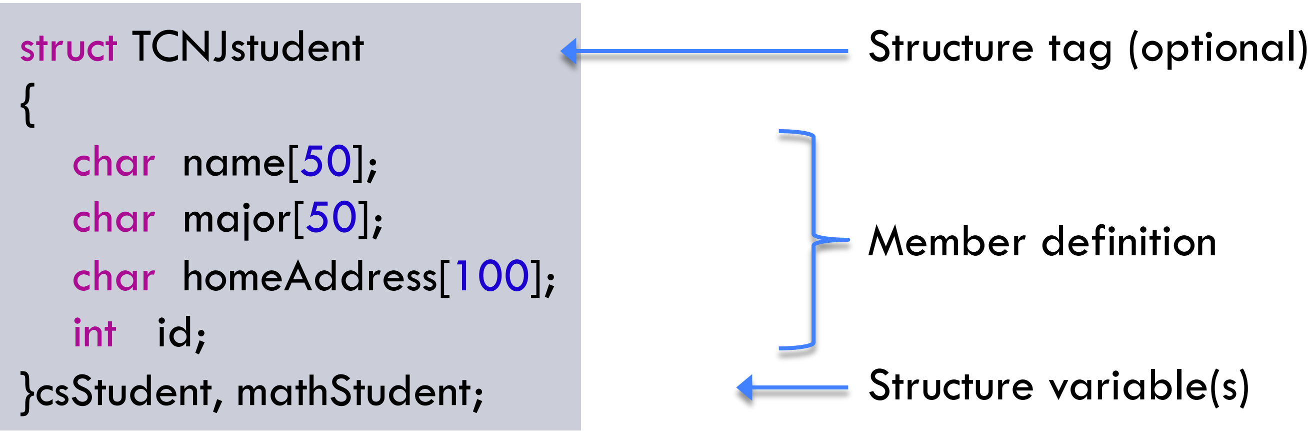
C Structure

array:

- User defines
- Combine multiple data items of same type

structure:

- User defines
- Combine multiple **data** items of different types



```
struct TCNJstudent
{
    char  name[50];
    char  major[50];
    char  homeAddress[100];
    int   id;
}csStudent, mathStudent;
```

Structure tag (optional)

Member definition

Structure variable(s)

C++ Structure

In C++, you can have functions inside a structure. By default, the access is **public**.

- It looks like a class, which has data and functions as well
- In C++, the default access of **class** members is **private**.

```
#include <iostream>
using namespace std;

struct TCNJstudent
{
    char  name[50];
    char  major[50];
    char  homeAddress[100];
    int   id;

    void getId(){
        cout << id << endl;
    }
}csStudent, mathStudent;

int main(){
    csStudent.id = 100;
    csStudent.getId();
}
```

C++ class

```
#include <iostream>
using namespace std;

class TCNJstudent
{
    char  name[50];
    char  major[50];
    char  homeAddress[100];
    int   id;

    void getId(){
        cout << id << endl;
    }
}csStudent, mathStudent;

int main(){
    csStudent.id = 100;
    csStudent.getId();
}
```

✗

```
#include <iostream>
using namespace std;

class TCNJstudent
{
    char  name[50];
    char  major[50];
    char  homeAddress[100];
public:
    int   id;

    void getId(){
        cout << id << endl;
    }
}csStudent, mathStudent;

int main(){
    csStudent.id = 100;
    csStudent.getId();
}
```

✓

Template

Generic programming: Writing code in a way that is **independent** of any particular **type.**

Template: A blueprint for creating a generic class or function.

```
#include <iostream>
using namespace std;

int sum (int x, int y)
{
  return x+y;
}

double sum (double x, double y)
{
  return x+y;
}

int main ()
{
  cout << sum (3,4) << '\n';
  cout << sum (1.5,5.0) << '\n';
  return 0;
}
```

```
#include <iostream>
using namespace std;

template <class T>
T sum (T x, T y)
{
  T result;
  result = x + y;
  return result;
}

int main () {
  int a=1, b=2, u;
  double m=1.0, n=4.5, v;
  u=sum<int>(a,b);
  v=sum<double>(m,n);
  cout << u << '\n';
  cout << v << '\n';
  return 0;
}
```

Template

Function template:

```
template <class type> ret-type func-name(parameter list)
{
   // body of function
}
```

```
#include <iostream>
using namespace std;

template <class T, class U>
bool are_equal (T x, U y)
{
  return (x==y);
}

int main ()
{
  if (are_equal(5,5.0))
    cout << "equal\n";
  else
    cout << "not equal\n";
  return 0;
}
```

