18CSC202J - OBJECT ORIENTED DESIGN AND PROGRAMMING

Session 1

Topic :Generic - Templates : Introduction

Templates-Introduction

- Allows functions and classes to operate with **generic types**.
- Allows a function or class to work on many **different data types without being rewritten** for each one.
- Great utility when combined with **multiple inheritance and operator overloading**
- The **C++ Standard Library** is based upon conventions introduced by the Standard Template Library (STL)

Types of Templates

- **Function Template**

  A *function template* behaves like a function except that the template can have arguments of many different types

- **Class Template**

  A class template provides a specification for generating classes based on parameters.

  Class templates are generally used to implement containers.

Function Template

- A function templates work in similar manner as function but with one key difference.
- A single function template can work on different types at once but, different functions are needed to perform identical task on different data types.
- If you need to perform identical operations on two or more types of data then, you can use function overloading. But better approach would be to use function templates because you can perform this task by writing less code and code is easier to maintain.

Cont.

- A generic function that represents several functions performing same task but on different data types is called function template.
- For example, a function to add two integer and float numbers requires two functions. One function accept integer types and the other accept float types as parameters even though the functionality is the same. Using a function template, a single function can be used to perform both additions.
- It avoids unnecessary repetition of code for doing same task on various data types.

Cont.

Why Function Templates?

- Templates are instantiated at compile-time with the source code.
- Templates are used less code than overloaded C++ functions.
- Templates are type safe.
- Templates allow user-defined specialization.
- Templates allow non-type parameters.

Function Template

- A function template starts with keyword template followed by template parameter/s inside <> which is followed by function declaration.
- T is a template argument and class is a keyword.
- We can also use keyword **typename** instead of class.
- When, an argument is passed to some_function( ), compiler generates new version of some_function() to work on argument of that type.

Template Syntax

```
template <class T>
T some_function(T argument)
{
 …. … ….
}
```

The template type keyword specified can be either "class" or " typename":

template<class T>

or

template<typename T>

Both are valid and behave exactly the same.

Session 2

Topic :Example Program Function Template, Class Template

Example program Function Templates

```
#include<iostream.h>
template <typename T>
T Sum(T n1, T n2)                    // Template function
{
        T rs;
        rs = n1 + n2;
        return rs;
}
int main()
{
        int A=10,B=20,C;
        long I=11,J=22,K;
        C = Sum(A,B);
        cout<<"nThe sum of integer values : "<<C;
        K = Sum(I,J);
        cout<<"nThe sum of long values : "<<K;
}
```

More than One Template Argument

```
template<class T , class U>
void multiply(T a , U b)
{
    cout<<"Multiplication= "<<a*b<<endl;
}
int main()
{
    int a, b;
    float x, y;
    cin>>a>>b;
    cin>>x>>y;
    multiply(a,b);          // Multiply two integer type data
    multiply(x,y);          // Multiply two float type data
    multiply(a,x);          // Multiply a float and integer type data
    return 0;
}
```

Class Template

- Like function template, a class template is a common class that can represent various similar classes operating on data of different types.

- Once a class template is defined, we can create an object of that class using a specific basic or user-defined data types to replace the generic data types used during class definition.

Syntax for Class Template

```
template <class T1, class T2, ...>
class classname
{
    attributes;
    methods;
};
```

Example Program

```
#include <iostream.h>
using namespace std;
const int MAX = 100;              //size of array
template <class Type>
class Stack
    {
    private:
      Type st[MAX];          //stack: array of any type
      int top;               //number of top of stack
    public:
      Stack()                //constructor
            { top = -1; }
      void push(Type var)          //put number on stack
            { st[++top] = var; }
      Type pop()             //take number off stack
            { return st[top--]; }
    };
```

Cont.

```
int main()
{
    Stack<float> s1;            //s1 is object of class Stack<float>
    s1.push(1111.1F);           //push 3 floats, pop 3 floats
    s1.push(2222.2F);
    s1.push(3333.3F);
      cout << "1: " << s1.pop() << endl;
      cout << "2: " << s1.pop() << endl;
      cout << "3: " << s1.pop() << endl;

    Stack<long> s2;             //s2 is object of class Stack<long>
    s2.push(123123123L);              //push 3 longs, pop 3 longs
    s2.push(234234234L);
    s2.push(345345345L);
      cout << "1: " << s2.pop() << endl;
      cout << "2: " << s2.pop() << endl;
      cout << "3: " << s2.pop() << endl;
    return 0;
}
```

Session 3

Topic :Class Template, Example Program for Class and Function Template

Class Templates with Multiple parameter

- We can use more than one generic data type in a class template.
- Syntax:

```
template<class T1, class T2>
    class classname
    {
        .....
        .....
    };
```

Example Program

```
template<class T1, classT2>
class Test
{
    T1 a;
    T2 b;
}
void show()
{
cout<<a;
cout<<b;
}
};
```

```
int main()
{
test<float, int> test1(1.23, 123);
test<int, char> test2(100,'w');
test1.show();
test2.show();
return 0;
}
```

Output:

```
1.23
123
100
w
```

Class Template Object

To create a class template object, you need to define the data type inside a < > when creation.

Syntax:

className<dataType> classObject;

Example:

className<int> classObject;
className<float> classObject;
className<string> classObject;

Program

1. Program to display largest among two numbers using function templates.
2. Program to swap data using function templates.
3. Program to add, subtract, multiply and divide two numbers using class template.

Solution:1

```
#include <iostream>
using namespace std;
template <class T>
T Large(T n1, T n2)
{
return (n1 > n2) ? n1 : n2;
}
int main()
 {
int i1, i2;
float f1, f2;
char c1, c2;
cout << "Enter two integers:\n";
cin >> i1 >> i2;
cout << Large(i1, i2) <<" is larger." << endl;
cout << "\nEnter two floating-point numbers:\n";
cin >> f1 >> f2;
cout << Large(f1, f2) <<" is larger." << endl;
cout << "\nEnter two characters:\n";
cin >> c1 >> c2;
cout << Large(c1, c2) << " has larger ASCII value.";
return 0;
}
```

Output

```
Enter two integers:
5
10
10 is larger.
Enter two floating-point numbers:
12.4
10.2
12.4 is larger.
Enter two characters:
z
Z
z has larger ASCII value.
```

Solution: 2

```
#include <iostream>
using namespace std;
template <typename T>
void Swap(T &n1, T &n2)
 {
 T temp;
temp = n1;
 n1 = n2;
n2 = temp;
 }
 int main()
 {
 int i1 = 1, i2 = 2;
 float f1 = 1.1, f2 = 2.2;
 char c1 = 'a', c2 = 'b';
 cout << "Before passing data to function template.\n";
 cout << "i1 = " << i1 << "\ni2 = " << i2;
 cout << "\nf1 = " << f1 << "\nf2 = " << f2;
 cout << "\nc1 = " << c1 << "\nc2 = " << c2;
 Swap(i1, i2);
 Swap(f1, f2);
 Swap(c1, c2);
 cout << "\nAfter passing data to function template.\n";
 cout << "i1 = " << i1 << "\ni2 = " << i2;
 cout << "\nf1 = " << f1 << "\nf2 = " << f2;
 cout << "\nc1 = " << c1 << "\nc2 = " << c2;
 return 0;
  }
```

Output

Before passing data to function template.

i1 = 1
i2 = 2
f1 = 1.1
f2 = 2.2
c1 = a
c2 = b

After passing data to function template.

