21CSC101T

Object Oriented Design and Programming

UNIT-4

- Generic - Templates - Function templates - Class Templates

- Exceptional Handling: try and catch - Multilevel exceptional - throw and throws - finally - User defined exceptional

- Dynamic Modeling: Package Diagram - UML Component Diagram - UML Deployment Diagram

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.

Topic :Example Program Function Template, Class Template

Defining a Function Template

A function template starts with the keyword template followed by template parameter(s) inside <> which is followed by the function definition.

```
template <typename T>
T functionName(T parameter1, T parameter2, ...)
{
  // code
}
```

Invoking function

functionName<dataType>(parameter1, parameter2,...);

Method Overloading

```cpp
#include <iostream>
using namespace std;
class sample
{
    public:
    void add(int x, int y)
    {
        cout<<"Addint="<<x+y<<endl;

    }
    void add(double a, double b)
    {
        cout<<"AddFloat="<<a+b;
    }
};
int main()
{
    sample s1;
    s1.add(2, 3);
    s1.add(2.2, 3.3);
    return 0;
}
```

Template Function

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

template <typename T>
T add(T num1, T num2)
{
    return (num1 + num2);
}


int main()
{
cout << "Result1=" <<add(2,3)<< endl;
cout << "Result2=" <<add(2.3, 3.3)<< endl;
return 0;
}
```

```
Result1=5
Result2=5.6
```

Calling a Function Template

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

template <typename T>
T add(T num1, T num2)
{
    return (num1 + num2);
}
```

```cpp
int main()
{
int result1;
double result2;
// calling with int parameters
result1 = add<int>(2, 3);
cout << "Result1=" << endl;

// calling with double parameters
result2 = add<double>(2.2, 3.3);
 cout << "Result2=" << endl;

return 0;
}
```

```
Result1=5
Result2=5.5
```

Example program Function Templates

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

template <typename T>
T Sub(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 = Sub(A,B);
    cout<<"The sub of integer values : "<<C<<endl;
    K = Sub(I,J);
    cout<<"The sub of long values : "<<K;
}
```

```
The sub of integer values : -10
The sub of long values : -11
```

More than One Template Argument

```cpp
template<class T , class U>
void multiply(T p , U q)
{
    cout<<"Multiplication= "<<p*q<<endl;
}
int main()
{
    int a, b;
    float x, y;
   cout<<"Enter the value of a and b"<<endl;
    cin>>a>>b;
    cout<<"Enter the value of x and y"<<endl;
    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;
}
```

```
Enter the value of a and b
2 3
Enter the value of x and y
2.6 3.6
Multiplication= 6
Multiplication= 9.36
Multiplication= 5.2
```

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 T>
class className
{
private:
      T var; ... .. ...
public:
  T functionName(T arg);
... .. ...
};
```

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

Syntax: className <dataType> classObject;

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

```cpp
#include <iostream>
using namespace std;
// Class template

template <class T>
class Number
{
  private:
    T num;

  public:
    Number(T n)
    {
      num=n;
    }    // constructor

    T getNum()
    {
        return num;
    }

};
int main()
{
    Number <int> n1(7);
    Number <double> n2(7.7);
    cout << "int Number = " << n1.getNum() << endl;
    cout << "double Number = " << n2.getNum() << endl;
    return 0;

}
```

```
int Number = 7
double Number = 7.7
```

```cpp
// Class template
template <class T>
class Number
{
  private:
      T num;

  public:
   Number(T n) : num(n)
   {

   }

   T getNum()
   {
      return num;
   }
};
```

```cpp
int main()
{
Number <int> n1(7);
Number <double> n2(7.7);
cout << "int Number = " << n1.getNum() << endl;
cout << "double Number = " << n2.getNum() << endl;
   return 0;
}
```

```
int Number = 7
double Number = 7.7
```

```cpp
#include<iostream>
using namespace std;
template <class T1, class T2>
class Number
{
    private:
            T1 num1;
            T2 num2;

    public:
     Number(T1 a, T2 b)
     {
     num1=a;
     num2=b;
     }
     T1 getNum()
     {
         return num1*num2;
     }
};

int main()
{
Number <int, double> n1(2,3.7);
Number <double, int > n2(2.2, 3);
cout << "int Number = " << n1.getNum() << endl;
cout << "double Number = " << n2.getNum() << endl;
return 0;
}
```

```
int Number = 7
double Number = 6.6
```

Example Program

```cpp
#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.

```cpp
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;
}
```

St[2]	3333.3F
St[1]	2222.2F
St[0]	1111.1F

```
1:  3333.3
2:  2222.2
3:  1111.1
1:  345345345
2:  234234234
3:  123123123
```

1)Which of the following is used for generic programming?
a) Virtual functions
b) Modules
c) Templates
d) Abstract Classes

2)Which of the following is correct about templates?
a) It is a type of compile time polymorphism
b) It allows the programmer to write one code for all data types
c) Helps in generic programming
d) All of the mentioned

3)Which among the following is the proper syntax for the template class?
a) template <typename T1, typename T2>;
b) Template <typename T1, typename T2>;
c) template <typename T> T named(T x, T y){ }
d) Template <typename T1, typename T2> T1 named(T1 x, T2 y){ }

1) Find the Output

```cpp
#include <iostream>
using namespace std;
template <typename T>
void fun(const T&x)
{
    static int count = 0;
    cout << "x = " << x << " count = " << count;
    ++count;
    return;
}
int main()
{
    fun<int> (1);
    cout << endl;
    fun<int>(1);
    cout << endl;
    fun<double>(1.1);
    cout << endl;
    return 0;
}
```

a) x = 1 count = 0 x = 1 count = 1 x = 1.1 count = 0

b) x = 1 count = 0 x = 1.0 count = 1.0 x = 1.1 count = 0

c) x = 1.0 count = 0.0 x = 1 count = 1 x = 1.1 count = 0

d) x =1.0 count = 0.0 x = 1.0 count =1.0 x = 1.1 count = 0.0

What will be the output of the following C++ function?

