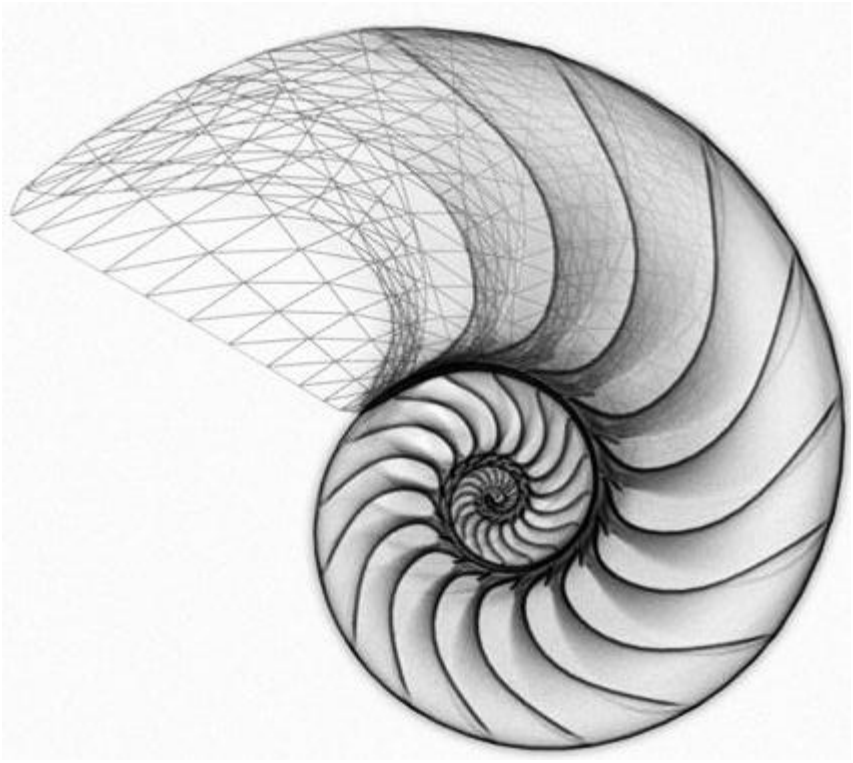
# Lecture 14



- Template
  - Function Template
  - Class Template
- Exception Handling
- Dynamic Memory Allocation

# GENERIC FUNCTIONS

- A generic function defines a general set of operations that will be applied to various types of data.
- A generic function has the type of data that it will operate upon passed to it as a parameter.
- Using this mechanism, the same general procedure can be applied to a wide range of data.

# GENERIC FUNCTIONS

- The general form of a template function definition is shown here:

```
template < class Ttype > ret_type func_name ( parameter list )
{
        // body of function
}
```

# Function template example

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

template <class X>
void swapargs (X &a, X &b)
{
    X temp ;
    temp = a;
    a = b;
    b= temp ;
}
```

```
int main ()
{
    int i=10, j =20;
    float x=10, y =23.3;

    cout << " Original i, j: " << i << ' ' << j << endl ;
    cout << " Original x, y: " << x << ' ' << y << endl ;

    swapargs (i, j); // swap integers
    swapargs (x, y); // swap floats

    cout << " Swapped i, j: " << i << ' ' << j << endl ;
    cout << " Swapped x, y: " << x << ' ' << y << endl ;

    return 0;
}
```

# GENERIC CLASSES

- In addition to defining generic functions, you can also define generic classes.
- When you do this, you create a class that defines all algorithms used by that class, but the actual type of the data being manipulated will be specified as a parameter when objects of that class are created.
- Generic classes are useful when a class contains generalizable logic.

# Class template example

```
// Class template
template <class T>
class Number
{
private:
    // Variable of type T
    T num;

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

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

```
int main()
{

    // create object with int type
    Number<int> numberInt(7);

    // create object with double type
    Number<double> numberDouble(7.7);

    cout << "int Number = " << numberInt.getNum() << endl;
    cout << "double Number = " << numberDouble.getNum() << endl;

    return 0;
}
```

# EXCEPTION HANDLING

- C++ provides a built-in error handling mechanism that is called exception handling.
- Using exception handling, you can more easily manage and respond to run-time errors.
- C++ exception handling is built upon three keywords: try, catch, and throw.
- In the most general terms, program statements that you want to monitor for exceptions are contained in a try block.
- If an exception (i.e., an error) occurs within the try block, it is thrown (using throw. The exception is caught, using catch, and processed.

# EXCEPTION HANDLING

The general form of try and catch are shown here:

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

# EXCEPTION HANDLING

- When an exception is thrown, it is caught by its corresponding catch statement, which processes the exception.
- There can be more than one catch statement associated with a try.
- The catch statement that is used is determined by the type of the exception. That is, if the data type specified by a catch matches that of the exception, that catch statement if executed (and all others are bypassed).
- When an exception is caught, arg will receive its value. If you don’t need access to the exception itself, specify only type in the catch clause-arg is optional.
- Any type of data can be caught, including classes that you create. In fact, class types are frequently used as exceptions.

- The general form of the throw statement is shown here:

```
throw exception ;
```

throw must be executed either from within the try block proper or from any function that the code within the block calls (directly or indirectly). exception is the value thrown.

# EXCEPTION HANDLING example

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

int main (){
    cout << " start \n";
    try // start a try block
    {
        cout << " Inside try block \n";
        throw 10; // throw an error
        cout << " This will not execute ";
    }
    catch (int i) // catch an error
    {
        cout << " Caught One ! Number is: ";
        cout << i << "\n";
    }
    cout << "end ";
    return 0;
}
```