Template

Class template:

```
template <class type> class class-name {
. . .
}
```

constructor

Function definition

Function implementation

```
#include <iostream>
using namespace std;
template <class T>
class pairT {
  T x, y;
public:
  pairT (T m, T n)
  {x=m; y=n;}
  T max ();
};
template <class T>
T pairT<T>::max ()
{
  T result;
  result = x>y? x : y;
  return result;
}
int main () {
  pairT <int> obj (300, 15);
  cout << obj.max();
  pairT <double> obj2(5.0, 3.0);
  cout << obj2.max();
  return 0;
}
```

Template

- The **type** of the variable in a class or function to be a **parameter** specified by the programmer
- **Compiler** generates separate class/struct code versions for any type desired (i.e. instantiated as an object)
  - **pairT <int> obj (300, 15);** generates a int version of the object.
  - **pairT <double> obj2(5.0, 3.0);** generates a double version of the object.

Template, caveat

- For **normal class**, the class definition should be in a header file (.h file) and implementation should be in a .cpp file.

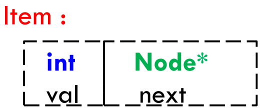
- The **template** class implementation **MUST** be in header file.

- You cannot pre-compile a template file because compiler has no idea what data type should be used inside the template class.

Template example: Linked List

- This is our original List class definition
- Each node has an int value and a next pointer
- Do we need to define a different class for **double** value?

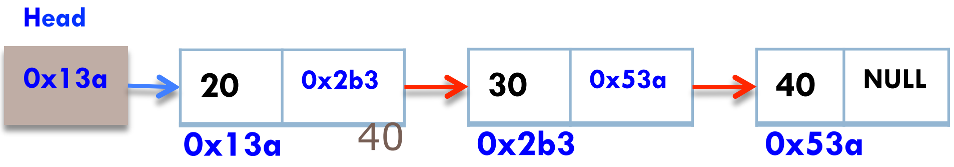
```
#include<iostream> using  namespace  std;

struct  Node{

 int  val;

    Node*  next;
};

class  List
{
public: List();
~List();
void  append(int  v);  ... private:
Node*  head;
};
```



Item :

int
val

Node*
next



Head

0x13a

20 | 0x2b3

30 | 0x53a

40 | NULL

0x13a

0x2b3

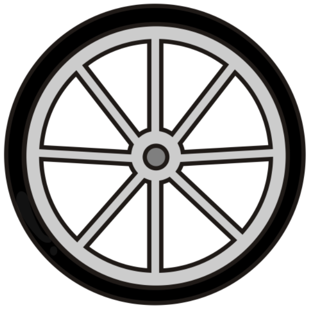
0x53a

Template example: Linked List

Some function needs cin, cout

This struct is a template, U is a **placeholder** for data type

This class is a template, T is a **placeholder** for data type



Function implementation, V is a **placeholder** for data type

```
using namespace std;

template <class U>
struct Node{
  U val;
  Node<U>* next;
};

template <class T>
class List
{
public:
  List();
  ~List();
  void append(T v);
private:
  Node<T>* head;
};

template <class V>
List<V>::List(){
  head = NULL;
}
……
```

List.h

Template example: Linked List

Import template file

Create a List **object** with **int** as data type

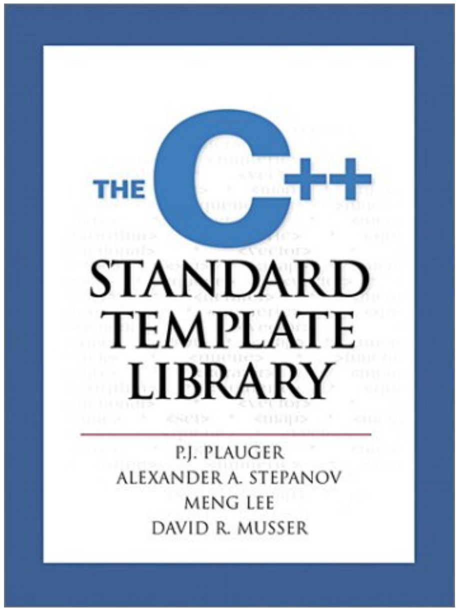
int will replace T in the List template, including
Node<**T**>* head;
After replacement, we have
Node<**int**>* head;

Then, int will replace **U** in Node structure.

```
#include<iostream>
#include "List.h"
using namespace std;

int main(){
  List<int> list1;
  list1.append(20);
  list1.append(30);
  list1.append(40);
}
```

List.cpp

STL



THE C++ STANDARD TEMPLATE LIBRARY

P.J. PLAUGER
ALEXANDER A. STEPANOV
MENG LEE
DAVID R. MUSSER

Standard Template Library (STL)

- vector
- list
- slist
- queue
- deque
- priority queue
- stack
- set
- map
- hash_map
- …