```cpp
#include <iostream>
using namespace std;
template <typename T1, typename T2>
T1 max1(T1 x, T2 y)
{
    return (x > y)? x : y;
}
int main()
{
    cout << max1(3, 7) << endl;
    cout << max1(3.2, 7) << endl;
    cout << max1(3.1, 7.2) << endl;
    return 0;
}
```

a) 7   7.0   7.0
b) 7   7   7.2
c) 7   7.0   7.0
d) error

```cpp
#include <iostream>
using namespace std;
template <class T>
class Test
{
  private:
    T val;
  public:
    static int count;
    Test()  {
            count++;
    }};
template<class T>
int Test<T>::count = 0;
int main()
{
    Test<int> a;
    Test<int> b;
    Test<double> c;
    cout << Test<int>::count << endl;
    cout << Test<double>::count << endl;
    return 0;
}
```

```cpp
#include<iostream>
#include<stdlib.h>
using namespace std;
template<class T, class U>
class A
{
   T x;
   U y;
};
 int main()  {
  A<char, char> a;
  A<int, int> b;
  cout << sizeof(a) << endl;
  cout << sizeof(b) << endl;
  return 0;
}
```

Template Scenario1

1)Define a function template "func_swap" that swaps two values. The function takes two reference arguments of type T. It then swaps the values. As the arguments are references, whatever changes we do to arguments in the function will be reflected in the caller function.

Template Scenario2

- Write a C++ program for implementing the concept class template. Create the template class myclass. Inside this, create a constructor that will initialize the two members a and b of the class. There is another member function getMaxval which is also a function template that returns a maximum of a and b.In the main function,construct two objects, myobject of type integer and mychobject of type character. Then call the getMaxval function on each of these objects to determine maximum value.

- Note that apart from template type parameters (parameters of type T), template functions can also have ordinary parameters like normal functions and also default parameter values.

Direct Questions

1) **C++ program to determine which triangle is greater in term of area**

2) **C++ program to Display Largest Element of an array**

3) **C++ program to find Maximum or largest number in array**

4) **C++ program for ATM cash withdrawal/c++ program for ATM machine**

Topic :Exceptional Handling: try and catch, multilevel exceptional

Exceptions

- It Indicate problems that occur during a program's execution
- It is 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
- Functions/ Methods can handle any exception they choose
- Grouping of Error types

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
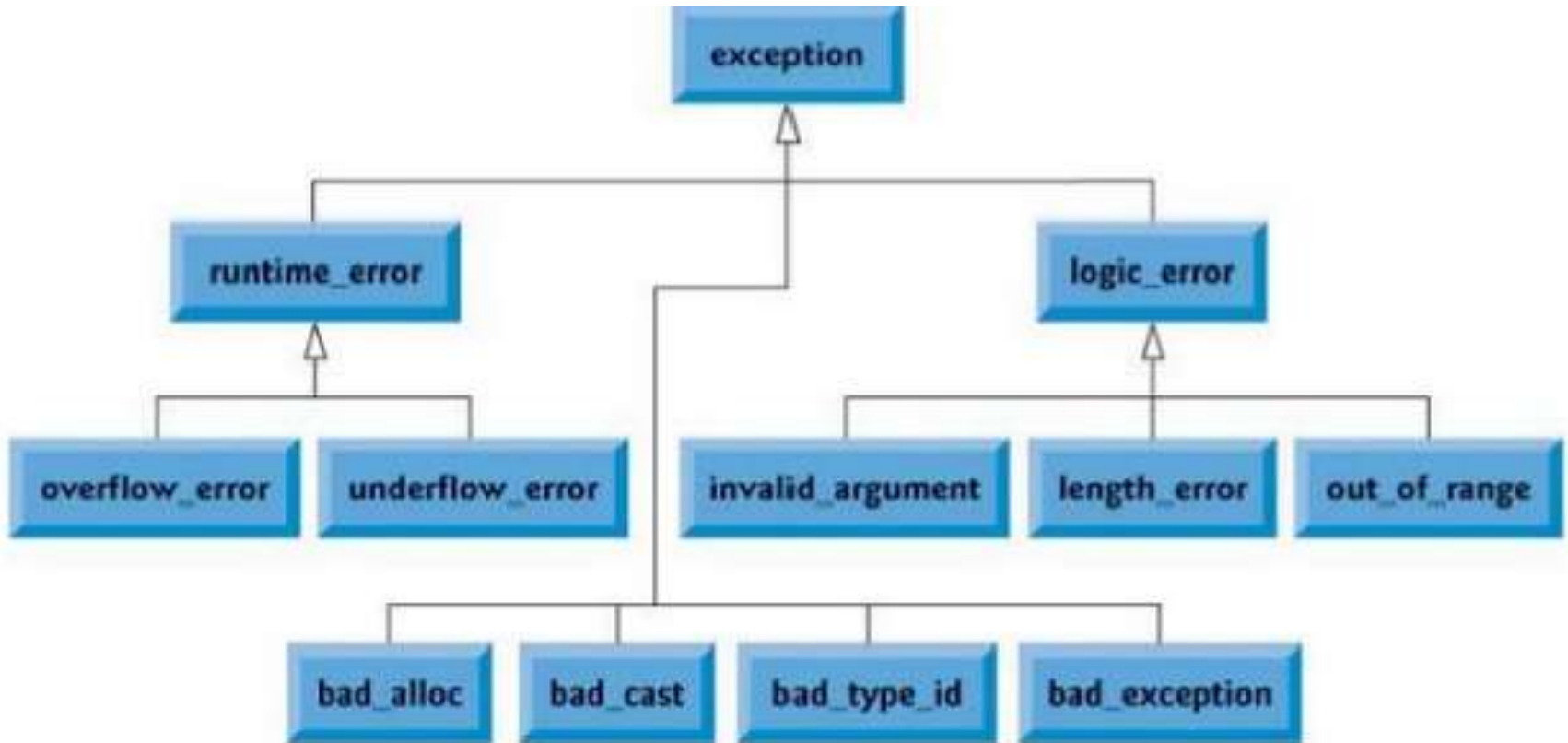
C++ Standard Exceptions

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: keywords

•Handling the Exception is nothing but converting system error message into user friendly error message
•Exception handling use three keywords for handling the exception

   •try
   •catch
   •throw

Simple Exceptions : syntax

```
........
.........
try
{
          .........
          throw exception;
          .........
          .........
}
catch(type arg)
{
          ..........
          ...........
}
.......
.......
```

Exceptions


```
void Func3()
{

    try
    {

        Func4();

    }
    catch ( ErrType )
    {

    }

}
```

Function call

Normal return

```
void Func4()
{

    if ( error )
        throw ErrType();

