MECHTRON 2MD3

Data Structures and Algorithms for Mechatronics

Winter 2022

# 07 C++ Class Templates and Exceptions

Department of Computing and Software

Instructor:

Omid Isfahanialamdari

January 27, 2022

## Function Template

```
int integerMin(int a, int b)             // returns the minimum of a and b
  { return (a < b ? a : b); }
```

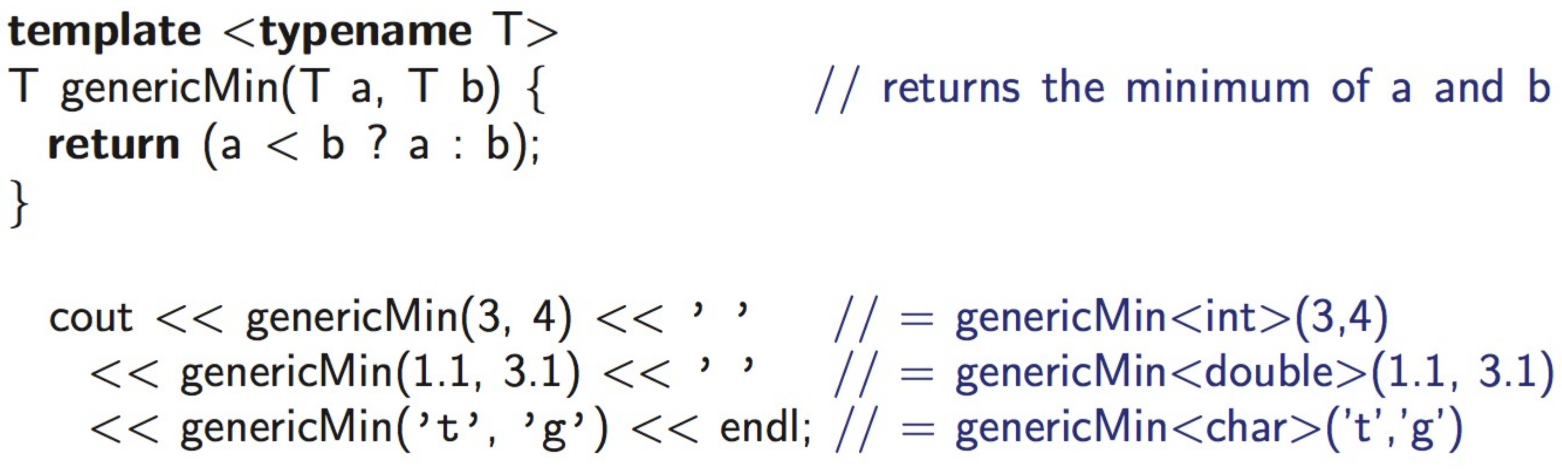
- Useful, but what about min of two doubles?
  - C-style answer: double doubleMin(double a, double b)

- Function template is a mechanism that enables this
  - Produces a generic function for an arbitrary type T.

```
template <typename T>
T genericMin(T a, T b) {                 // returns the minimum of a and b
  return (a < b ? a : b);
}
```

## Function Template

- Function template is a mechanism that enables this
  - Produces a generic function for an arbitrary type T.