i1 = 2
i2 = 1
f1 = 2.2
f2 = 1.1
c1 = b
c2 = a

Solution: 3

```
#include <iostream>
using namespace std;
 template <class T>
class Calculator
 {
private: T num1, num2;
public: Calculator(T n1, T n2)
{
num1 = n1;
 num2 = n2;
}
void displayResult()
 {
cout << "Numbers are: " << num1 << " and " << num2 << "." ;
cout << "Addition is: " << add() << endl;
cout << "Subtraction is: " << subtract() << endl;
cout << "Product is: " << multiply() << endl;
cout << "Division is: " << divide() << endl;
}
T add()
{
 return num1 + num2;
}
T subtract()
 {
return num1 - num2;
}
T multiply()
{
return num1 * num2;
 }
T divide()
{
 return num1 / num2;
 }
 };
 int main()
 {
Calculator<int> intCalc(2, 1);
Calculator<float> floatCalc(2.4, 1.2);
 cout << "Int results:" << endl;
 intCalc.displayResult();
 cout << endl << "Float results:" << endl;
floatCalc.displayResult();
return 0;
 }
```

Output

Int results:
Numbers are: 2 and 1.
Addition is: 3
Subtraction is: 1
Product is: 2
Division is: 2

Float results:
Numbers are: 2.4 and 1.2.
Addition is: 3.6
Subtraction is: 1.2
Product is: 2.88
Division is: 2

Session 6

Topic :Exceptional Handling: try and catch, multilevel exceptional

Introduction

- It is normal to commit mistakes in programming that prompts unusual conditions called **errors**. These errors are classified as:
- **Syntax Errors** - Errors that occur when you **violate** the **rules** of writing C++ syntax, e.g, missing paranthesis
- **Logical Errors** - These errors solely depend on the **logical thinking** of the programmers.
- **Runtime Errors** - Errors which occur during program **execution(run-time)** after successful compilation are called run-time errors, e.g, division by zero
- **Semantic errors** - Errors due to an **improper use of program statements**

Introduction

- The **errors** that occur at **run-time** are known as **exceptions.**
- They occur due to different conditions such as division by zero, accessing an element out of bounds of an array, unable to open a file, running out of memory and so on
- **Exception Handling** in C++ is defined as a **method** that takes care of a surprising condition like runtime errors.
- At whatever point a sudden situation happens, there is a movement of the program control to a unique function known as **Handlers**.

Exceptions

- Indicate problems that occur during a program's execution
- Occur infrequently
- Exceptions provide a way to transfer control from one part of a program to another.
- 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.

Exception Handling

- Can resolve exceptions
  - Allow a program to continue executing or
  - Notify the user of the problem and
  - Terminate the program in a controlled manner
- Makes programs robust and fault-tolerant
- Types
  - **Synchronous exception** (out-of-range index, overflow)
  - **Asynchronous exception** (keyboard interrupts)

Types of Exception

Two types of exception:

Synchronous Exceptions
Asynchronous Exceptions

Synchronous Exceptions

- Occur during the program execution due to some fault in the input data or technique that is not suitable to handle the current class of data, within the program .

- For example:
  - Errors such as out of range
  - Overflow
  - Underflow and so on

Asynchronous Exceptions

- Caused by events or faults unrelated (external) to the program and beyond the control of the program.

- For example
  - errors such as keyboard interrupts
  - hardware malfunctions
  - disk failure and so on

The exception handling mechanism of C++ is designed to **handle only synchronous exceptions** within a program.

Exception levels

Exceptions can occur at many levels:

1. **Hardware/operating system level**.
   - Arithmetic exceptions; divide by 0.
   - Memory access violations; stack over/underflow.
2. **Language level**.
   - Type conversion; illegal values, improper casts.
   - Bounds violations; illegal array indices.
   - Bad references; null pointers.
3. **Program level**.
   - User defined exceptions.

Need of Exceptions

- Detect and report an “exceptional circumstance”
- Separation of error handling code from normal code
- That is, you will isolate your error handling code from your ordinary code. The code will be more coherent and simpler to keep up with.
- Functions/ Methods can handle any exception they choose
- Grouping of Error types

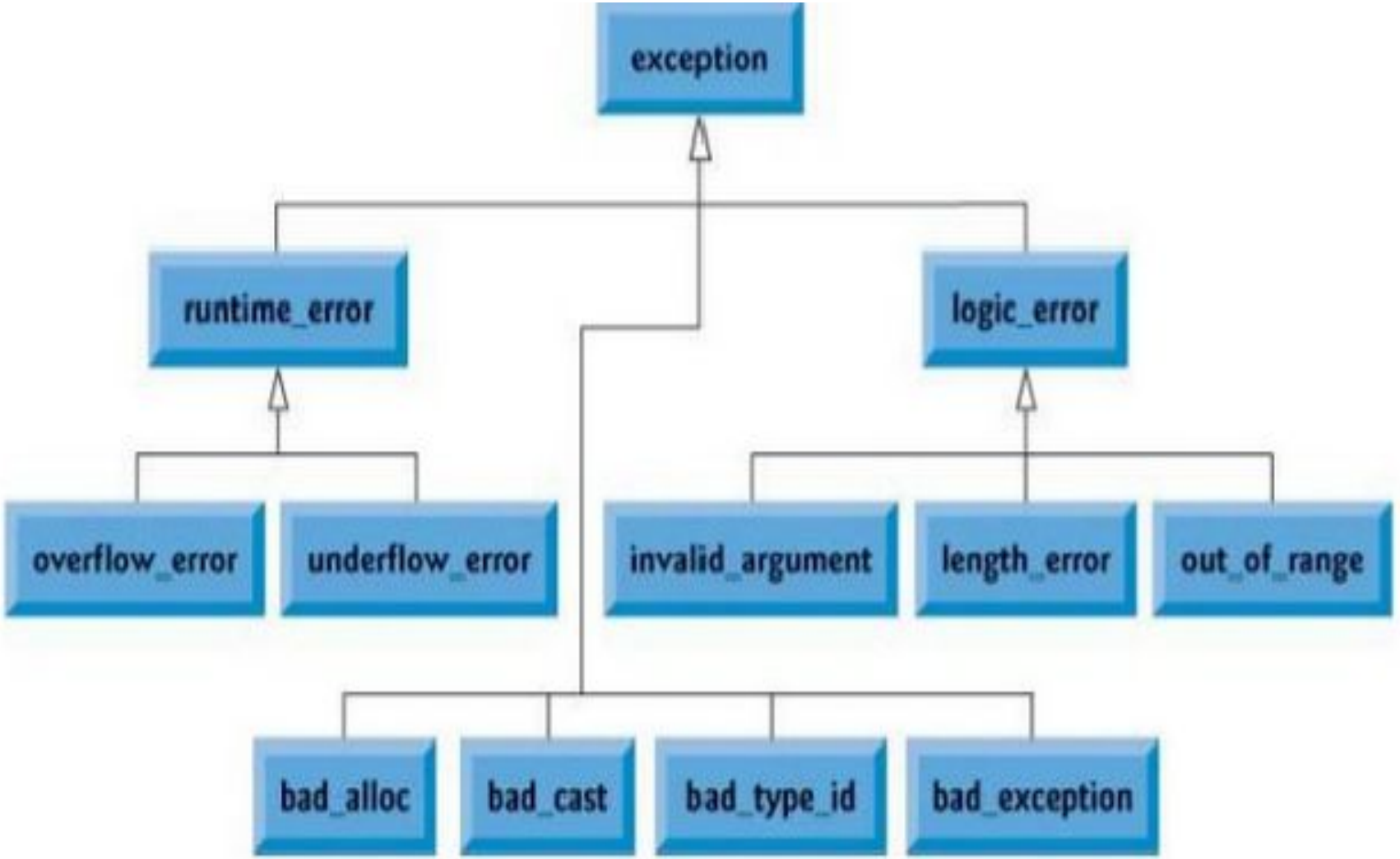
Basic Keywords in Exception Handling

- Exception Handling in C++ falls around these three keywords:
  - **Throw:** when a program experiences an issue, it throws an Exception. The throw keyword assists the program by performing throw
  - **Catch:** a program that utilises an exception handler to catch an Exception. It is added to the part of a program where you need to deal with the error
  - **Try:** the try block recognises the code block for which certain exceptions will be enacted. It ought to be followed by one/more catch blocks

Exception Handling Mechanism

1. Find the problem (*Hit* the exception)
2. Inform that an error has occurred (*Throw* the exception)
3. Receive the error information (*Catch* the exception)
4. Take corrective actions (*Handle* the exception)

C++ Standard Exceptions



C++ Standard Exceptions

Exception	Description
std::exception	An exception and parent class of all the standard C++ exceptions.
std::bad_alloc	This can be thrown by **new**.
std::bad_cast	This can be thrown by **dynamic_cast**.
std::bad_exception	This is useful device to handle unexpected exceptions in a C++ program
std::bad_typeid	This can be thrown by **typeid**.