}
```

Return from thrown exception
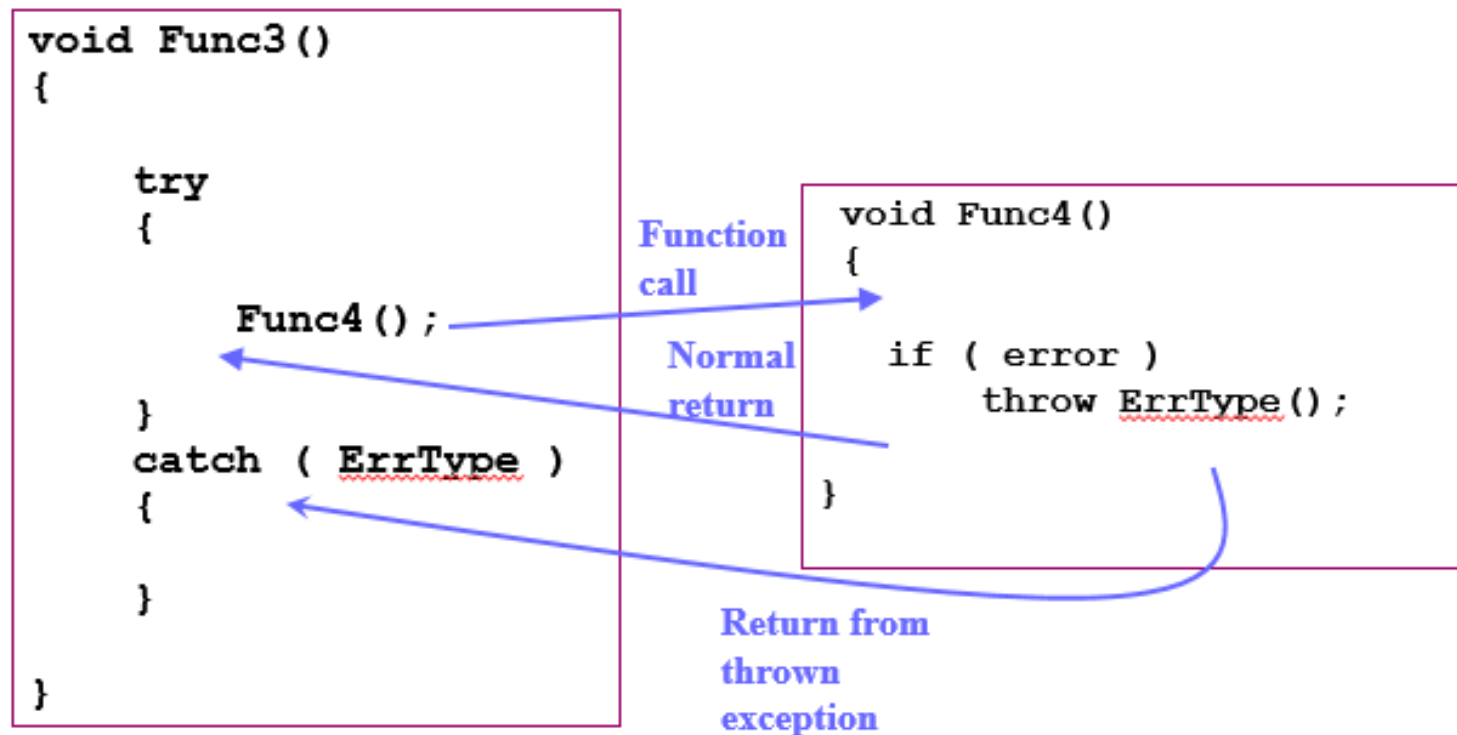
Simple Exceptions : Example

```cpp
#include<iostream>
using namespace std;
 int main()
{
   int a,b;
   cout<<"Enter the value of a and b";
   cin >> a>> b;
   try
    {
      if (b!=0)
       {
         cout<<"result (a/b)="<<a/b;
       }
```

```cpp
else
        {
        throw(b);
        }
    }
}
catch(int i)
 {
  cout <<"Attempted to divide by zero!";
 }
}
```

```
Enter the value of a and b
10 2
result (a/b)=5
```

```
Enter the value of a and b
10 0
Attempted to divide by zero!
```

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

```cpp
#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";
}
}
```

Bad Exception: Example

std::cerr:-

Standard output stream for errors
Object of class ostream that represents the standard error stream oriented to narrow characters (of type char). It corresponds to the C stream stderr.

The standard error stream is a destination of characters determined by the environment. This destination may be shared by more than one standard object (such as cout or clog).

The object is declared in header <iostream> with external linkage and static duration: it lasts the entire duration of the program.

```cpp
// bad_exception example
#include <iostream>      // std::cerr
#include <exception>     // std::bad_exception, std::set_unexpected

void myunexpected () {
  std::cerr << "unexpected handler called\n";
  throw;
}
void myfunction () throw (int,std::bad_exception) {
  throw 'x'; // throws char (not in exception-specification)
}
int main (void) {
  std::set_unexpected (myunexpected);
  try {
    myfunction();
  }
  catch (int) { std::cerr << "caught int\n"; }
  catch (std::bad_exception be) { std::cerr << "caught bad_exception\n"; }
  catch (...) { std::cerr << "caught some other exception\n"; }
  return 0;
}
```

Output:-
Unexpected handler called
Caught bad exception

Bad Allocation: Example

➢ Type of the exceptions thrown by the standard definitions of operator new and operator new[] when they fail to allocate the requested storage space.

➢ This class is derived from exception. See the exception class for the member definitions of standard exceptions.

➢ Its member what returns a null-terminated character sequence identifying the exception.

