OntarioTech Science

Kourosh Davoudi
kourosh@uoit.ca

Week 11: Function Template in STL
Exception Handling

CSCI 1061: Programming Workshop II

Learning Outcomes

In this week, we learn:

- Function templates in STL
  - How to use them via some case studies
- Exception Handling
  - Basin Ideas and syntax
  - Exception classes

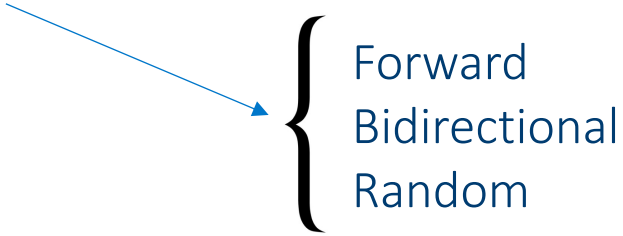
Generic Algorithms in STL

- The STL supplies a large set of generic algorithms that operate on containers:
- Hence, a generic algorithm may operate on any data structure that provides an iterator type that meets the iterator requirements of that algorithm.
- Parameters in a function call are iterators, not containers

```
#include <algorithm>
```

Case Study (find)

```
template <class ForwardIterator, class T>
ForwardIterator find(ForwardIterator first, ForwardIterator last, const T& target);
```



Description:

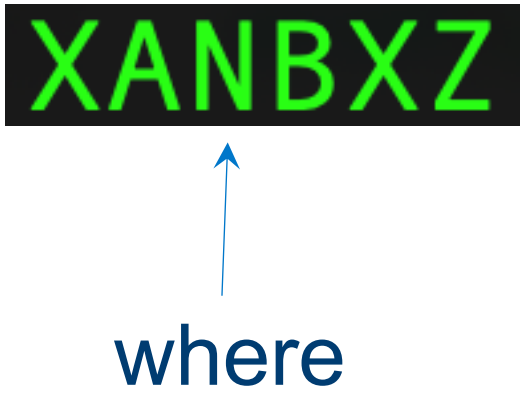
- Traverses the range [**first**, **last**) and returns an iterator located at the first occurrence of target.
- Returns second if target is not found (**last**).

Case Study (find)

```
vector<char> v = {'X', 'A', 'N', 'B', 'X', 'Z'};

vector<char>::iterator where;


where = find(v.begin(),v.end(), 'N');
```



Case Study (count)

```
template <class ForwardIterator, class T>
int count(ForwardIterator first, ForwardIterator last, const T & target);
```

Description:

- Traverse the range **[first, last)** and returns the number of elements equal to target.

```
vector<char> v = {'X', 'A', 'N', 'B', 'X', 'Z'};
cout << count(v.begin(),v.end(), 'X') << endl;
```

```
Count of X is : 2
```

Case Study (copy)

```
template <class ForwardIterator1, class ForwardIterator2>
ForwardIterator2 copy(
        ForwardIterator1 source_first, ForwardIterator1 source_last,
        ForwardIterator2 target_first);
```

Description:

- Action: Copies the elements at locations **[source_first, source_last)** to locations **[target_first, target_last).**
- Returns target_last

Case Study (copy)

```
template <class ForwardIterator1, class ForwardIterator2>
ForwardIterator2 copy(
        ForwardIterator1 source_first, ForwardIterator1 source_last,
        ForwardIterator2 target_first);
```

```
vector<char> v = {'X', 'A', 'N', 'B', 'X', 'Z'};

vector<char> w(6); // 6 is the size of vector

copy(v.begin(), v.end(), w.begin());
```

```
XANBXZ
```

Case Study (reverse)

```
template <class BidirectionalIterator>
void reverse(BidirectionalIterator first, BidirectionalIterator last);
```

Description:

- Reverses the order of the elements in the range [first, last).

```
vector<char> v = {'X', 'A', 'N', 'B', 'X', 'Z'};
reverse(v.begin(), v.end());
```

```
ZXBNAX
```