C++ Standard Exceptions

Exception	**Description**
std::logic_error	An exception that theoretically can be detected by reading the code.
std::domain_error	This is an exception thrown when a mathematically invalid domain is used
std::invalid_argument	This is thrown due to invalid arguments.
std::length_error	This is thrown when a too big std::string is created
std::out_of_range	This can be thrown by the at method from for example a std::vector and std::bitset<>::operator[]().

C++ Standard Exceptions

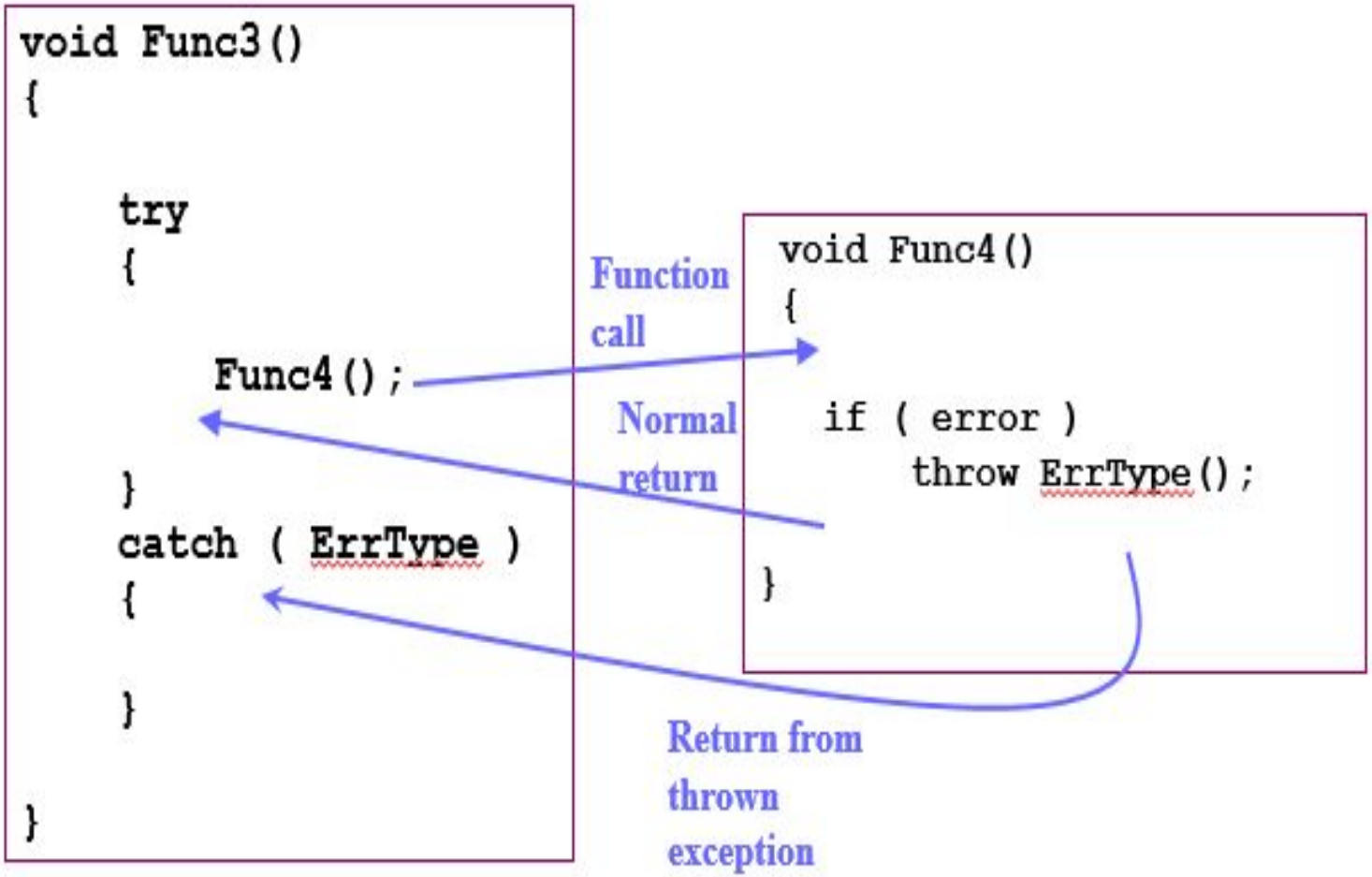
Exception	Description
std::runtime_error	An exception that theoretically can not be detected by reading the code.
std::overflow_error	This is thrown if a mathematical overflow occurs.
std::range_error	This is occured when you try to store a value which is out of range.
std::underflow_error	This is thrown if a mathematical underflow occurs.

Exceptions : syntax

```
try {
   // the protected code
} catch( Exception_Name exception1 ) {
   // catch block
} catch( Exception_Name exception2 ) {
   // catch block
} catch( Exception_Name exceptionN ) {
   // catch block
}
```

- We have one **try** statement with **many catch** statement.
- The '**ExceptionName**' is the name of the Exception for being caught.
- The **exception1, exception2, exception3 and exceptionN** are your defined names for referring to the exceptions.

Exceptions


void Func3()
{
try
{
Func4();
}
catch (ErrType)
{
}
}
Function call
Normal return
void Func4()
{
if (error)
throw ErrType();
}
Return from thrown exception

Simple Exceptions : Example

```
#include<iostream>
using namespace std;
 int main()
{
int a,b;
cin >> a>> b;
try
{
   if (b!=0)
{
  cout<<“result (a/b)=”<<a/b;
 }
else
     {
     throw(b);
     }
}
catch(int i)
{
     cout <<“exception
                         caught”;
}
}
```

Nested try blocks


```
try
{  ……..
     try
     {
          …….
     }
     catch (type arg)
     {
     …….
     }
}
catch(type arg)
{    …….
     ……..
}
```

Multiple Catch Exception

- Used when a user wants to handle different exceptions differently.
- For this, a user must include catch statements with different declaration.

Multiple Catch Exception

- It is possible to design a separate catch block for each kind of exception
- Single catch statement that catches all kind of exceptions
- Syntax

```
catch(…)
{
……..
}
```

Note :

A better way to use this as a default statement along with other catch statement so that it can catch all those exception which are not handle by other catch statement

Multiple catch statement : Syntax


```
try {
……..
}
catch (type1 arg) {
……..
}
catch (type2 arg) {
      ……..
}
……..
catch(typeN arg) {
      ……..
}
```

Multiple Exceptions : Example

```
#include<iostream>
using namespace std;
 int main() {
      int a,b;
      cin >> a>> b;
      try {
      if (b!=a)
      {
      float div = (float) a/b;
      if(div <0)
        throw 'e';
      cout<<div;
      }
      else
      throw b;
      }
      catch(int i)
      {cout <<"exception caught";
      }
      catch(int i)
      {
       cout <<"exception caught :
Division by zero";
      }
      catch (char st)
      {
      cout << "exception caught :
Division is less than 1";
      }
      catch(…)
      {
      cout << "Exception : unknown";
      }
      }
```