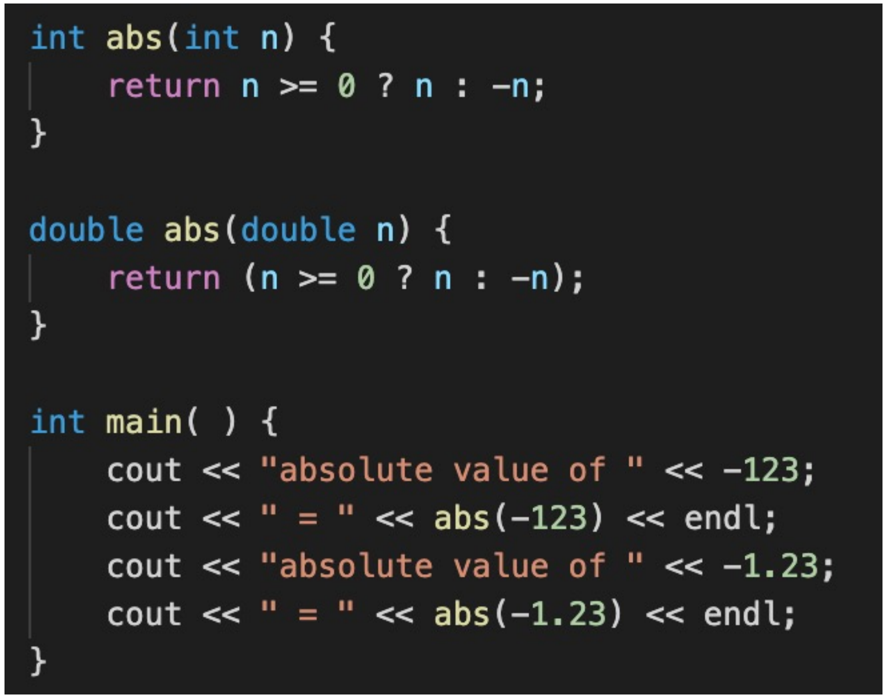


```
template <typename T>
T genericMin(T a, T b) {                // returns the minimum of a and b
  return (a < b ? a : b);
}

  cout << genericMin(3, 4) << ' '       // = genericMin<int>(3,4)
    << genericMin(1.1, 3.1) << ' '      // = genericMin<double>(1.1, 3.1)
    << genericMin('t', 'g') << endl;    // = genericMin<char>('t','g')
```

## Function Template

- Function overloading
  - Same function name, but different function prototypes
  - These functions do not have to have the same code
  - Does not help in code reuse, but helps in having a consistent name
- Function template
  - Same code piece, which applies to different types



```cpp
int abs(int n) {
    return n >= 0 ? n : -n;
}

double abs(double n) {
    return (n >= 0 ? n : -n);
}

int main( ) {
    cout << "absolute value of " << -123;
    cout << " = " << abs(-123) << endl;
    cout << "absolute value of " << -1.23;
    cout << " = " << abs(-1.23) << endl;
}
```

## Class Template

- In addition to function, we can define a generic template class
    - Example: BasicVector
        - Stores a vector of elements
        - Can access i-th element using [ ] just like an array (Vect class in this week's tutorial)

```
template <typename T>
class BasicVector {                          // a simple vector class
public:
  BasicVector(int capac = 10);               // constructor
  T& operator[](int i)                       // access element at index i
    { return a[i]; }
  // . . . other public members omitted
private:
  T* a;                                      // array storing the elements
  int capacity;                              // length of array a
};
```

## Class Template

- BasicVector
  - Constructor code?

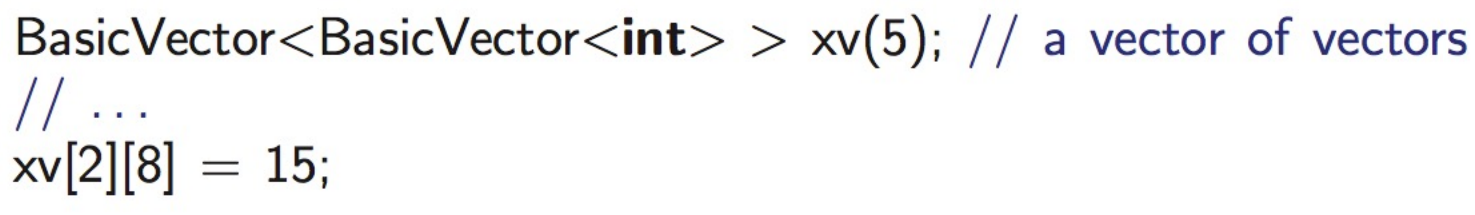
```
template <typename T>                     // constructor
BasicVector<T>::BasicVector(int capac) {
  capacity = capac;
  a = new T[capacity];                    // allocate array storage
}
```

- How to use?

```
BasicVector<int>     iv(5);        iv[3] = 8;
BasicVector<double>  dv(20);       dv[14] = 2.5;
BasicVector<string>  sv(10);       sv[7] = "hello";
```

