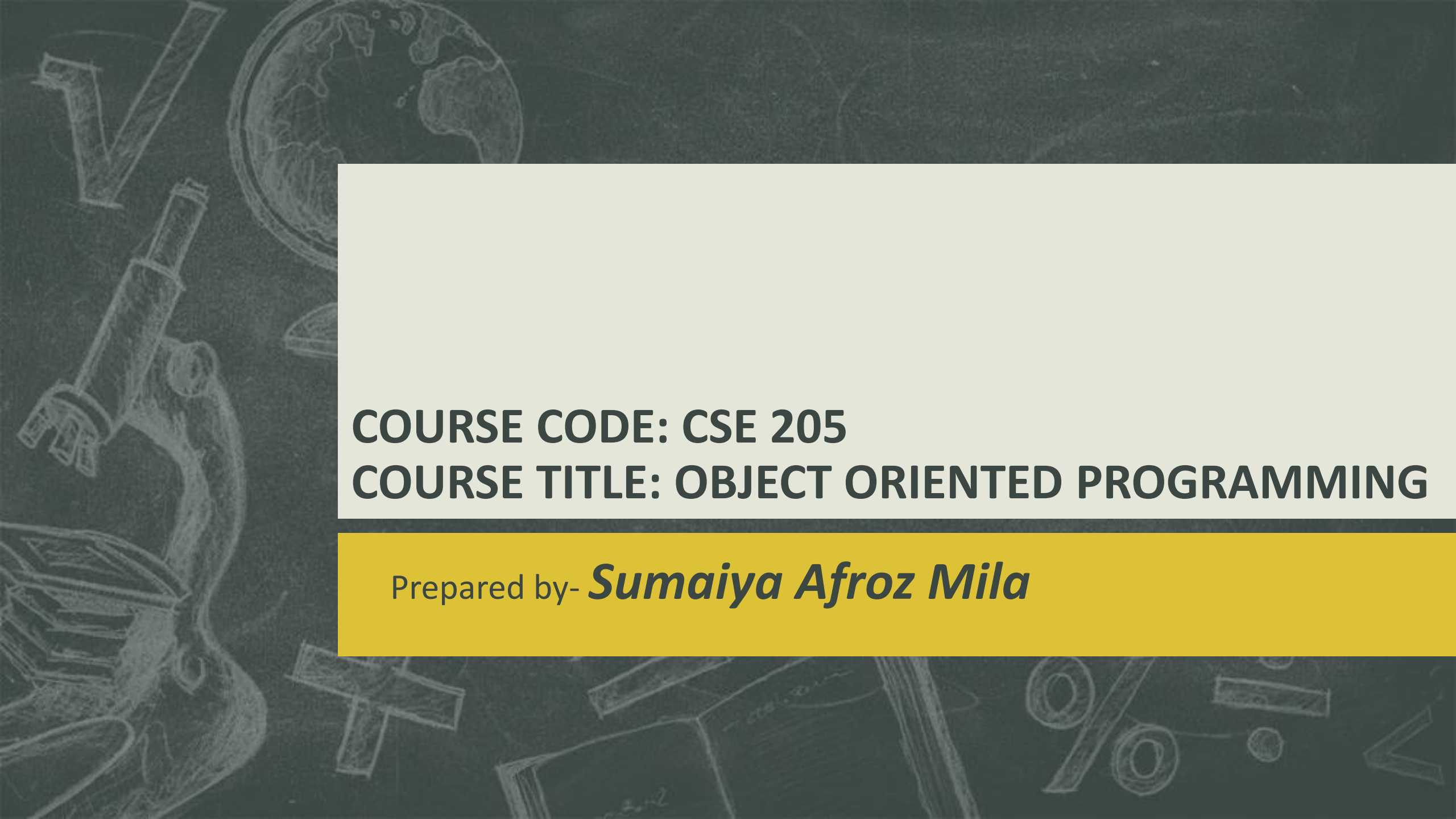


COURSE CODE: CSE 205
COURSE TITLE: OBJECT ORIENTED PROGRAMMING

Prepared by- ***Sumaiya Afroz Mila***

Exception Example 1

```
double divide(double a, double b){
        double result=a/b;
         return result;
}
```

What if b=0??

```
main(){
   double a,b,c;
   cin>>a>>b;
   c=divide(a,b);
   cout<<c;
}
```

This is divide by zero exception

Exception Example 2

```
class Numlist{
  int* a;
  int size;
  public:
  Numlist(int s) {a=new int[s]; size=s; }
  void putAt(int index, int v){a[index]=v;}
  int getAt(int index){ return a[index];}
};
```

```
main(){
   Numlist list(100);          int index, val;
   cin>> index>>val;
   list.putAt(index,val);
   cout<< list.getAt(index);
}
```

What if

index<0 ?

index>100 ?