The standard library saves programmers from having to reinvent the wheel.
----- Bjarne Stroustrup

STL

container	Description
Vector	A dynamic array with the ability to resize itself automatically when inserting or deleting an object.
List	A doubly linked list.
Slist	A singly linked list.
Queue	Provides FIFO queue interface with push/pop/front/back operations.
Priority queue	Provides priority queue interface with push/pop/top operations.
Stack	Provides LIFO stack interface with push/pop/top operations (the last-inserted element is on top)
Set	A mathematical set
…	…

Array vs. vector

```
size_t size = 10;
int sarray[10];

// do something with them:
for(int i=0; i<10; ++i){
    sarray[i] = i;
}
delete [] sarray;
```

```
#include <vector>
//...
size_t size = 10;
std::vector<int> array(size);
// make room for 10 integers,
// and initialize them to 0
// do something with them:
for(int i=0; i<size; ++i){
    array[i] = i;
}
```

```
// no need to delete anything
```

See the difference?

Vector example

```
#include <iostream>
#include <vector>

int main ()
{
  std::vector<int> myvector;      // empty vector of ints
  std::vector<int>::size_type sz;
  myvector.push_back (1);         // append 1 to vector
  myvector.push_back (2);
  myvector.push_back (3);
  myvector[2]=5;                  // assign 5
  sz = myvector.size();           // vector size
  for (unsigned i=0; i<sz; i++)
    std::cout << ' ' << myvector[i];
  std::cout << '\n';
  return 0;
}
```

What if?

```
#include <iostream>
#include <vector>

int main ()
{
  std::vector<int> myvector;      // empty vector of ints
  std::vector<int>::size_type sz;
  myvector.push_back (1);         // append 1 to vector
  myvector.push_back (2);
  myvector.push_back (3);
  myvector[20]=5;                 // assign 5
  sz = myvector.size();           // vector size
  for (unsigned i=0; i<sz; i++)
    std::cout << ' ' << myvector[i];
  std::cout << '\n';
  return 0;
}
```

Program prints out:

1 2 3

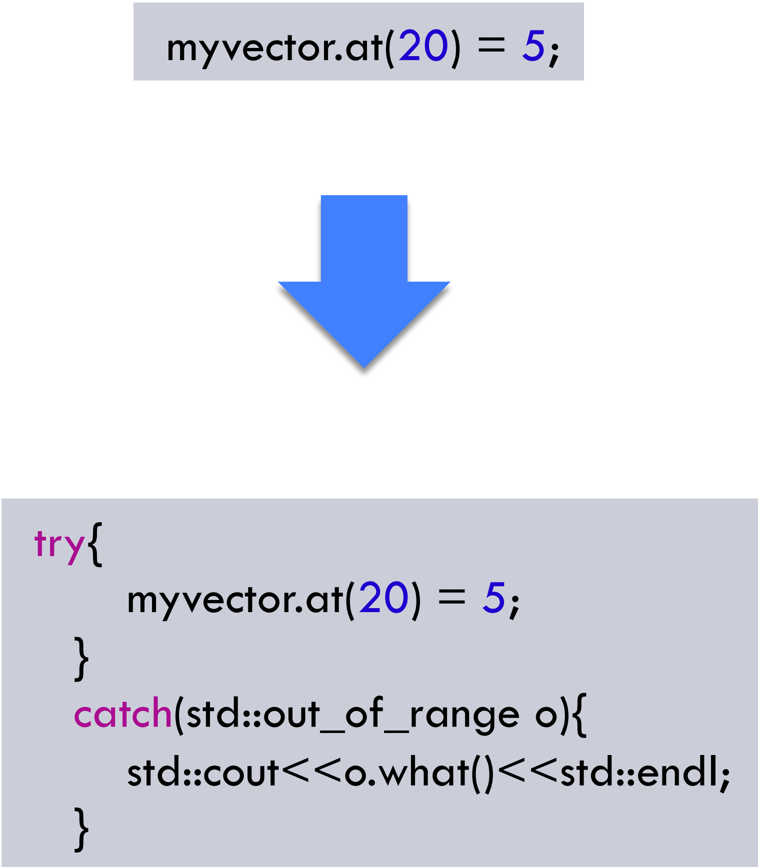
But, the program writes 5 to somewhere out of the bound of myvector vector.