Exception Handling in C++

Error Handling in Programming

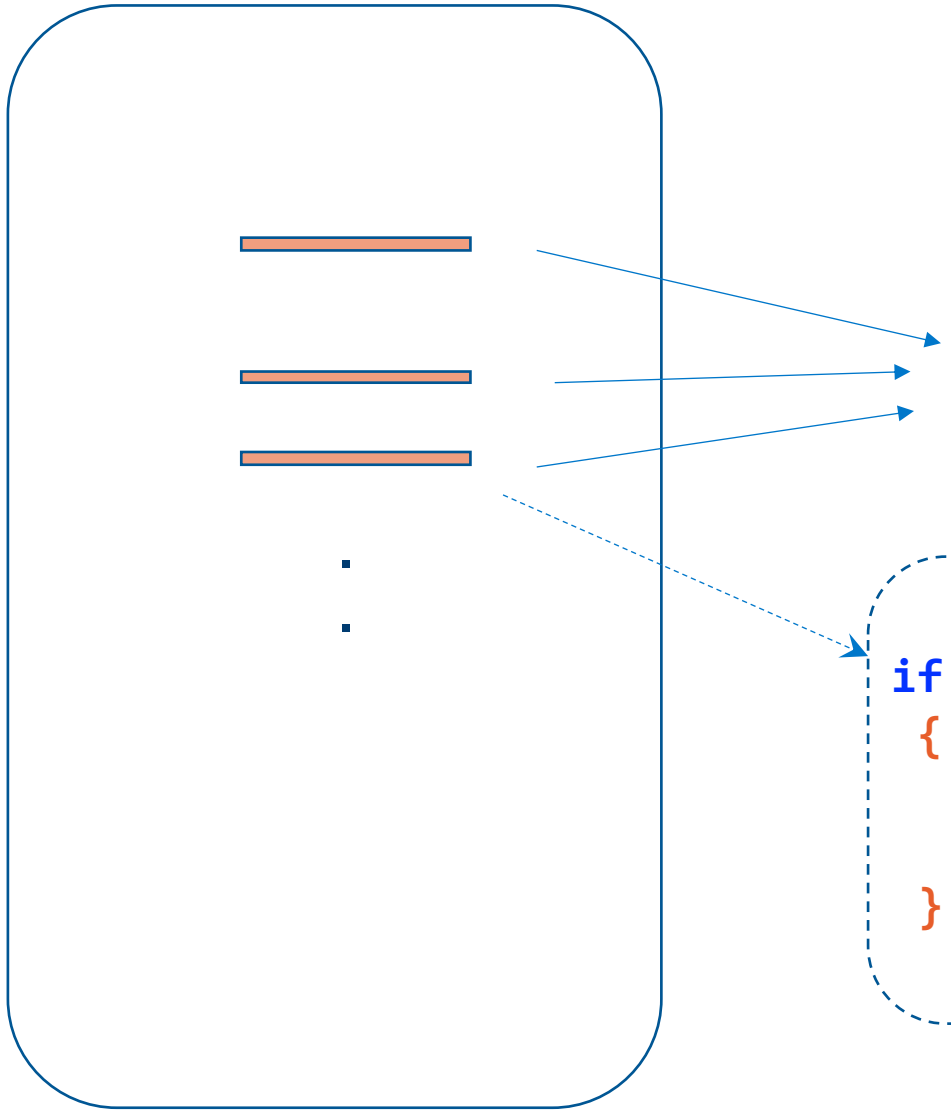
- Typical approach to development:
  - Write programs assuming things go as planned
  - Get "core" working
  - Then take care of "exceptional" cases:

File does not exist !

Divide by zero!

Why Exception?