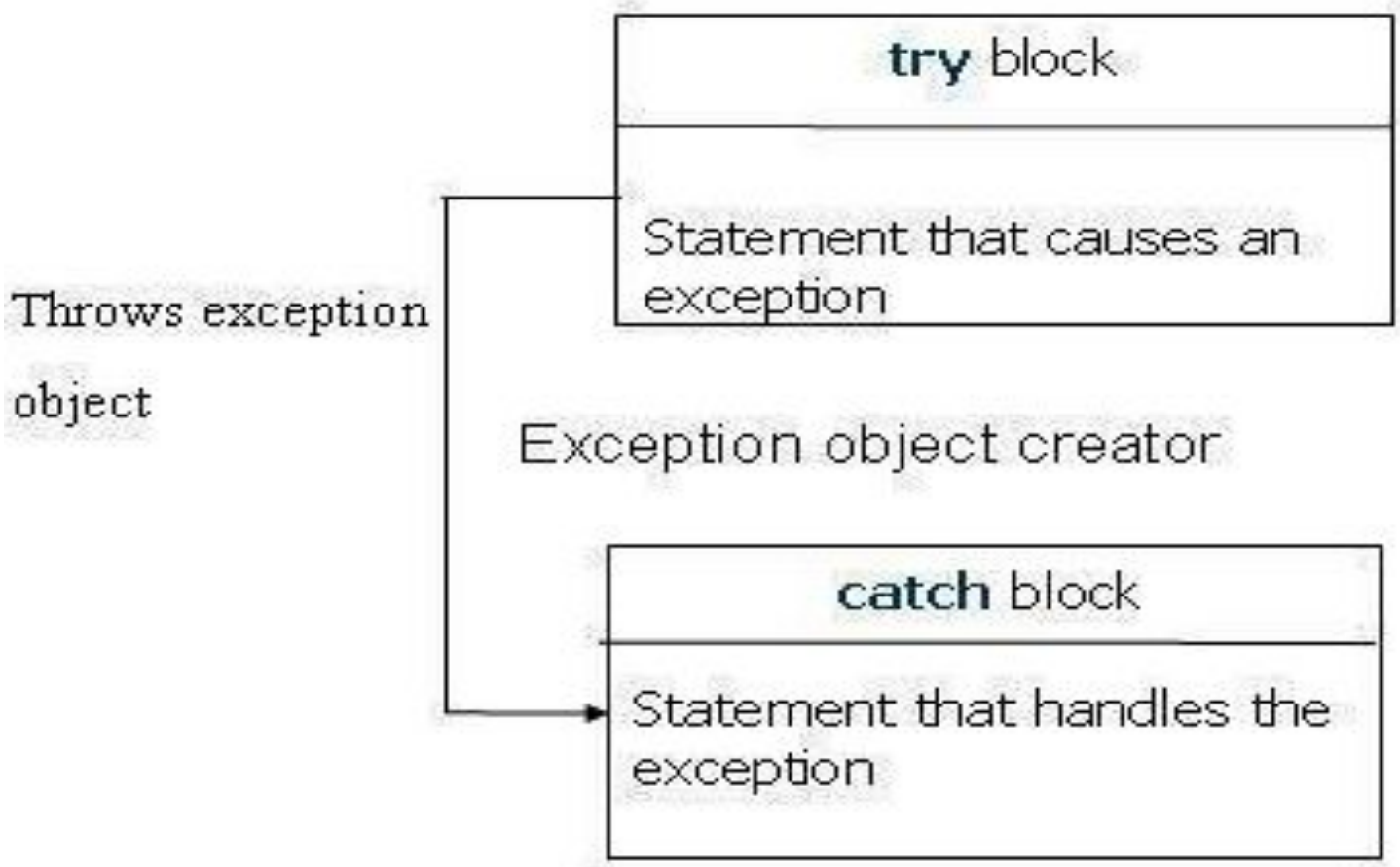
SRM INSTITUTE OF SCIENCE & TECHNOLOGY
Deemed to be University u/s 3 of UGC Act, 1956

18CSC202J - OBJECT ORIENTED DESIGN AND PROGRAMMING

Session 7

Topic :throw, throws and finally

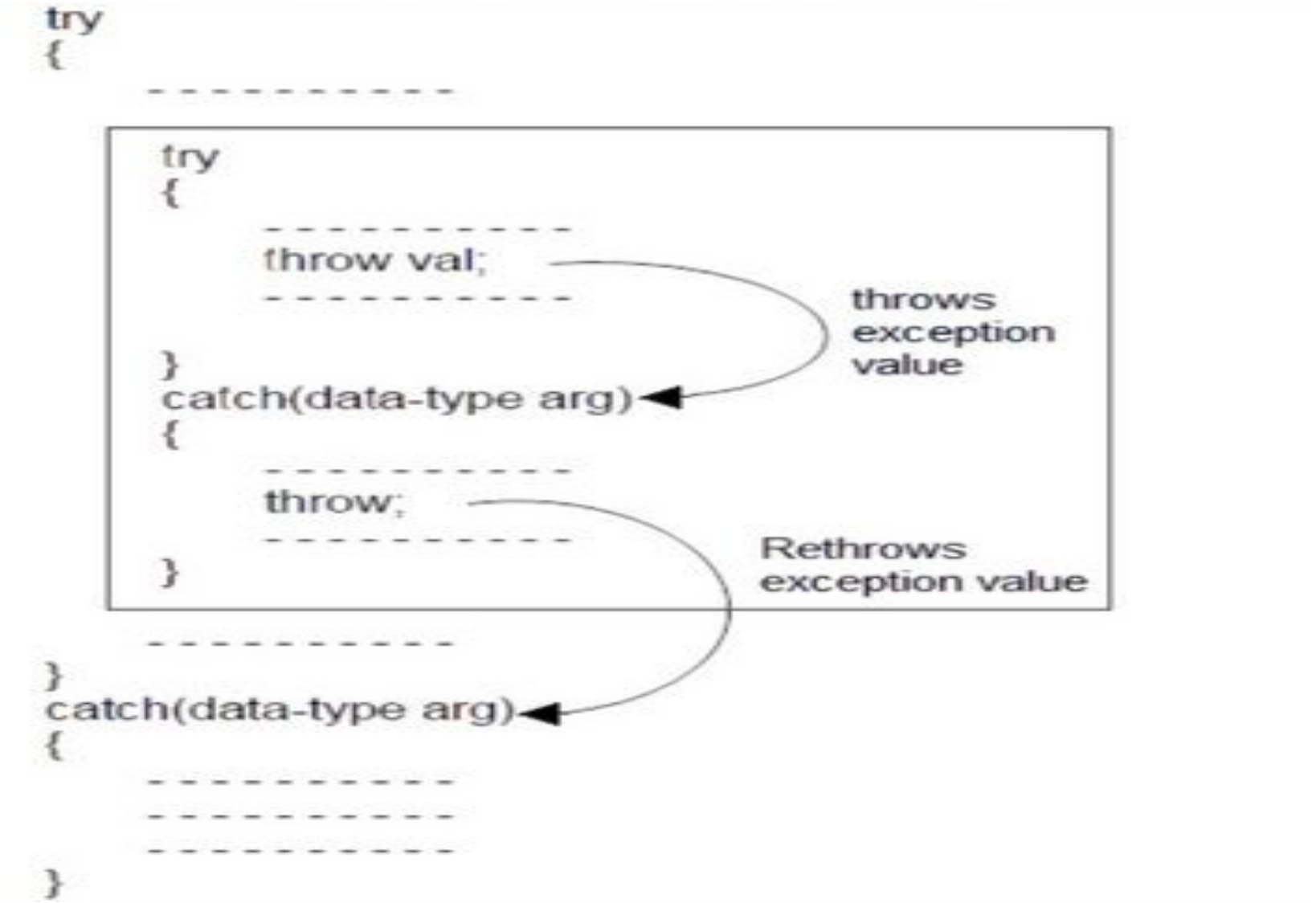
throwing Exception



throwing Exception

- When an exception is detected, it is thrown using **throw** statement in the try block
- It is also possible to have nested try-catch statement
- It cause the current exception to be thrown to the **next** enclosing try/catch sequence and is caught by a catch statement listed after that enclosing try block.