What if?

```
#include <iostream>
#include <vector>

int main ()
{
  std::vector<int> myvector;     // empty vector of ints
  std::vector<int>::size_type sz;
  myvector.push_back (1);        // append 1 to vector
  myvector.push_back (2);
  myvector.push_back (3);
  myvector.at(20)=5;              // assign 5
  sz = myvector.size();          // vector size
  for (unsigned i=0; i<sz; i++)
    std::cout << ' ' << myvector[i];
  std::cout << '\n';
  return 0;
}
```

```
libc++abi.dylib: terminating with uncaught exception of type std::out_of_range: vector
Abort trap: 6
```

Exceptions



```
myvector.at(20) = 5;
```

```
try{
        myvector.at(20) = 5;
   }
   catch(std::out_of_range o){
        std::cout<<o.what()<<std::endl;
   }
```

Outline

- □ Template in C++
- □ Exceptions

Exceptions handling

When something goes wrong in one function, how should we notify the function caller?

- Return a special value to the caller?
- Return a boolean value to the caller?
- Set a global variable? (Toyota, is it your style?)
- Print out a message?
- Print out a message and exit the program?
- Handle the problem without telling the caller?
- Set a failure flag?
- Example : divide-1.cpp

```
#include <iostream>     // std::cerr
#include <fstream>      // std::ifstream

int main () {
  std::ifstream is;
  is.open ("test.txt");
  if ( (is.rdstate() & std::ifstream::failbit ) != 0 )
    std::cerr << "Error opening 'test.txt'\n";
  return 0;
}
```

What is the problem with these options?

All these options are passive (the caller need to check whether there is a problem).

- The function with problem/error should always notify the caller. Do not keep quiet.
- If constructor has a problem
  - It cannot return a value. A constructor does not have a return value.
- The error happens inside a function that does not know how to handle it.
- Example : divide-2.cpp

Exception handling

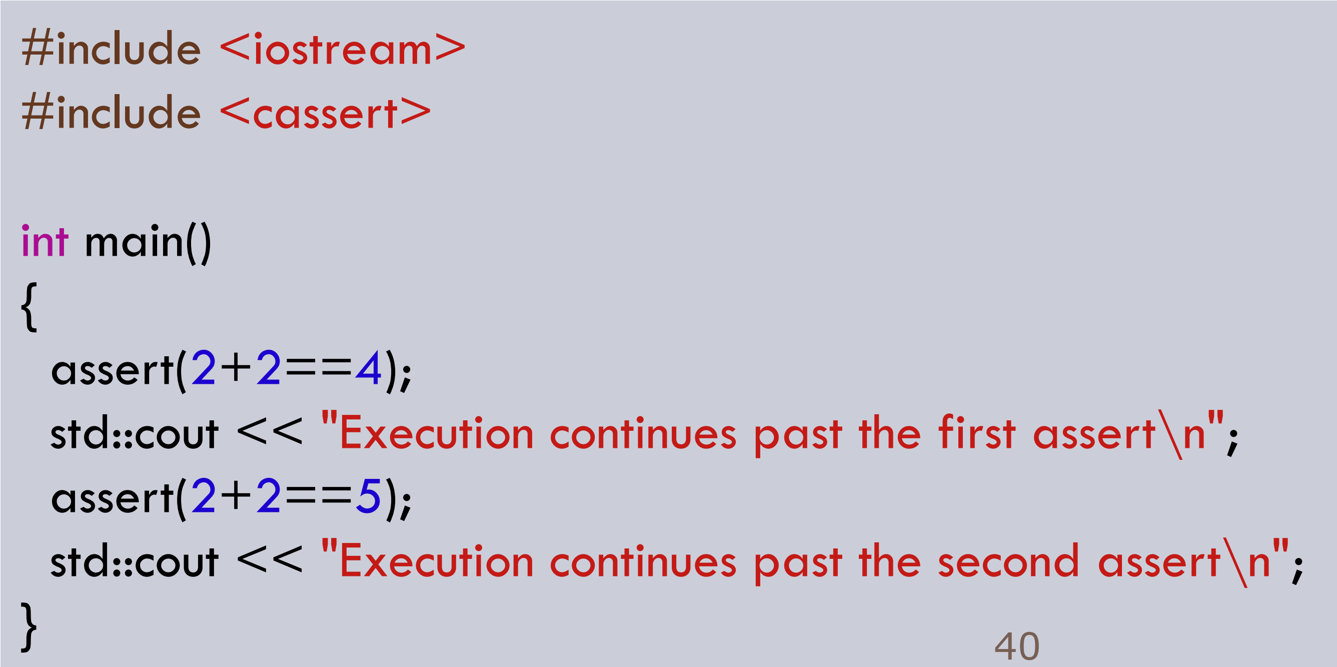
- Caller has a choice on how to handle the problem.
    - The function caused the error does not need to guess what to do.
- The normal control flow and the exception handling are separated.
- The program is easy to read.

```
try
{
   // protected code
}catch( ExceptionName e1  )
{
   // catch block
}catch( ExceptionName e2 )
{
   // catch block
}catch( ExceptionName eN )
{
   // catch block
}
```

assert

The assert statement checks certain boolean condition is true or not. If it is false, the program will be terminated.