Array Index out of bound Exception

Exception Example 3

```
class Numlist{
  int* a;
  int size;
  public:
  Numlist(int s) {a=new int[s]; size=s; }
  void putAt(int index, int v){a[index]=v;}
  int getAt(int index){ return a[index];}
};
```

```
main(){
   Numlist list(100);          int index, val;
   cin>> index>>val;
   list.putAt(index,val);
   cout<< list.getAt(index);
}
```

What if

size> memory available?

Memory Allocation Exception

Exception Example 4

```
void manipList (Numlist* list, int op){
   int index;
   cin>>index;
   if (op==1){ int val; cin>>val;
                  list->putAt(index,val); }
   else if (op==2){cout<<list->getAt(index);}
   return;
}
```

```
main(){
   Numlist* list;
   int option;
   cout<<"Put/Get: ";
   cin>>option;
   manipList (list,option);
}
```

What if

list is not initialized?

Null Pointer Exception

Exception

- A C++ exception is a response to an exceptional circumstance that arises while a program is running, such as, an attempt to **divide by zero**.
- Exceptions provide a way to transfer control from one part of a program to another.
- C++ exception handling is built upon three keywords: **try**, **catch**, and **throw**.

Exception

- **try:** program statements to be monitored for exceptions are contained in try block. A function called from within a try block can throw exception too
- **Throw:** If an exception occurs within the **try block**, it is thrown
- **Catch:** when an exception is thrown, it is caught using **catch,** and processed. There can be more than one catch statement associated with a try

```
try{
        //try block
        throw exception;
}
catch(type1 arg){
        //catch block
}
catch(type2 arg){
        //catch block
}

catch(typeN arg){
        //catch block
}
```

throw statement

- throw is used to signal the fact that an exception has occurred; also called *raise*
- Can throw any valid expression/ object.
- **Syntax:**

  throw expression;

Default catch statement

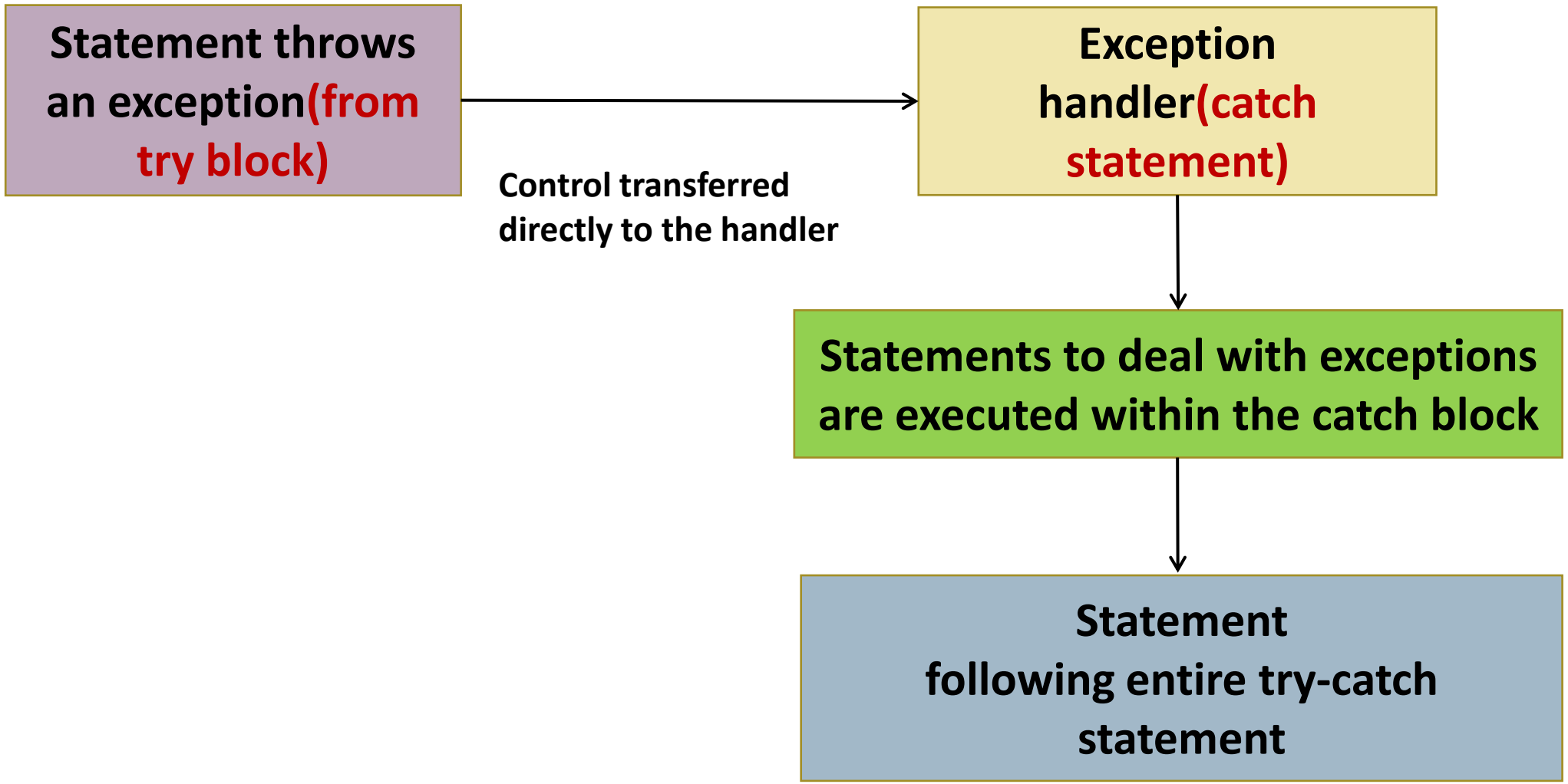
- If an exception is thrown for which there is no applicable catch statement, an abnormal program termination might occur
- In some circumstances you will want an exception handler to catch all exceptions instead of just a certain type
- To solve above situations, we can use default **catch statement**
- **Syntax:**

```
catch(...){
// process all exceptions
}
```

