

Agenda

- Exception Handling
- Templates

Exception Handling

- Following are the operating system resources that we can use in application development
 1. Memory
 2. File
 3. Thread
 4. Socket
 5. Network connection
 6. IO Devices etc.
- Since OS resources are limited, we should use it carefully.
- If we make syntactical mistake in a program then compiler generates error.
- Without definition, if we try to access any member then linker generates error.
- Logical error / syntactically valid but logically invalid statements represents bug.
- If we give wrong input to the application then it generates runtime error/exception.
- Exception is an object, which is used to send notification to the end user of the system if any exceptional situation occurs in the program.
- If we want to manage OS resources carefully then we should use exception handling mechanism.
- Need of exception Handling:
 1. To avoid resource leakage.
 2. To handle all the runtime errors(exception) centrally.
- If we want to handle exception then we should use 3 keywords:
 1. try
 2. catch
 3. throw

1. try:

- try is keyword in C++.
- If we want to inspect exception then we should put statements inside try block/handler.
- try block must have at least one catch block/handler

2. throw:

- throw is keyword in C++.
- If we want to generate exception explicitly then we should use throw keyword.
- "throw statement" is a jump statement.

3. catch:

- If we want to handle exception then we should use catch block/handler.
- Single try block may have multiple catch block.
- Catch block can handle exception thrown from try block only.
- With the help of function, we can throw exception from outside try block.

- For thrown exception, if we do not provide matching catch block then C++ runtime gives call the `std::terminate` function which implicitly give call the `std::abort` function.
- A catch block, which can handle any type of exception is called generic catch block / catch-all handler.
- Generic catch block must appear after all specific catch block.

```
try
{
}
catch(...)
{
}
```

Exception Specification List

- Note : Dynamic Exception Specification List Deprecated in c++ 11 and removed in c++ 17

```
int calculate( int num1, int num2 )throw( ArithmeticException )
{
    if( num2 == 0 )
        throw ArithmeticException("Divide by zero exception");
    return num1 / num2;
}
```

- If an function fails to perform operation then it can throw exception. To maintain documentation of exception thrown by the function we should use exception specification list.
- To define exception specification list, we should use `throw` keyword.
- If exception specification list do not contain type of thrown exception then during failure it doesn't execute catch block rather C++ runtime give call to `std::unexpected` function which implicitly gives call to the `std::terminate` function.

Nested Exception Handling

- We can write try catch block inside another try block as well as catch block. It is called nested try catch block.
- Outer catch block can handle exception's thrown from inner try block.
- Inner catch block, cannot handle exception thrown from outer try block.
- If information, that is required to handle exception is incomplete inside inner catch block then we can rethrow that exception to the outer catch block.

```
class ArithmeticException{
private:
    string message;
public:

    ArithmeticException( string message ) : message( message ){}
    void printStackTrace( void )const{
```

```

        cout<<this->message<<endl;
    }
};
int main( void ){
    try{
        try{
            throw ArithmeticException("/ by zero");
        }
        catch( ArithmeticException &ex)
        {
            cout<<"Inside inner catch"<<endl;
            throw; //throw ex;
        }
    }
    catch( ArithmeticException &ex){
        cout<<"Inside outer catch"<<endl;
    }
    catch(...){
        cout<<"Inside generic catch block"<<endl;
    }
    return 0;
}

```

Stack Unwinding

- During execution of function if any exception occurs then process of destroying FAR and returning control back to the calling function is called stack unwinding.
- During stack unwinding, destructor gets called on local objects(not on dynamic objects).

Template

- If we want to write generic program in C++ then we should use template.
- Using template we can not reduce code size or execution time but we can reduce developers effort.
- It is designed for implementing generic data structure and algorithms
- Types of template:
 1. Function Template
 2. Class Template

1. Function Template

```

//template<typename T> //T : Type Parameter
template<class T> //T : Type Parameter
void swap_number( T &o1, T &o2 )
{
    T temp = o1;
    o1 = o2;
    o2 = temp;
}
int main( void )
{

```

```

int num1 = 10;
int num2 = 20;
swap_number<int>( num1, num2 );
//Here int is type argument
cout<<"Num1 : "<<num1<<endl;
cout<<"Num2 : "<<num2<<endl;
return 0;
}

```

- Type inference : It is ability of compiler to detect type of argument at compile time and passing it as a argument to the function.

```

template<class X, class Y>
void swap_number( X &o1, Y &o2 )
{
    X temp = o1;
    o1 = o2;
    o2 = temp;
}
int main( void )
{
    float num1 = 10.5f;
    double num2 = 20.5;
    swap_number<float, double>(num1, num2 );
    cout<<"Num1 : "<<num1<<endl;
    cout<<"Num2 : "<<num2<<endl;
    return 0;
}

```

- We can pass multiple type arguments to the function.
- Using template argument list, we can pass data type as a argument to the function.
- Using template we can write type safe generic code.

2. Class Template

- In C++, by passing data type as a argument, we can write generic code hence parameterized type is called template.

```

template<class T>
class Array // Parameterized type
{
private:
    int size;
    T *arr;
public:
    Array( void ) : size( 0 ), arr( NULL )
    {
    }
    Array( int size )

```

```
{
    this->size = size;
    this->arr = new T[ this->size ];
}
void acceptRecord( void ){
}
void printRecord( void ){
}
~Array( void ){ }
};
int main( void )
{
    Array<char> a1( 3 );
    a1.acceptRecord();
    a1.printRecord();
    return 0;
}
```