- Good for developing/testing
- Not good for final product
- assert is usually used for testing / you can turn on or off the assertion\
- What is the difference between assert and exception?



```
#include <iostream>
#include <cassert>

int main()
{
  assert(2+2==4);
  std::cout << "Execution continues past the first assert\n";
  assert(2+2==5);
  std::cout << "Execution continues past the second assert\n";
}
```

Why exception?

- With exception handling, a program can continue executing (rather than terminating) after dealing with a problem.

- This helps to support robust applications that contribute to *mission*
  - *-critical* computing or *business-critical* computing

- When no exceptions occur, there is no performance reduction
- Example : divide-3.cpp

throw statement

- Exception can be thrown anywhere within a code block
- throw statement creates an exception
- The value (operand) of the throw statement determines the type of exception
- The operand of the throw statement can be any expression

```
double division(int x, int y)
{
    if( y == 0 )
    {
        throw "Division by zero condition!";
    }
    return (x/y);
}
```

try Blocks

- Keyword **`try`** followed by braces (**{ }**)
- What should enclose?
  - Statements that might cause exceptions
  - Statements that should be skipped in case of an exception
  - revisit : divide-3.cpp

catch Handlers

- Immediately follow a **try** block
  - One or more **catch** handlers for each **try** block
- Keyword **catch**
- Exception parameter enclosed
  - Represents the type of exception to process
  - Can provide an optional parameter name to interact with the caught exception object
- Executes if exception parameter type matches the exception thrown in the **try** block
  - Could be a base class of the thrown exception's class

Catching exceptions

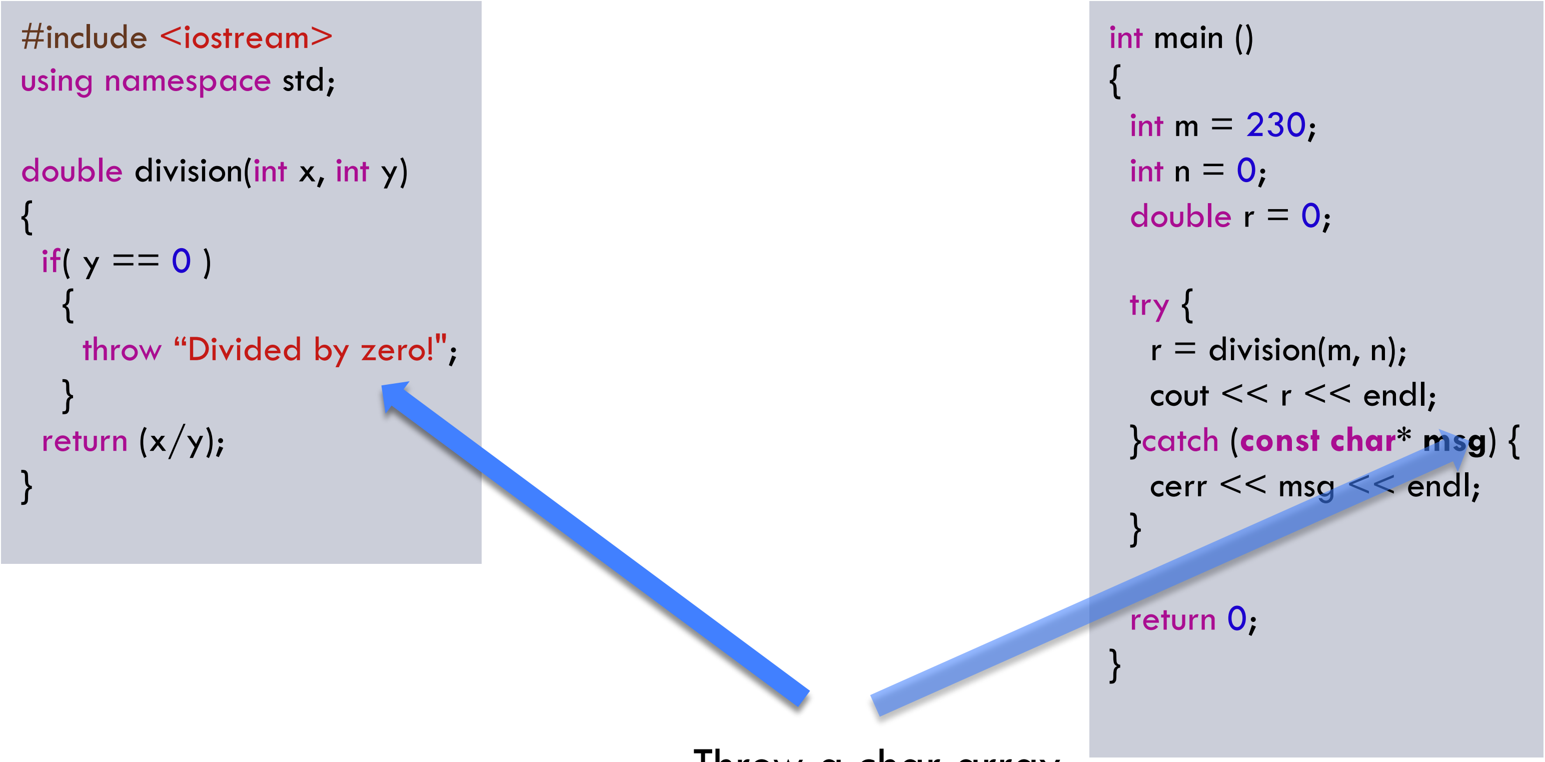
- You can specify what type of exception to catch

```
try
{
   // protected code
}catch( ExceptionName e )
{
  // code to handle ExceptionName exception
}
```