Re-throwing Exception



Throw Example


```
try
{
    if(denominator == 0)
    {
        throw denominator;
    }
    result = numerator/denominator;
    cout<<"\nThe result of division is:" <<result;
}
```

Handle Any Type of Exceptions (...)

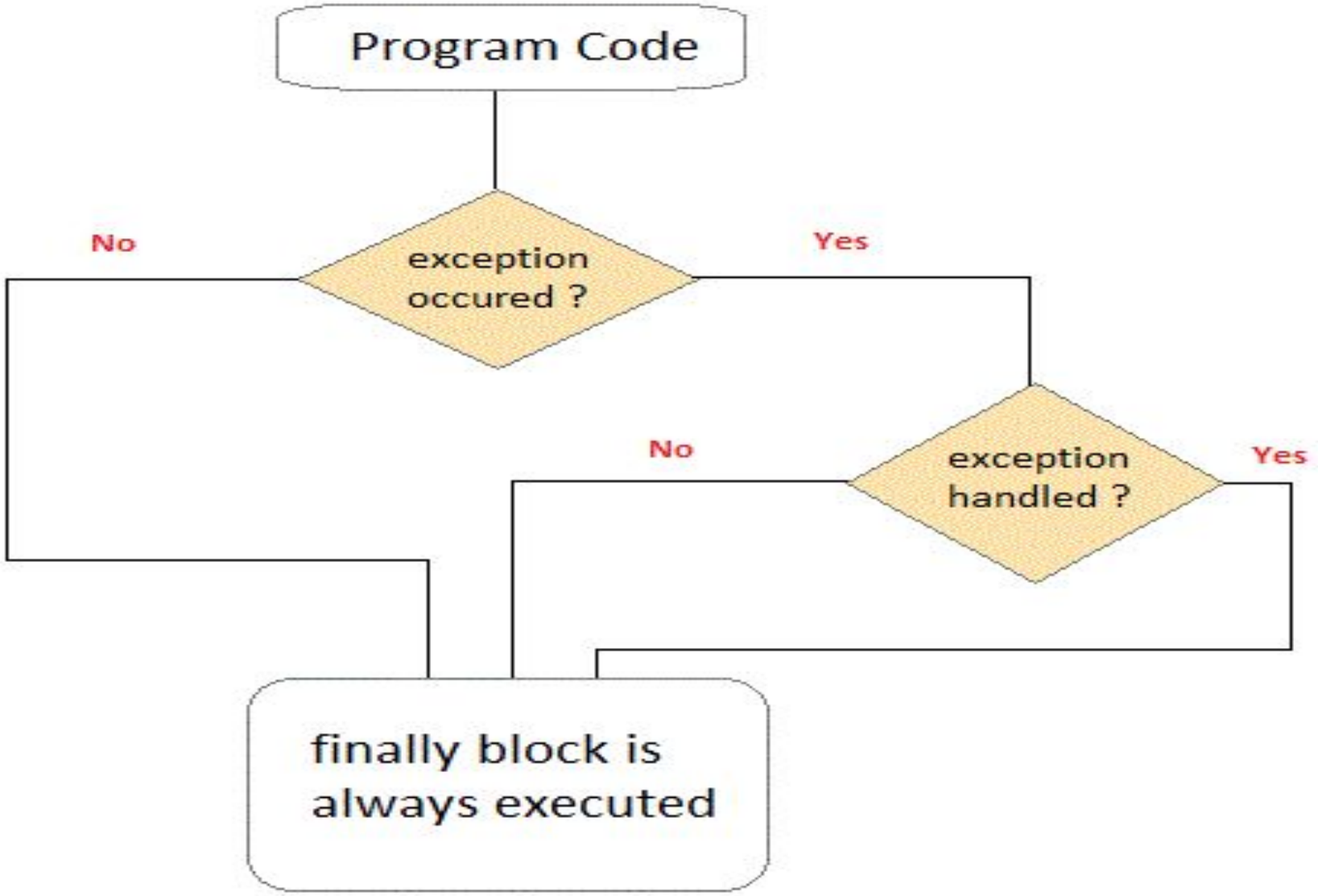
- If you do not know the throw **type** used in the try block, you can use the "**three dots**" syntax (...) inside the catch block, which will handle any type of exception.

```
try {
  int age = 15;
  if (age > 18) {
    cout << "Access granted - you are old enough.";
  } else {
    throw 505;
  }
}
catch (...) {
  cout << "Access denied - You must be at least 18 years old.\n";
}
```

Finally

- The application always executes any statements in the finally part, even if an exception occurs in the try block.
- When any code in the **try** block raises an exception, execution halts at that point.
- Once an exception handler is found, execution jumps to the finally part.
- After the **finally** part executes, the exception handler is called.
- If no exception occurs, the code in the **finally** block executes in the normal order, after all the statements in the **try** block.

Finally


Program Code
exception occured ?
No
Yes
exception handled ?
No
Yes
finally block is always executed

Finally - Syntax


```
try
{
// statements that may raise an exception
 }
 __finally
 {
 // statements that are called even
//if there is an exception in the try block
 }
```

Finally - Example

```
#include<iostream>
using namespace std;
 int main()
{
int a,b;
cin >> a>> b;
try{
    if (b!=0) {
     cout<<“result
                (a/b)=”<<a/b;
    }
Else {
     throw(b);
       }
}
catch(int i) {
   cout <<“exception caught”;
}
__finally {
        cout<<“Division”;
             }
}
```

SRM INSTITUTE OF SCIENCE & TECHNOLOGY
Deemed to be University u/s 3 of UGC Act, 1956

Session 8

Topic : Exceptional Handling: User defined exception

User Defined Exceptions

- We can define your own exceptions by inheriting and overriding exception class functionality.
- User defined exception classes inherit exception class provided by C++ and overrides it's functionality according to our needs.
- To use class exception, we must include exception header using the pre-processor directive.

#include <exception>

Rules for User Defined Exceptions

- Always include exception header using pre-processor directive at the very first step.
- The function which will return an exception string should have a return type of **char followed by *,**

```
char* what()
{
// codes here
}
```

char is as return type because we will return a string

- Should have a try and catch block.

User Defined Exceptions- Example


```
class MyCustomException : public std::exception {
   public:
char * what () {
      return "Custom C++ Exception";
   }
};

int main() {
   try {
      throw MyCustomException();
   } catch (MyCustomException mce) {
      cout << "Caught MyCustomException" << endl;
      cout << mce.what();
   }
}
```

User Defined Exceptions

Example

- Let's say that the password must consists of at least 6 characters.
- If we write a exception for this case, when the program receives a password in length of 5 characters it will throw an exception so that we could know the password is not valid

User Defined Exceptions

Example

```
class BadLengthException : public exception {
   public:
      int N;
      BadLengthException(N) {
         this->N=N;
      };
      int what() {
         return this->N;
      }
}
```

User Defined Exceptions

- The BadLengthException inherited all properties from the exception class.
- When this class is initialized, it takes the username length and stores it in public variable N.
- When the catch block detected exception, it will dial with the function **what of our exception class** to get **what is happened**.

User Defined Exceptions


```
int main() {
   int usernameLength;
   cin>>usernameLength;
   try {
     if(usernameLength<5)
        throw BadLengthException(usernameLength);
     else
        cout<<"Valid";
   } catch(BadLengthException e) {
      cout<<"Too short: "<<e.what();
   }
   return 0;
}
```

Passing Parameters to Custom Exceptions

- Custom exceptions can include parameters to provide relevant information about the exception and customize the error message. This can help programmers to better handle the exception.

```
class MyCustomException : public std::exception {
   private:
   char * message;
   public:
   MyCustomException(char * msg) : message(msg) {}
   char * what () {
      return message;    }
};
```

Passing Parameters to Custom Exceptions


```
int main() {
    try {
        throw MyCustomException("Custom C++
                                                Exception");
    } catch (MyCustomException mce) {
        cout << "Caught MyCustomException" << endl;
        cout << mce.what();
    }
}
```

Output

```
Caught MyCustomException
Custom C++ Exception
```

Questions in Exceptions

What is the advantage of exception handling ?