## Class Template

- The actual argument in the instantiation of a class template can itself be a templated type
- Example: Two-dimensional array of int



```
BasicVector<BasicVector<int> > xv(5); // a vector of vectors
// ...
xv[2][8] = 15;
```

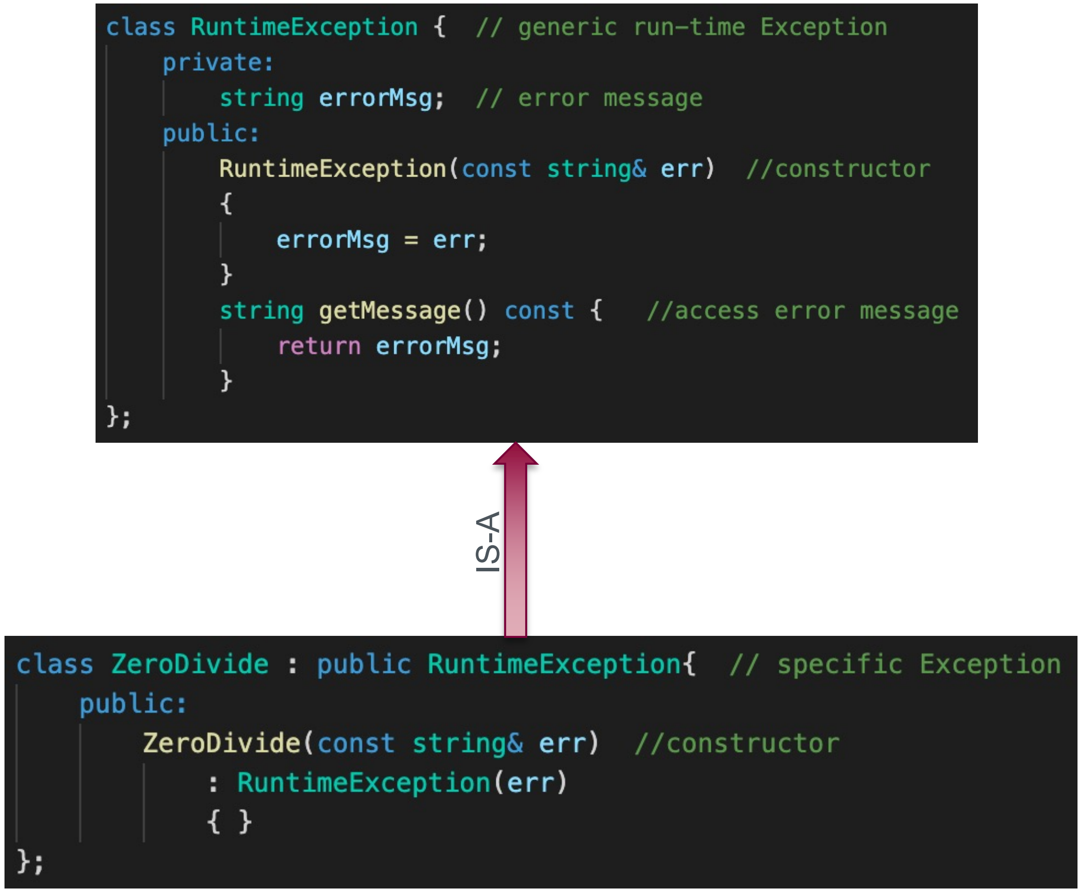
- BasicVector consisting of 5 elements, each of which is a BasicVector consisting of 10 integers
  - In other words, 5 by 10 matrix

## Exceptions

- Exception
  - Unexpected event, e.g., divide by zero
  - Can be user-defined, e.g., input of id > 1000
  - In C++, exception is said to be “thrown”
    - By your implemented code
    - By C++ runtime environment
  - A thrown exception is said to be “caught” by other code (exception handler)
  - In C, we often check the value of a variable or the return value of a function, and if… else… handles exceptions
    - Errors are notified by the returned value of the function, the exit code of the process, …
    - Dirty, inconvenient, hard to read