```cpp
#include <iostream>
#include <new>

int main () {
  try
  {
    int * myarray = new int[1000000000000];
  }
  catch (std::bad_alloc & exception)
  {
    std::cerr << "bad_alloc detected: " << exception.what();
  }
  return 0;
}
```

https://www.sololearn.com/Play/CPlusPlus

https://www.hackerrank.com/domains/cpp

Topic :throw, throws and finally

throwing Exception

- When an exception is detected, it is thrown using throw statement in the try block

- It is also possible, where we have nested try-catch statement

throw;

- 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.

throwing Exception
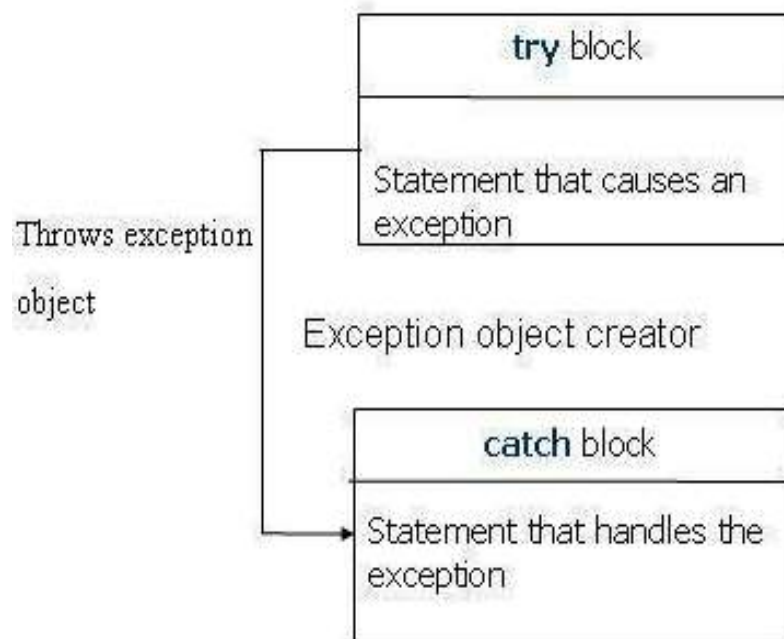
Rethrowing Exception
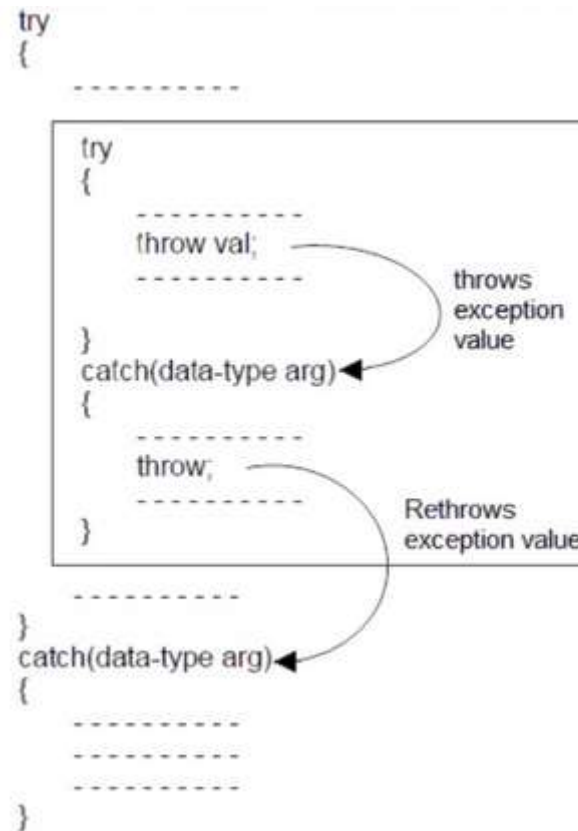
The *throw* point

- Used to indicate that an exception has occurred
- It is called "throwing an exception"
- Throw normally specifies an operand:
- Will be caught by closest exception handler

The *throw* point Example

```cpp
#include <iostream>
using namespace std;
int main() {
  try {
    int age = 15;
    if (age >= 18) {
      cout << "Access granted - you are old enough.";
    } else {
      throw (age);
    }
  }
  catch (int myNum) {
    cout << "Access denied - You must be at least 18 years old.\n";
    cout << "Age is: " << myNum;
  }
  return 0;   }
```

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

Try with Multiple Catch Block Example

```cpp
#include<iostream>
#include<vector>
using namespace std;
int main() {
        vector<int> vec;
        vec.push_back(0);
        vec.push_back(1);
        // access the third element, which doesn't exist
        try
        {
                vec.at(2);
        }
        catch (Exception i)
        { cout<<"Exception I caught"<<endl;}

        catch (exception& ex)
        {
                cout << "Exception occurred!" << endl;
        }
        return 0;  }
```

Topic : Exceptional Handling: predefined exceptional

User Defined Exception

We can define your own exceptions by inheriting and overriding exception class functionality. Following is the example, which shows how you can use exception class to implement your own exception in standard way:

Define New Exception

```cpp
class MyException:public exception {
        public: const char * what () const throw () {
                return "C++ Exception";
        }
};
int main()
{
try {
throw MyException();
}catch(MyException& e) {
 cout << "MyException caught" <<endl;
cout << e.what() << endl;
} catch(exception& e) {
 //Other errors
}
}
```

```cpp
class MyException : public exception {
        public: const char * what () const throw () {
                return "C++ Exception";
}
 };
int main() {
 try {
throw MyException();
 }catch(MyException& e) {
cout << "MyException caught" <<endl;
cout << e.what() << endl;
} catch(exception& e) {
//Other errors
}
 }
```

Here, what() is a public method provided by exception class and it has been overridden by all the child exception classes. This returns the cause of an exception.

Practices Program

1. C++ program to divide two numbers using try catch block.
2. Simple C++ Program for Basic Exception Handling.
3. Simple Program for Exception Handling Divide by zero Using C++ Programming
4. Simple Program for Exception Handling with Multiple Catch Using C++ Programming
5. Simple C++ Program for Catch All or Default Exception Handling
6. Simple C++ Program for Rethrowing Exception Handling in Function
7. Simple C++ Program for Nested Exception Handling

Program