1. Remove error-handling code from the software's main line of code.
2. A method writer can choose to handle certain exceptions and delegate others to the caller.
3. An exception that occurs in a function can be handled anywhere in the function call stack.

(A) Only 1

(B) 1, 2 and 3

(C) 1 and 3

(D) 1 and 2

Questions in Exceptions

Predict the output of the code?

```
class Base {};
class Derived: public Base {};
int main() {
  Derived d;
  try {
     throw d;
  }  catch(Base b) {
      cout<<"Caught Base Exception";
  }  catch(Derived d) {
      cout<<"Caught Derived Exception";
  }
  return 0;
}
```

(A) Caught Derived Exception
(B) **Caught Base Exception**
(C) Compiler Error
(D) Run Time Error

Questions in Exceptions

Predict the output of the code?

```
int main()
{
   try
   {
     throw 'a';
   }
   catch (int param)
   {
      cout << "int exception\n";
   }
   catch (...)
   {
      cout << "default exception\n";
   }
   cout << "After Exception";
   return 0;
}
```

(A) default exception
After Exception

(B) int exception
After Exception

(C) int exception

(D) default exception

Questions in Exceptions

Predict the output of the code?

```
int main()
{ try {
      throw 10;
   }
   catch (...)   {
      cout << "default
exception\n";
   }
   catch (int param)  {
      cout << "int exception\n";
   }
    return 0;
}
```

((A) default exception
(B) int exception
(C) Compiler Error
(D) Run Time Error

Reason: The catch(…) must be the last catch block.

Questions in Exceptions

Predict the output of the code?

```
int main() {
    try {
        try {
            throw 20;
        }
        catch (int n) {
            cout << "Inner Catch\n";
            throw; }
    }
    catch (int x){
        cout << "Outer Catch\n";
    }
    return 0;
}
```

((A) Outer Catch
(B) Inner Catch
(C) Inner Catch
Outer Catch
(D) Compiler Error

Reason: The statement ‘throw;’ is used to re-throw an exception.

Questions in Exceptions

Which of the following is true about exception handling in C++?

1) There is a standard exception class like Exception class in Java.

2) All exceptions are unchecked in C++, i.e., compiler doesn't check if the exceptions are caught or not.

3) In C++, a function can specify the list of exceptions that it can throw using comma separated list like following.

void fun(int a, char b) throw (Exception1, Exception2, ..)

(A) 1 and 3

(B) 1, 2 and 3

(C) 1 and 2

(D) 2 and 3

SRM
INSTITUTE OF SCIENCE & TECHNOLOGY
Deemed to be University u/s 3 of UGC Act, 1956

Package and Component Diagram

Dynamic Modelling

The dynamic model is used to express and model the behaviour of the system over time. It includes support for activity diagrams, state diagrams, sequence diagrams and extensions including business process modelling.

SRM
INSTITUTE OF SCIENCE & TECHNOLOGY
Deemed to be University u/s 3 of UGC Act, 1956

Package Diagram

- All the interrelated classes and interfaces of the system when grouped together form a package.
- To represent all these interrelated classes and interface UML provides package diagram.
- Package diagram helps in representing the various packages of a software system and the dependencies between them.
- It also gives a high-level impression of use case and class diagram.

SRM
INSTITUTE OF SCIENCE & TECHNOLOGY
Deemed to be University u/s 3 of UGC Act, 1956

Package Diagram: purpose

- To provide static models of modules, their parts and their relationships
- To present the architectural modelling of the system
- To group any UML elements
- To specify the logical distribution of classes
- To emphasize the logical structure of the system
- To offer the logical distribution of classes which is inferred from the logical architecture of the system

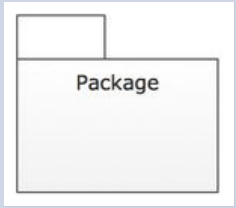
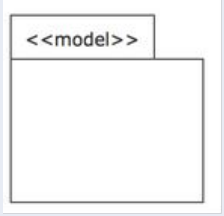
SRM
INSTITUTE OF SCIENCE & TECHNOLOGY
Deemed to be University u/s 3 of UGC Act, 1956

Package Diagram: Uses

- To illustrate the functionality of a software system.
- To illustrate the layered architecture of a software system.
- The dependencies between these packages can be adorned with labels / stereotypes to indicate the communication mechanism between the layers.

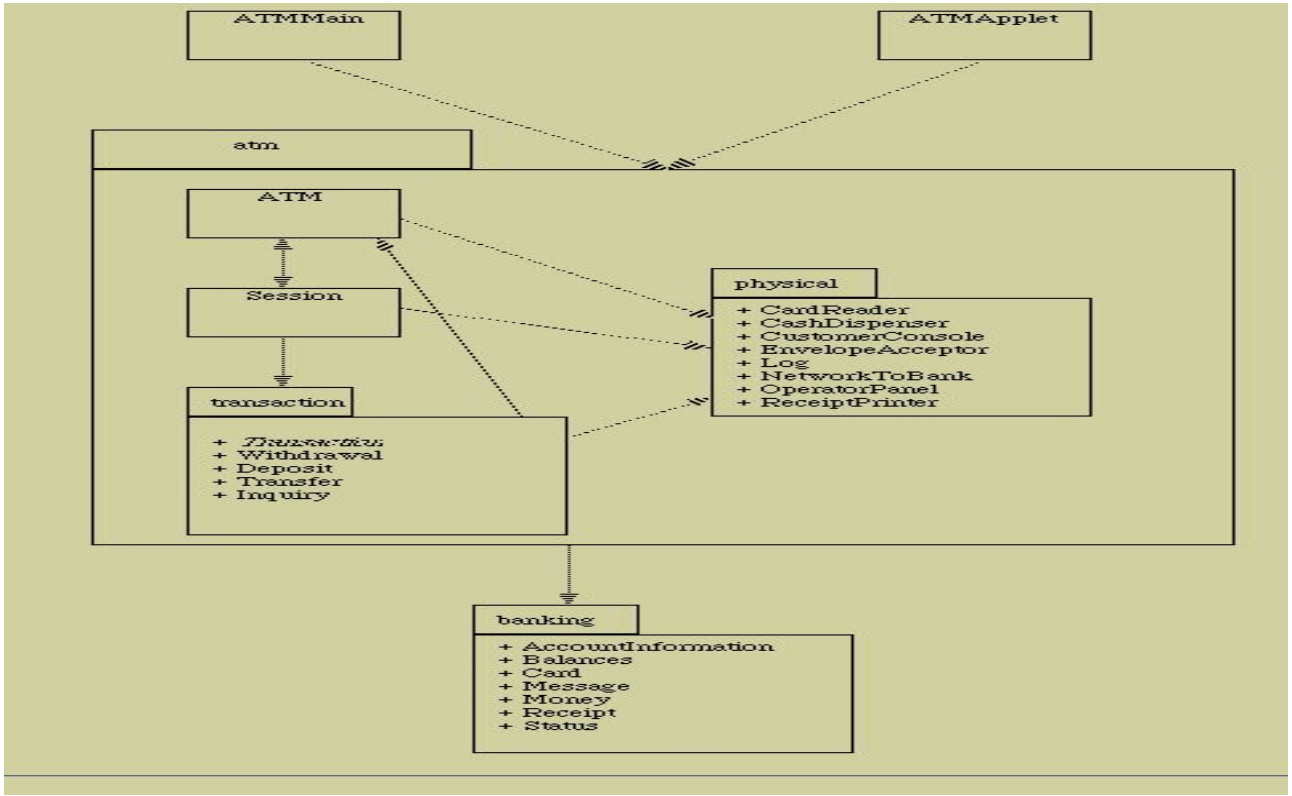
SRM INSTITUTE OF SCIENCE & TECHNOLOGY
Deemed to be University u/s 3 of UGC Act, 1956

Notations

S.NO	NAME	SYMBOL	DESCRIPTION
1	Package		organize elements into groups to provide better structure for system model.
2	Mode		show only a subset of the contained elements according to some criterion.