- Above code will catch an exception of ExceptionName type.
- If you want to catch any exceptions, you must put an ellipsis, ….

```
try
{
   // protected code
}catch(…)
{
  // code to handle any exception
}
```

Exception example



```
#include <iostream>
using namespace std;

double division(int x, int y)
{
  if( y == 0 )
    {
      throw “Divided by zero!";
    }
  return (x/y);
}
```

```
int main ()
{
  int m = 230;
  int n = 0;
  double r = 0;

  try {
    r = division(m, n);
    cout << r << endl;
  }catch (const char* msg) {
    cerr << msg << endl;
  }

  return 0;
}
```

Throw a char array
Catch a char array

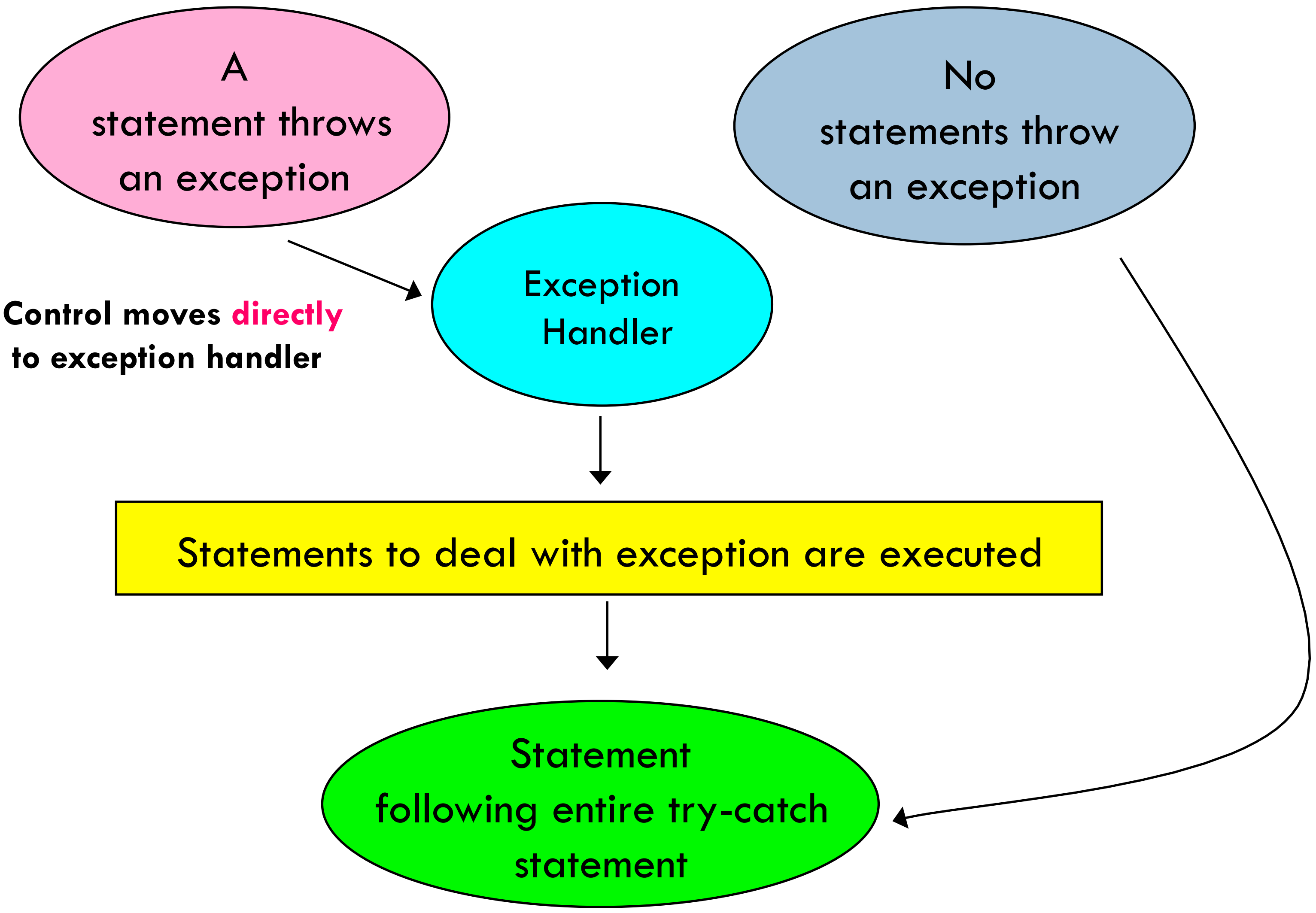
try-blocks and if-else

- try-blocks are very similar to if-else statements
  - If everything is normal, the entire try-block is executed
  - else, if an exception is thrown, the catch-block is executed
- A big difference between try-blocks and if-else statements is the try-block's ability to send a message to one of its branches