```cpp
#include <iostream>
 #include <conio.h>
using namespace std;
int main() {
 int a,b;
cout << "Enter 2 numbers: ";
 cin >> a >> b;
 try {
 if (b != 0)
 {
float div = (float)a/b;
if (div < 0)
 throw 'e';
cout << "a/b = " << div;
 }
else throw b;
} catch (int e)
 { cout << "Exception: Division by zero"; }
catch (char st)
{ cout << "Exception: Division is less than 1"; }
catch(...) { cout << "Exception: Unknown";
}
getch();
 return 0;
 }
```

The throw point: Example

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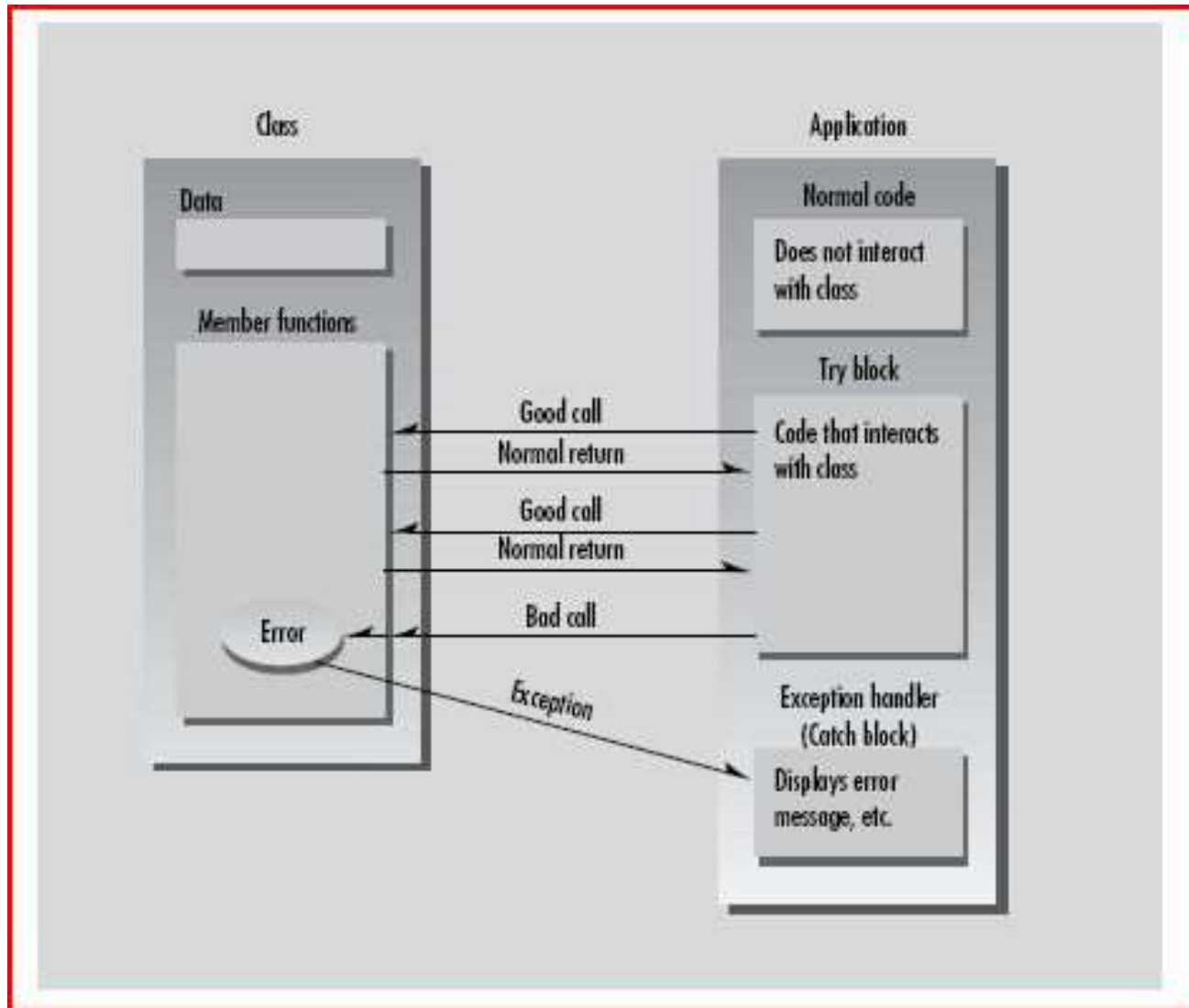
Finally

- 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.

Syntax

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

Example