SRM INSTITUTE OF SCIENCE & TECHNOLOGY
Deemed to be University u/s 3 of UGC Act, 1956

Example



SRM
INSTITUTE OF SCIENCE & TECHNOLOGY
Deemed to be University u/s 3 of UGC Act, 1956

18CSC202J - OBJECT ORIENTED DESIGN AND PROGRAMMING

Session 12

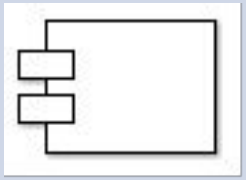
Topic : UML Component Diagram

SRM INSTITUTE OF SCIENCE & TECHNOLOGY
Deemed to be University u/s 3 of UGC Act, 1956

Guidelines to Draw: Component Diagram

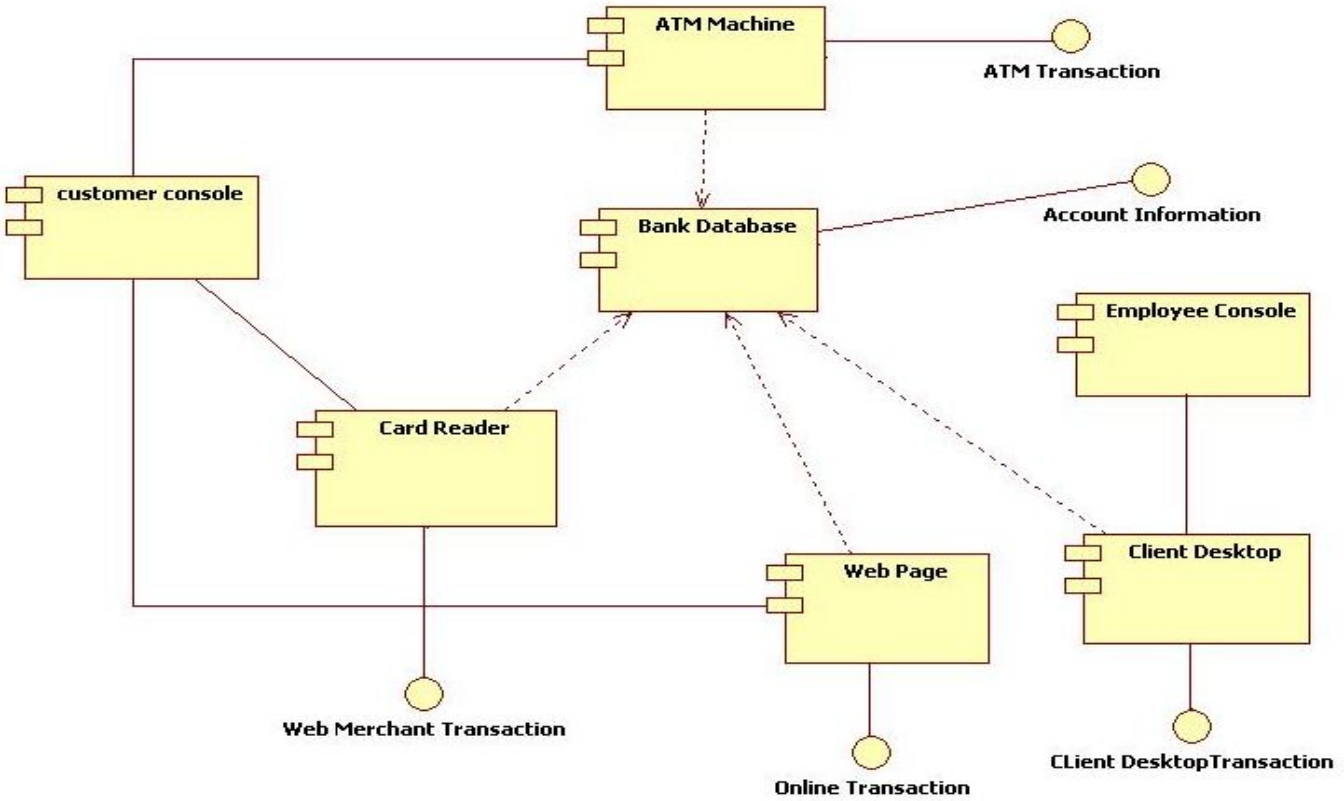
- Based on the analysis of the problem description of the system, identify the major subsystem.
- Group the individual packages and other logical entities in the system to provide as separate components.
- Then identify the interfaces needed for components interaction.
- If needed, identify the subprograms which are part of each of the components and draw them along with their associated components.
- Use appropriate notations to draw the complete component diagram.

Notations

S.NO	NAME	SYMBOL	DESCRIPTION
1	Component		Component is used to represent any part of a system for which UML diagrams are made.
2	Association		A structural relationship describing a set of links connected between objects.

SRM
INSTITUTE OF SCIENCE & TECHNOLOGY
Deemed to be University u/s 3 of UGC Act, 1956

Example



SRM
INSTITUTE OF SCIENCE & TECHNOLOGY
Deemed to be University u/s 3 of UGC Act, 1956

MCQs

Which of the following diagram displays the structural relationship of components of a software?

(A). Component Diagram
(B). class diagram
(C). use case diagram
(D). sequence diagram
Answer: Component Diagram

Which of the following diagram is required when working with big and complex systems with many components?

(A). Component Diagram
(B). class diagram
(C). use case diagram
(D). sequence diagram
MCQ Answer: Component Diagram

SRM
INSTITUTE OF SCIENCE & TECHNOLOGY
Deemed to be University u/s 3 of UGC Act, 1956

MCQs

In Component Diagram, Components communicate with each other using which of the following?
(A). Components
(B). interfaces
(C). Use cases
(D). attributes
MCQ Answer: interfaces

The interfaces in component diagrams are linked using which of the following?
(A). connectors
(B). interfaces
(C). Components
(D). None of these
MCQ Answer: connectors

MCQs

A package diagram consists of the following?

(A). Package symbols
(B). Groupings of Use cases, classes, components
(C). Interface
(D). Package symbols, Groupings of Use cases, classes & components
MCQ Answer: Package symbols, Groupings of Use cases, classes & components

Which one of the following is not a structural thing?