f

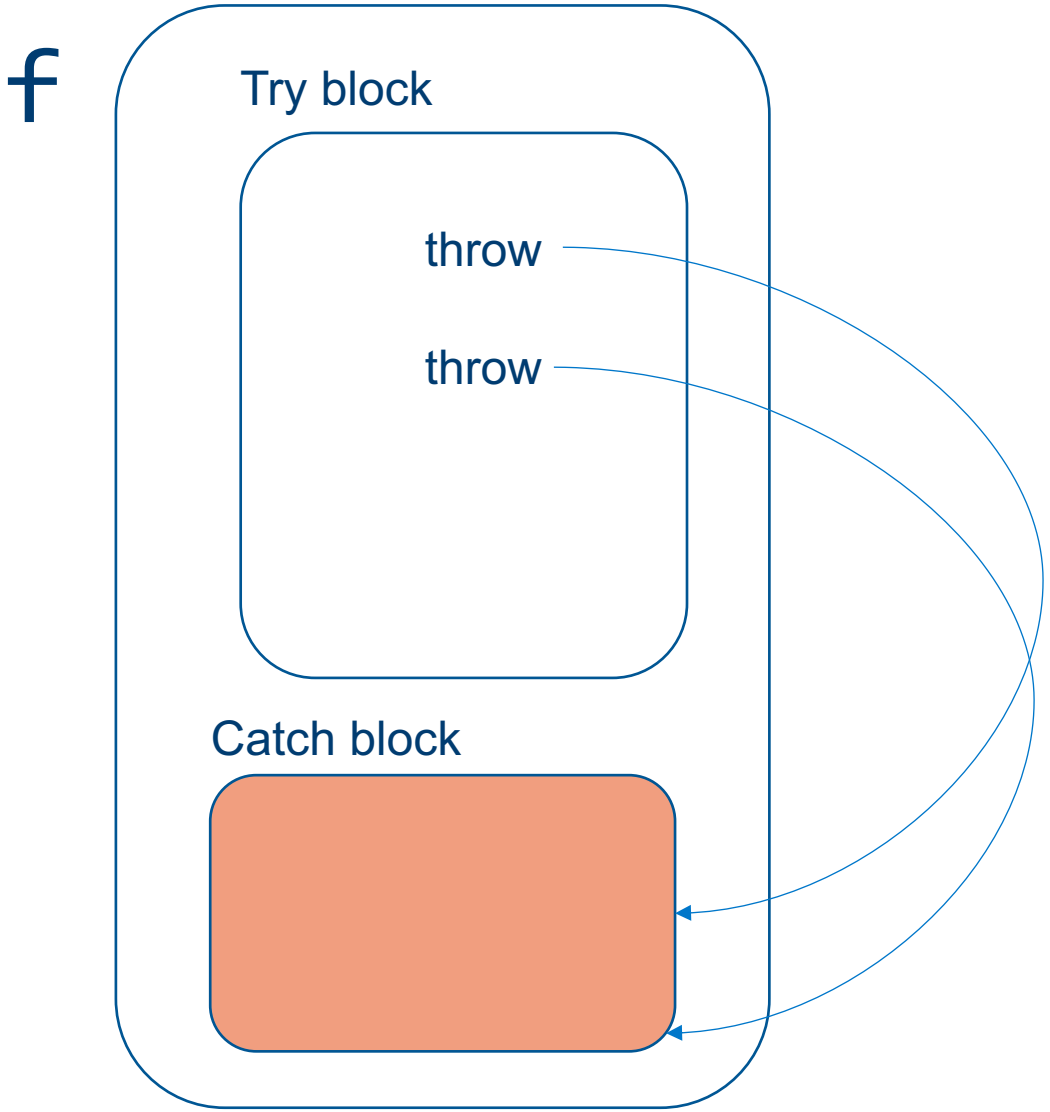


😕

Error handling codes speeds all over the program

```
if (fin.fail())
 {
    cerr << " Cannot open the input file!”;
    return 1;
 }
```

Why Exception?



Error handling codes are concentrated in the catch block

You can catch the exception in side the function which called or in caller function

A Toy example (without exception)

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


int main()
{

    int donuts, milk;

    double dpg;
    cout << "Enter number of donuts:";
    cin >> donuts;
    cout << "Enter number of glasses of milk:";
    cin >> milk;

    if (milk <= 0){
        cout << donuts << " donuts, and No Milk!\n" << "Go buy some milk.\n";
    }
    else{
        dpg = donuts / double(milk);
        cout << donuts << " donuts.\n";
        cout << milk << " glasses of milk.\n";
        cout << "You have " << dpg << " donuts for each glass of milk.\n";
    }

    cout << "End of program.\n";

    return 0;

}
```

Dealing with the error

A Toy example (with exception)

```
int main()
{

    int donuts, milk;

    double dpg;
    cout << "Enter number of donuts:";
    cin >> donuts;
    cout << "Enter number of glasses of milk:";
    cin >> milk;