```cpp
#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";
        }
}
```

Topic : Dynamic Modelling: Package Diagram, UML 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**.

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.

- A package is a collection of logically related UML elements.

- Packages are depicted as file folders and can be used on any of the UML diagrams.

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

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.

Package diagram is used to simplify complex class diagrams, you can group classes into packages. A package is a collection of logically related UML elements.
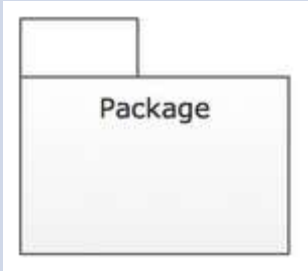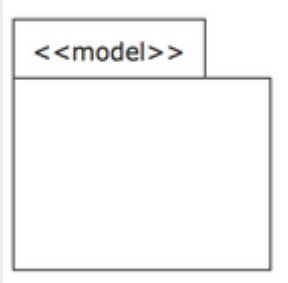
The diagram below is a business model in which the classes are grouped into packages:

•Packages appear as rectangles with small tabs at the top.

•The package name is on the tab or inside the rectangle.

•The dotted arrows are dependencies.

•One package depends on another if changes in the other could possibly force changes in the first.

Purpose

- It shows the relationship between software and hardware components in the target system.
- They are useful to show the system design that has subsystem, concurrent execution, compile time and execution time invocations, and hardware/software mapping by assigning the appropriate software components to the hardware devices.
- As they specify the distribution of software components in various devices and processors in the target environment, it will be easier for maintenance activities.
- Using this diagram it is easier to identify performance bottlenecks.

Notations

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