Example of a `try-catch` Statement

```
 try
{
        // Statements that process personnel data and may throw
        // exceptions of type int, string, and SalaryError
}
catch ( int )
{
        // Statements to handle an int exception
}
catch ( string s )
{
       cout << s << endl;  // Prints "Invalid customer age"
        // More statements to handle an age error
}
catch ( SalaryError )
{
       // Statements to handle a salary error
}
```

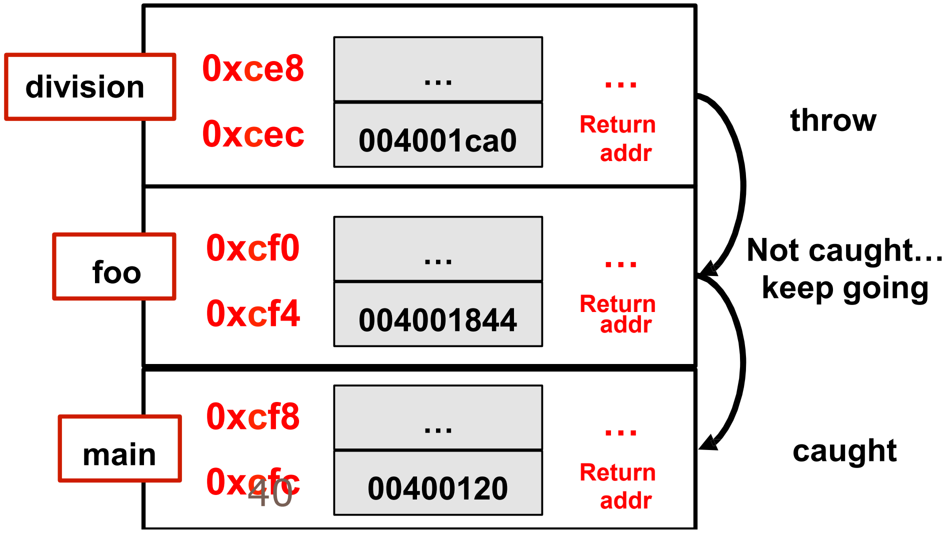
Execution of try-catch


A statement throws an exception
No statements throw an exception
Control moves directly to exception handler
Exception Handler
Statements to deal with exception are executed
Statement following entire try-catch statement

Who will catch

```
#include <iostream>
using namespace std;

double division(int x, int y)
{
   if( y == 0 )
     {
       throw “Divided by zero!";
     }
   return (x/y);
}

double foo(int x, int y)
{
   return division(x, y);
}
```