    // put the code that you want to catch its exceptions
    try{
        if (milk <= 0)
            throw donuts;
        dpg = donuts / double(milk);
        cout << donuts << " donuts.\n";
        cout << milk << " glasses of milk.\n";
        cout << "You have " << dpg << " donuts for each glass of milk.\n";
    }

    // catch block deals with the exception
    catch(int e){
                cout << e << " donuts, and No Milk!\n" << "Go buy some milk.\n";
    }

    cout << "End of program.\n";

    return 0;

}
```

Much cleaner code

Try block

- Basic method of exception-handling is try-throw-catch

```
try
{
    Some_Code;
}
```

- Contains code for basic algorithm when all goes smoothly

Throw

- Inside try-block, when something unusual happens:

```
try
{
        Code_To_Try
        if (exceptional_happened)
              throw donuts;
        More_Code
}
```

  - Keyword throw followed by exception type
  - Called "throwing an exception"

Catch Block

```
catch(int e)
{
        cout << e << " donuts, and no milk!\n";
        cout << " Go buy some milk.\n";
}
```

- Looks like function definition with int parameter!
  - Not a function, but works similarly
  - Throw like "function call"

Note on syntax

SYNTAX

```
try
{
        Some_Statements
        <Either some code with a throw statement or
                        a function invocation that might throw an exception>
        Some_More_Statements
}
catch(Type e)
{
        <Code to be performed if a value of the
                        catch-block parameter type is thrown in the try block>
}
```

Note on throw

`throw` Statement

SYNTAX

```
throw Expression_for_Value_to_Be_Thrown;
```

When the `throw` statement is executed, the execution of the enclosing `try` block is stopped. If the `try` block is followed by a suitable `catch` block, then flow of control is transferred to the `catch` block. A `throw` statement is almost always embedded in a branching statement, such as an `if` statement. The value thrown can be of any type.

EXAMPLE

```
if (milk <= 0)
    throw donuts;
```

Note on catch Block

catch-Block Parameter

The catch-block parameter is an identifier in the heading of a catch block that serves as a placeholder for an exception (a value) that might be thrown. When a suitable value is thrown in the preceding try block, that value is plugged in for the catch-block parameter. (In order for the catch block to be executed, the value thrown must be of the type given for its catch-block parameter.) You can use any legal (nonreserved word) identifier for a catch-block parameter.

EXAMPLE

```
catch(int e)
{
    cout << e << " donuts, and No Milk!\n"
         << "Go buy some milk.\n";
}
```

e is the catch-block parameter.

Defining Exception Classes

- throw statement can throw value of any type
- Exception class
  - Contains objects with information to be thrown

```
try
{
        More_Code

        throw Exception_Object(parameter);

        More_Code

}
```

We can returns an object containing the information about exception

Exception Class Example

```
class NoMilk
{
public:
    NoMilk( ) {}
    NoMilk(int howMany) : count(howMany) {}
    int getCount( ) const { return count; }
private:
    int count;
};
```

Exception Class for Toy Example

```
int main( )
{
    int donuts, milk;
    double dpg;
    try
    {
        cout << "Enter number of donuts: ";
        cin >> donuts;
        cout << "Enter number of glasses of milk: ";
        cin >> milk;

        if (milk <= 0)
            throw NoMilk(donuts);

        dpg = donuts / double(milk);
        cout << donuts << " donuts.\n";
        cout << milk << " glasses of milk.\n";
        cout << "You have " << dpg;
        cout << " donuts for each glass of milk.\n";
    }
    catch(NoMilk e)
    {
        cout << e.getCount( ) << " donuts, and No Milk!\n" << "Go buy some milk.\n";
    }
    cout << "End of program.\n";

    return 0;
}
```

Invokes constructor of NoMilk class

Throwing Exception in Function Example

function throws DividebyZero exception

```
try
{
      quotient = safeDivide(num, den);
}
catch (DivideByZero)
{
  …
}
```

We can handle error in a caller.

See: exception-function1.cpp and exception-function2.cpp and

Throw List in Function

- You can define the exception that we want to handle for a function in both definition and prototype:

```
void someFunction() throw(DividebyZero, OtherException);
```

- We need to have catch for exception types DividebyZero or OtherException
- If there in to proper catch, unexpected() is invoked, which by default terminates the
- If function throws one that are not in the list, it invokes unexpected(), which by default terminates the program.