Execution of try-catch

- Once exception is thrown, **control passed** to the **catch** expression and **try block is terminated**
- **Catch** is not called
- Program execution is transferred to it
- After the execution of the catch statement, program continues with the **statement following** the **catch**

Execution of try-catch



Exception

An exception is an unusual, often unpredictable event, detectable by software or hardware, that requires special processing occurring at runtime.

- If without handling,
  - Program crashes
  - Falls into unknown state

Using try-catch

```
main(){
        cout<<“start\n”;
        try{
                cout<<“Inside try block”<<endl;
                throw 10;
                 cout<<“This will not be executed”<<endl;
        }
        catch(int i ){
                cout<<“exception occurred”<<endl;
                cout<<“number is: ”<<i<<endl;
        }
}
```

OUTPUT

start
Inside try block
exception occurred
number is: 10

Using try-catch

```
main(){
        cout<<“start\n”;
        try{
                cout<<“Inside try block”<<endl;
                throw 10;
                 cout<<“This will not be executed”<<endl;
        }
        catch(double i ){
                cout<<“exception occurred”<<endl;
                cout<<“number is: ”<<i<<endl;
        }
}
```

OUTPUT

start
Inside try block
Abnormal Program Termination

This program produces such output because the **integer exception** will not be caught by a **double catch statement**

SOLUTION??

Write another catch statement with integer argument

Exception thrown from a stmt outside try block

```
void Xtest(int test){
    cout<<"inside Xtest: "<<test<<endl;
    if(test>0)
        throw test;
}
```

```
int main (){
    cout<<"start\n";
    try{
        cout<<"inside a try block\n";
        Xtest(0);
        Xtest(1);
        Xtest(2);
    }
    catch(int i){
        cout<<"caught number: “<<i<<endl;
     }
            cout<<“end”;
}
```

Exception Handling

- An exception handler is a section of program code that is designed to execute when a particular exception occurs
    - Resolve the exception
    - Lead to known state, such as exiting the program

Exception Handling(divide by zero exception)

```
double divide(double a,
                    double b){
   if(b!=0){
        double result=a/b;
        return result;
   }
   else{
      throw  -1;
  }
}
```

```
main(){
   double a,b,c;
   cin>>a>>b;
   try{
            c=divide(a,b);
            cout<<c;
   }
   catch (int i){
       cout<<"Division Exception: "<<i;
   }
}
```

Exception Handling(divide by zero exception)

```
double divide(double a,
                    double b){
    if(b!=0){
          double result=a/b;
          return result;
    }
    else{
       throw  "Divide by
zero";
   }
}
```

```
main(){
   double a,b,c;
   cin>>a>>b;
   try{
            c=divide(a,b);
            cout<<c;
   }
   catch (const char* message){
       cout<<"Exception: ";
       puts(message);
   }
}
```

Exception Handling

```
class MyException{
    char message[30];
    public:
    MyException ( char* m){
                strcpy (message, m);
     }
  char* getMessage(){ return message; }
};
```

```
double divide(double a,  double b){
   if(b!=0){ double result=a/b;
                return result;  }
  else{
     MyException Ex("Divide By Zero");
     throw  Ex ;    }
}
```

```
main(){
   double a,b,c;
   cin>>a>>b;
   try{
                c=divide(a,b);
                cout<<c;
   }
   catch (----------? e){
       cout << "Code: ";
       cout << e.getMessage();
   }
}
```