(A). Class
(B). Package
(C). Use case
(D). Node
MCQ Answer: Package

SRM
INSTITUTE OF SCIENCE & TECHNOLOGY
Deemed to be University u/s 3 of UGC Act, 1956

18CSC202J - OBJECT ORIENTED DESIGN AND PROGRAMMING

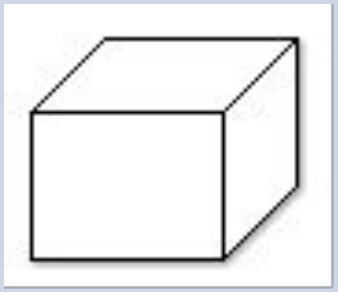
Session 13

Topic : UML Deployment Diagram, Examples

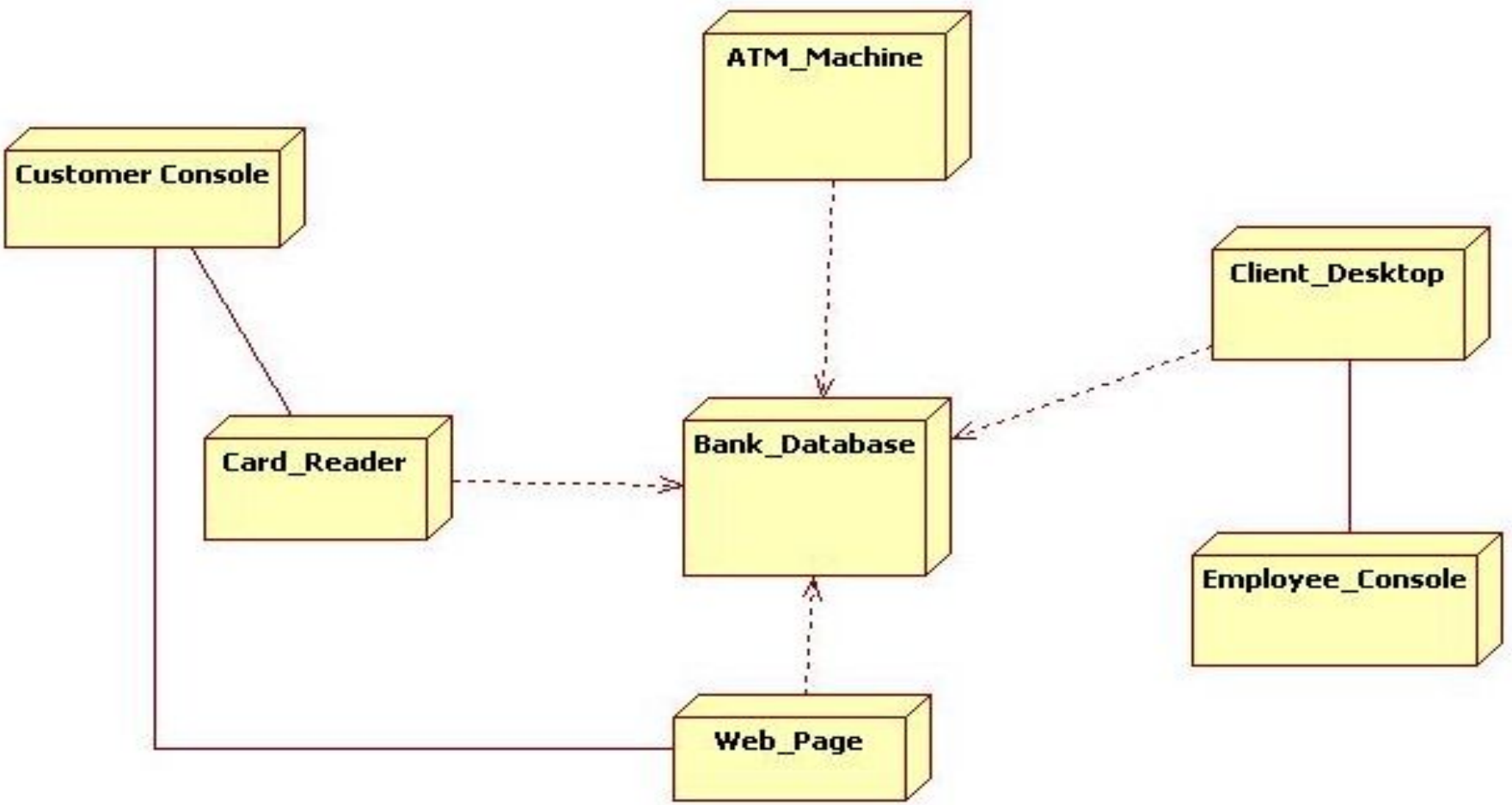
Guidelines to Draw: Deployment Diagram

- Identify the hardware components and processing units in the target system.
- Analyze the software and find out the subsystem, parallel execution of modules, server side components, client side components, business logic components, backend database servers and software and hardware mapping mechanism to map the software components to be mapped with appropriate hardware devices.
- Draw the hardware components and show the software components inside them and also show the connectivity between them.

Notations

S.NO	NAME	SYMBOL	DESCRIPTION
1	Node		A node represents a physical component of the system. Node is used to represent physical part of a system like server, network etc.
2	Association		A structural relationship describing a set of links connected between objects.

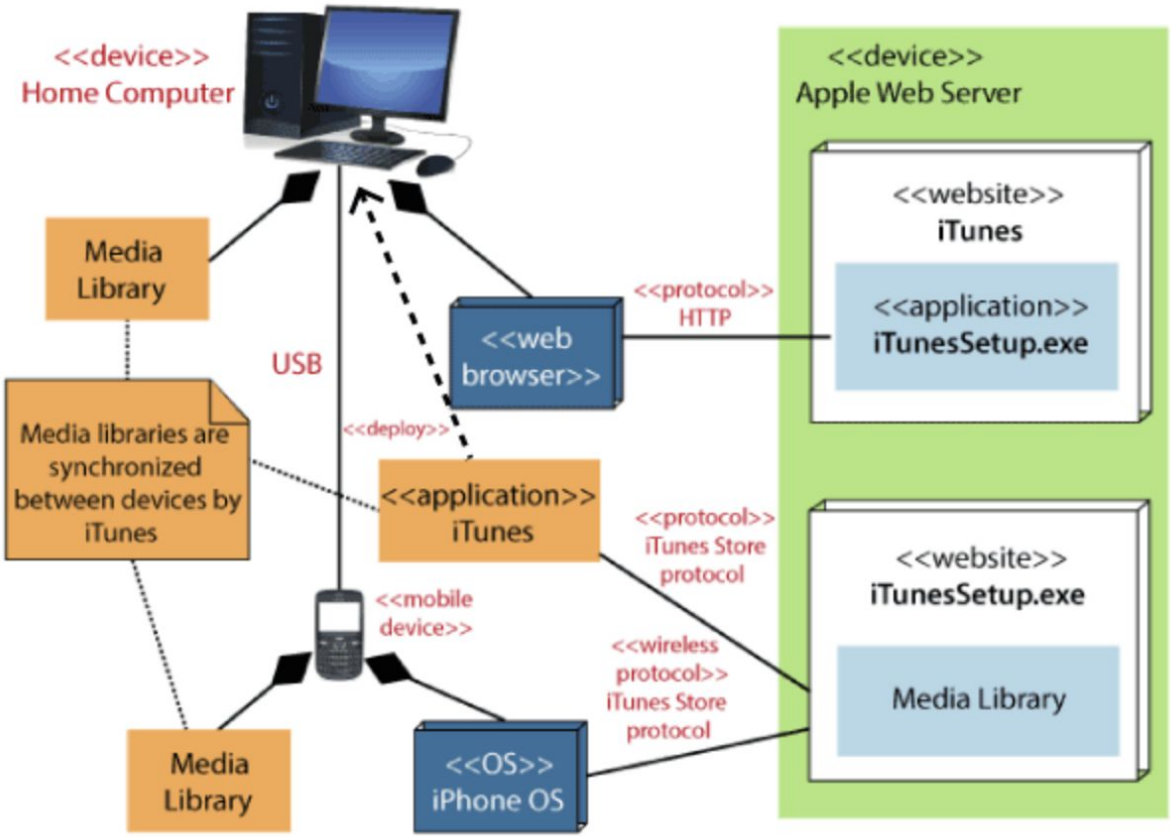
Example


ATM_Machine
Customer Console
Client_Desktop
Card_Reader
Bank_Database
Employee_Console
Web_Page

Example

1. online shopping UML diagrams
2. Ticket vending machine UML diagrams
3. Bank ATM UML diagrams
4. Hospital management UML diagrams
5. Digital imaging and communications in medicine (DICOM) UML diagrams
6. Java technology UML diagrams
7. Application development for Android UML diagrams

Deployment Diagram



Applications of Deployment Diagram

- To model the network and hardware topology of a system
- To model the distributed networks and systems
- Implement forwarding and reverse engineering processes
- To model the hardware details for a client/server system
- For modelling the embedded system