Package Diagram Example - Import

<<import>> - one package imports the functionality of other package
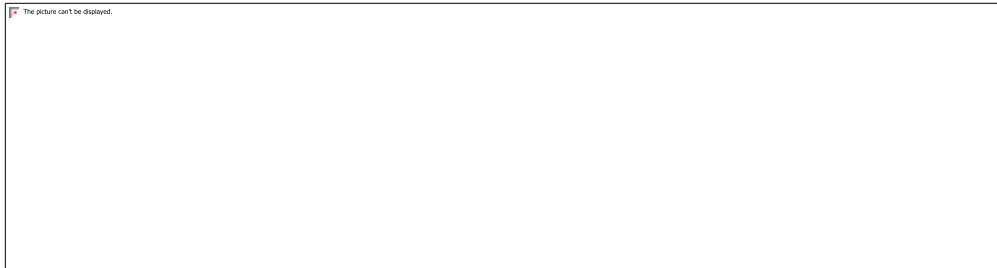
Package Diagram Example - Access

<<access>> - one package requires help from functions of other package.

Example
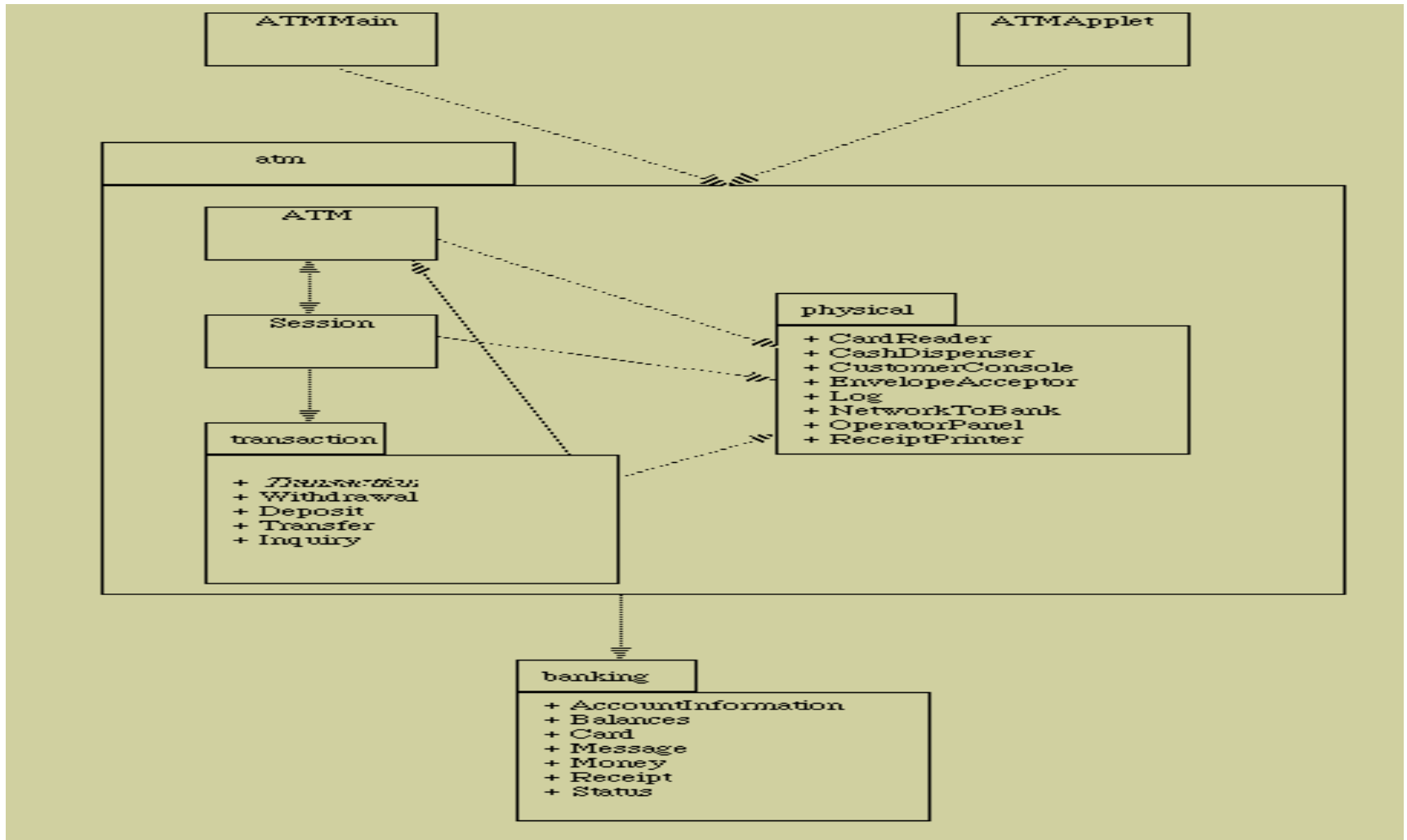
Package Diagram Example - Order Subsystem

The picture can't be displayed.

Component Diagram

- A component diagram shows the physical view of the system
- A component is an autonomous unit within a system.
- We combine packages or individual entities to form components.
- We can depict various components and their dependencies using a component diagram.
- Component diagram contain: component package, components, interfaces and dependency relationship.
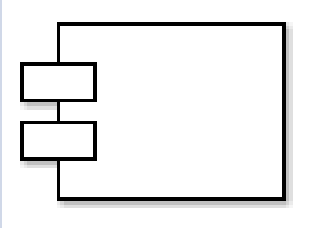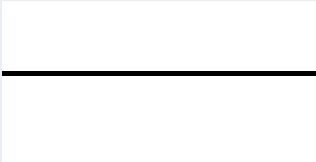
Component Diagram: Purpose

- It shows the structural relationship between the components of a system.
- It identifies the architectural perspective of the system as they enable the designer to model the high level software components with their interfaces to other components.
- It helps to organize source code into manageable chunks called components.
- It helps to specify a physical database.
- It can be easily developed by architects and programmers.
- It enables to model the high level software components and the interfaces to those components.
- The components and subsystem can be flexibly reused and replaced.

Guidelines to Draw

- 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.
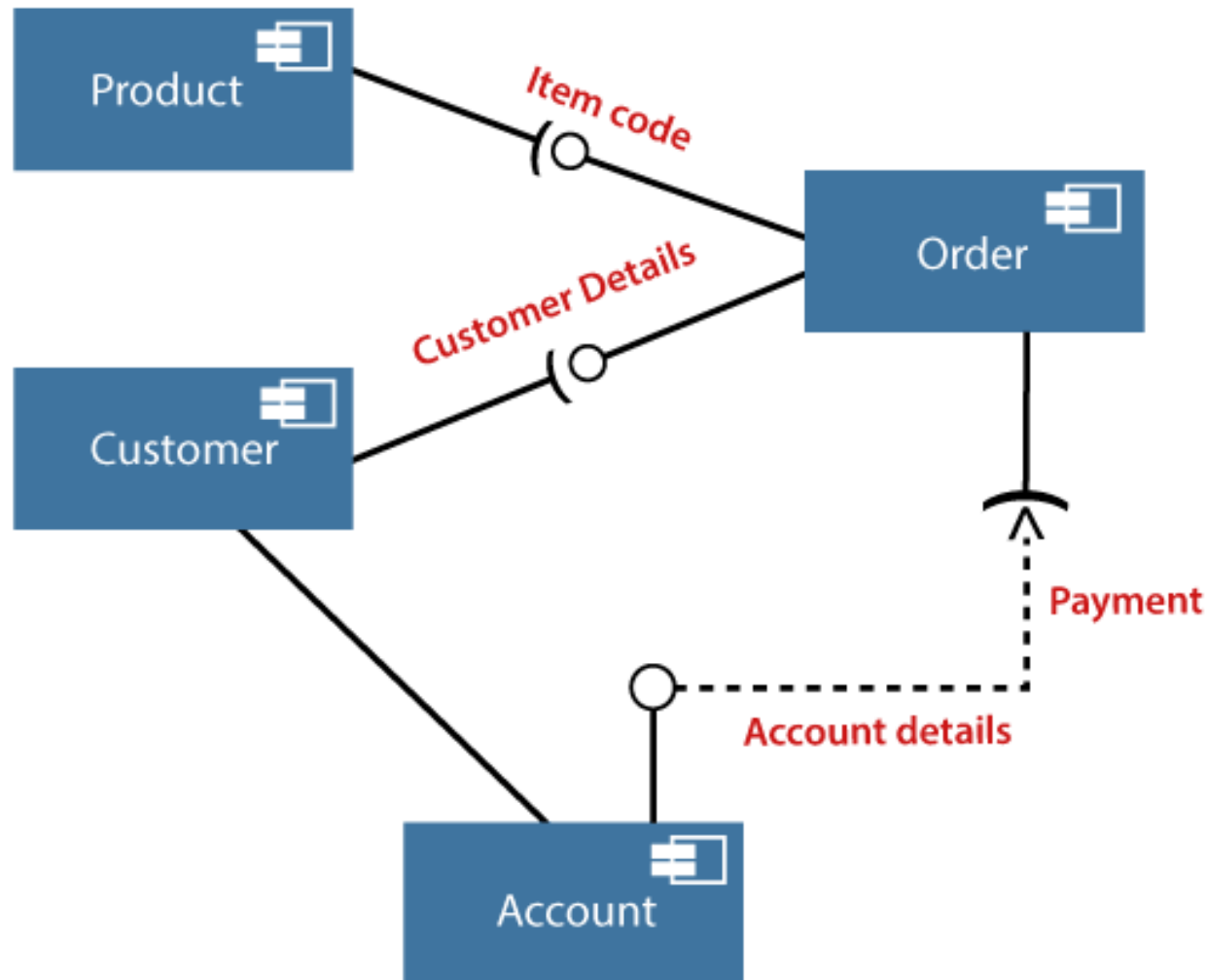- Use appropriate notations to draw the complete component diagram.

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.

Example