Exception Handling

```
class MyException{
    char message[30];
    public:
    MyException ( char* m){
                strcpy (message, m);
     }
  char* getMessage(){ return message; }
};
```

```
double divide(double a,  double b)
{
   if(b!=0){ double result=a/b;
                  return result;  }
   else{
     MyException Ex("Divide By Zero");
     throw  Ex ;    }
}
```

```
main(){
   double a,b,c;
   cin>>a>>b;
   try{
                c=divide(a,b);
                cout<<c;
   }
   catch (MyException e){
       cout << "Code: ";
       cout << e.getMessage();
   }
}
```

Anonymous object

- In certain cases, we need a variable only temporarily.

- This is done by creating objects like normal, but omitting the variable name.

Example:

```
ComplexNumber A(10, 2)          //normal object
ComplexNumber(10, 2)    // Anonymous Object
```

Multiple Exception

```
#define MAX 3
class StackFull { };
class StackEmpty{ };
class Stack
{
      int S[MAX], top;
      public:
    Stack () { top= -1;}
    void push(int v){
        if (top>= MAX-1)
            throw StackFull();
        S[++top] = v;    }
    int pop(){
        if (top<0)
            throw StackEmpty();
        return S[top--];     }
};
```

```
int main(){
    Stack st;
try{
    st.push(10);
    st.push(20);
    st.push(30);
    st.push(40);
cout<<st.pop()<<endl;    cout<<st.pop()<<endl;
cout<<st.pop()<<endl;    cout<<st.pop()<<endl;
}
catch(StackFull  F){
    cout<<"Exception: Stack is Full" <<endl;
catch(StackEmpty  E){
    cout<<"Exception: Stack is Empty" <<endl;
}
```

Nested try catch

```
try{
        try{
                try{
                }
                catch (dataType varname){
                        //Exception Handler
                }
        }
        catch (dataType varname){
                        //Exception Handler
        }

}
catch (...){
                //Default Exception Handler
}
```

Rethrow an exception

- An exception is rethrown to allow multiple handlers access to the exception
- Perhaps, one exception handler manages one aspect of an exception, and a second handler copes with another
- An exception can **only** be **rethrown** from within a **catch block(or from any function called from within that block)**
- When an exception is rethrown, it is not recaught by the same **catch** statement
- It will propagate to an outer catch exception

Rethrow an exception example

```
void Xhandler(){
    try{
        throw "hello";
    }
    catch(const char *c){
    cout<<"caught char* inside
xhandler\n";
        throw;
    }
}
```

```
int main (){
    cout<<"start\n";
    try{
        Xhandler();
    }
    catch(const char *c){
    cout<<"caught char* inside main\n";
    }
    cout<<"end";
}
```

Restricting Exception in a Function

- You can restrict the type of exceptions that a function can throw back to its caller
- You can in fact prevent a function from throwing any exception
- To do so, a **throw** clause is to be added to the function definition

```
Ret-type func_name(arg list)throw(type list)
{
        //body
}
```

Restricting Exception in a Function

```
// can throw anything
void func();
```

```
// promises not to throw
void func() throw();
```

```
// promises to only throw int
void func() throw(int);
```

```
// throws only char or int
void func() throw(char,int);
```

By default, a function can throw anything it wants.

- A throw clause in a function's signature
  - •Limits what can be thrown
  - •A promise to calling function
  - •If other type thrown, standard library function **unexpected()** is called
- A throw clause with no types
  Says nothing will be thrown
- Can list multiple types
  Comma separated

Source of Exceptions

- Exceptions Thrown by user code, using *throw* statement.

- Exceptions Thrown by the Language

  for example while using: new

- Exceptions Thrown by Standard Library Routines

Exception Thrown by New

```
#include<iostream>
#include<new>
using namespace std;
int main(){
        int *p;
        try{
                p = new int [100000];
        }
        catch (bad_alloc  xa){
                cout<<“allocation Failure”;
        }
        delete [ ] p;
}
```

Good Programming Style with C++ Exceptions

-Don’t use exceptions for normal program flow

- Only use where normal flow isn’t possible

- Don’t let exceptions leave main or constructors

- Violates “normal” initialization and termination

- Always throw some type

- So the exception can be caught

- Resources may have been allocated when exception thrown. Catch handler should delete space allocated by new and close any opened files

- Use exception specifications widely

- Helps caller know possible exceptions to catch


Thank
you!