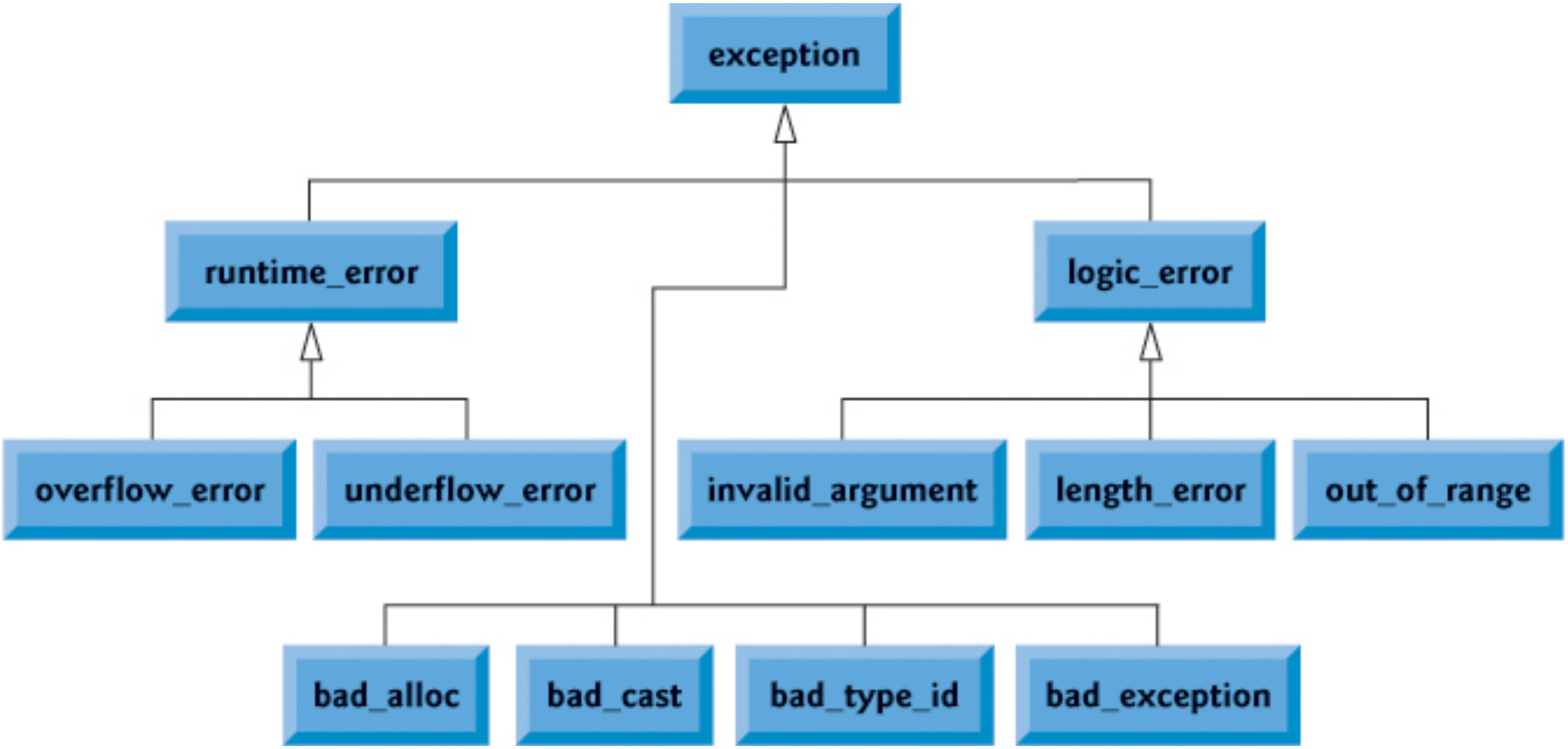
```
int main ()
{
   int m = 230;
   int n = 0;
   double r = 0;

   try {
     r = foo(m, n);
     cout << r << endl;
   }catch (const char* msg) {
     cerr << msg << endl;
   }
   return 0;
}
```



Throw something meaningful

- In general, do not throw primitive values, such as int or float
  - *throw 200*
  - It is hard for other function to figure out the meaning of the number
  - It does not provide context information
- In general, do not throw a string
  - It is easy for human being to understand
  - But it is hard for other function to figure out
- Use a **class**, especially those defined in <stdexcept>
  - *throw std::invalid_argument("value is negative");*
  - *throw std::runtime_error("Failed");*
  - Method what() with extra details

std::exception



std::exception

Excetption	**Description**
std::bad_alloc	Can be thrown by **new**
std::bad_cast	Can be thrown by **dynamic_cast**
std::bad_typeid	Can be thrown by **typeid**
std::logic_error	An exception can be detected by READING the code
std::domain_error	Caused by Mathematically invalid domain
std::invalid_argument	Caused by invalid arguments
std::length_error	Cause by a too big std::string
std::out_of_length	Caused by std::vector, std::bitset<>operator[]()
std::runtime_error	An exception can not be detected by reading the code
std::overflow_error	Caused by mathematical overflow

Define new exceptions

```
#include <iostream>
#include <exception>
using namespace std;

struct NewException : public exception
{
  const char * what ()
  {
    return "Exception";
  }
};
```

Inherits and overrides exception class

what() is defined in exception class, and overridden by every child exception class

```
int main()
{
  try
    {
      throw NewException();
    }
  catch(NewException& e)
    {
      std::cout << "NewException caught" << std::endl;
      std::cout << e.what() << std::endl;
    }
  catch(std::exception& e)
    {
      //Other errors
    }
}
```

Examples

- Check out some examples