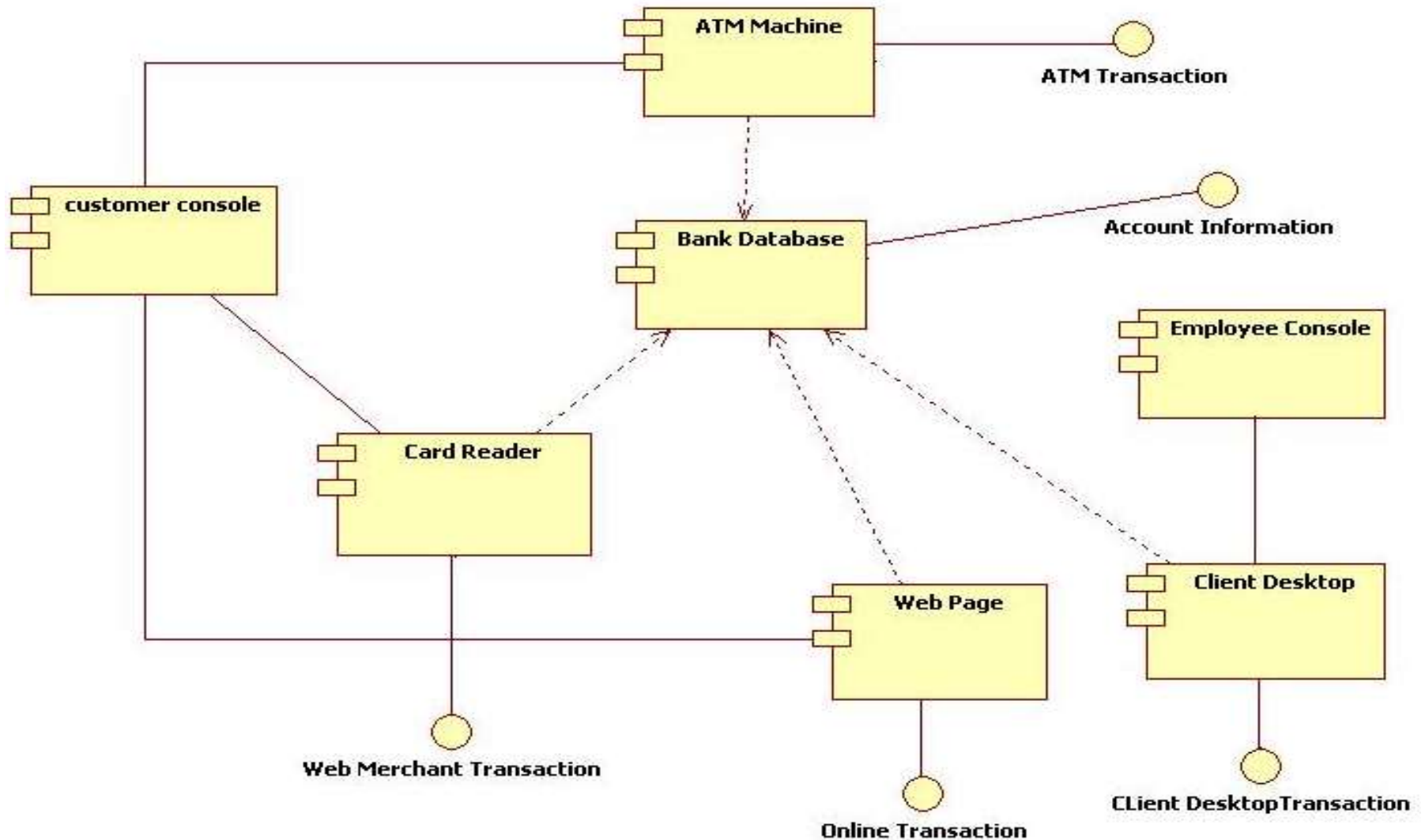
A component diagram for an online shopping system

Deployment Diagram

- A deployment diagram shows the physical placement of components in nodes over a network.
- A deployment diagram can be drawn by identifying nodes and components.
- A deployment diagram usually describes the resources required for processing and the installation of software components in those resources.
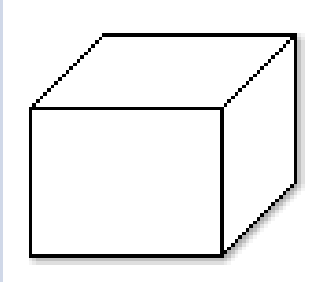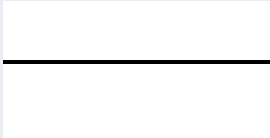
1. To envision the hardware topology of the system.
2. To represent the hardware components on which the software components are installed.
3. To describe the processing of nodes at the runtime.

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.
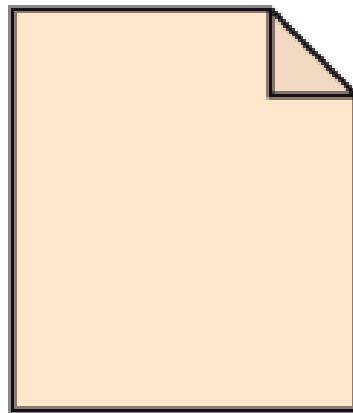
Notations

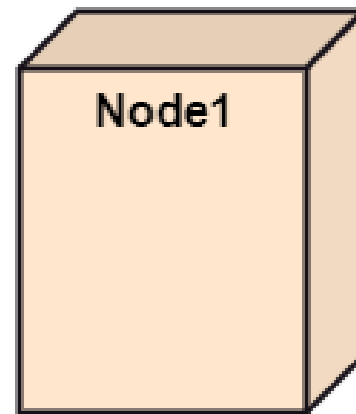| S.NO	NAME	SYMBOL	DESCRIPTION
1	Node		A node represents a physical component of the system. Node is used to represent physical part of a system like server, network etc.
2	Association		A structural relationship describing a set of links connected between objects.

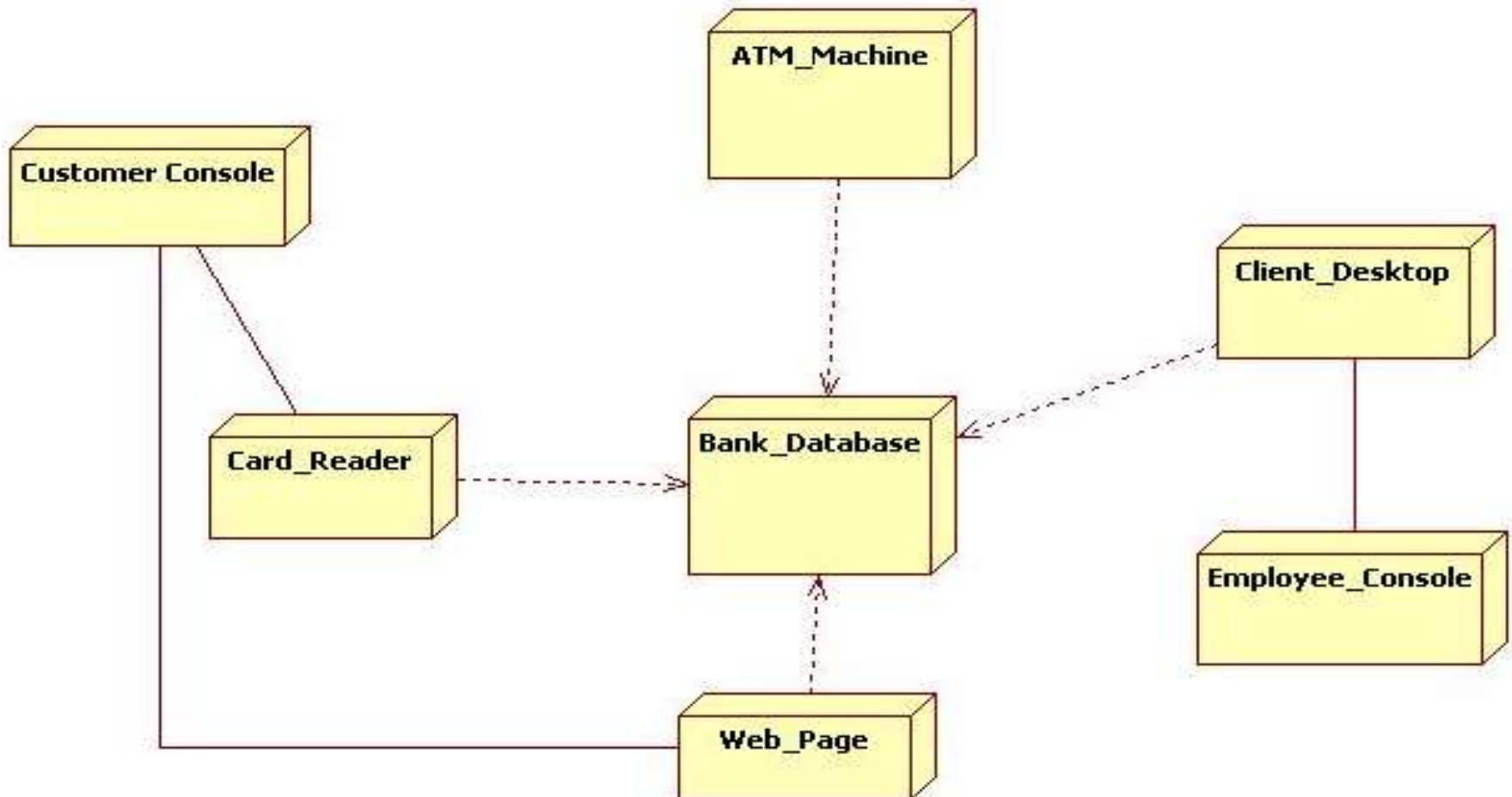# Symbol and notation of Deployment diagram

Component1
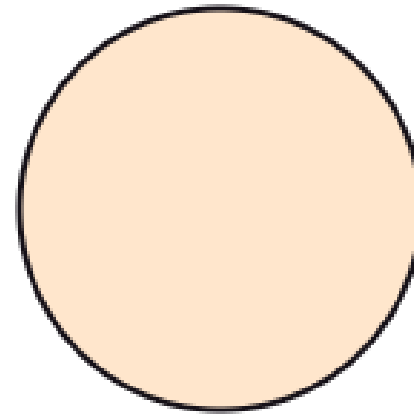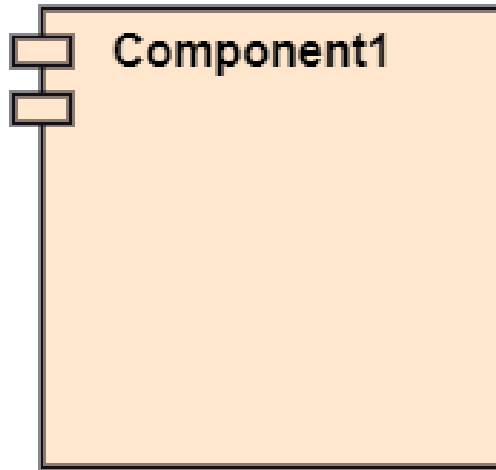
Interface1

Artifact1

Node1

Example

When to use Deployment Diagram

Deployment diagrams can be used for the followings:
1. To model the network and hardware topology of a system.
2. To model the distributed networks and systems.
3. Implement forwarding and reverse engineering processes.
4. To model the hardware details for a client/server system.
5. For modeling the embedded system.

Example

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