# Exception Class

- Exception handling with Inheritance!



```cpp
class RuntimeException {  // generic run-time Exception
    private:
        string errorMsg;  // error message
    public:
        RuntimeException(const string& err)  //constructor
        {
            errorMsg = err;
        }
        string getMessage() const {   //access error message
            return errorMsg;
        }
};
```

```cpp
class ZeroDivide : public RuntimeException{  // specific Exception
    public:
        ZeroDivide(const string& err)  //constructor
            : RuntimeException(err)
            { }
};
```

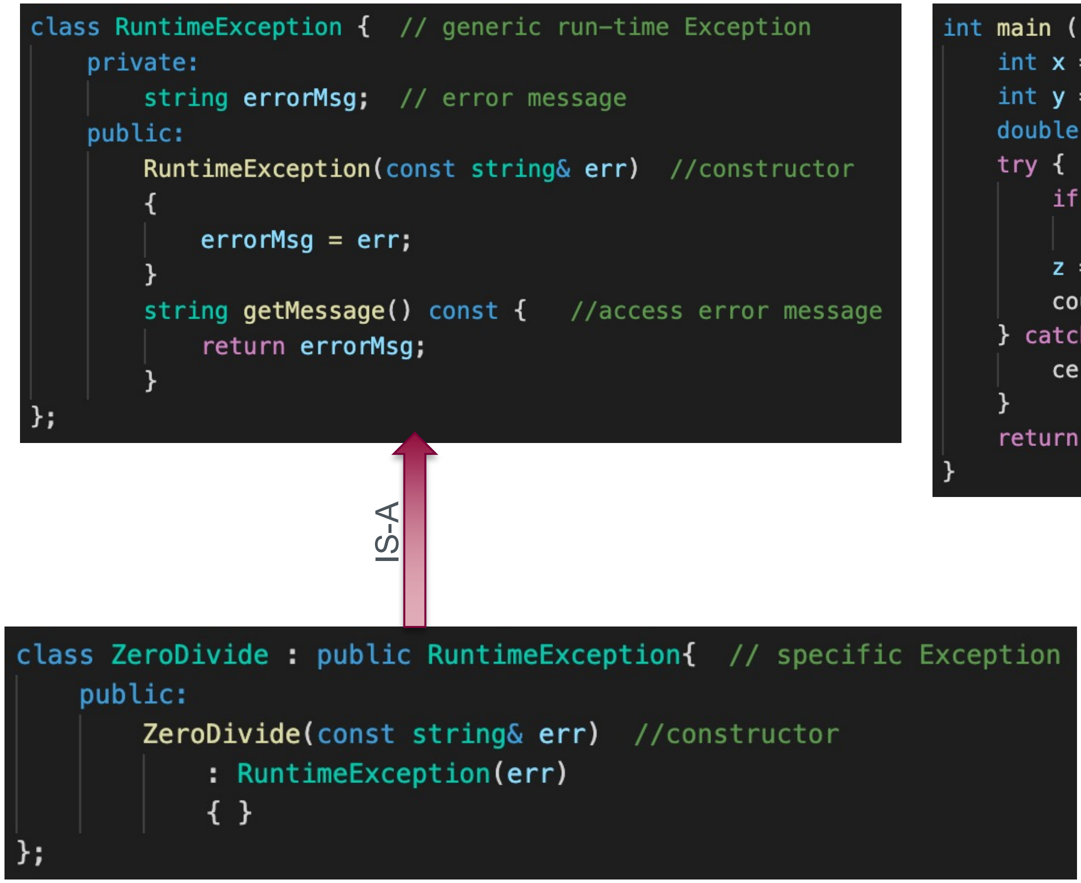
## Exception Throw, Try, Catch

- When an exception is thrown, it must be caught, or the program will abort
- If all goes smoothly, then execution leaves the **try** block and skips over its associated catch blocks.
- Otherwise, the control immediately **jumps** into the appropriate **catch** block for the exception thrown.

```
try {
  // ... application computations
  if (divisor == 0)                          // attempt to divide by 0?
    throw ZeroDivide("Divide by zero in Module X");
}
catch (ZeroDivide& zde) {
  // handle division by zero
}
catch (MathException& me) {
  // handle any math exception other than division by zero
}
```

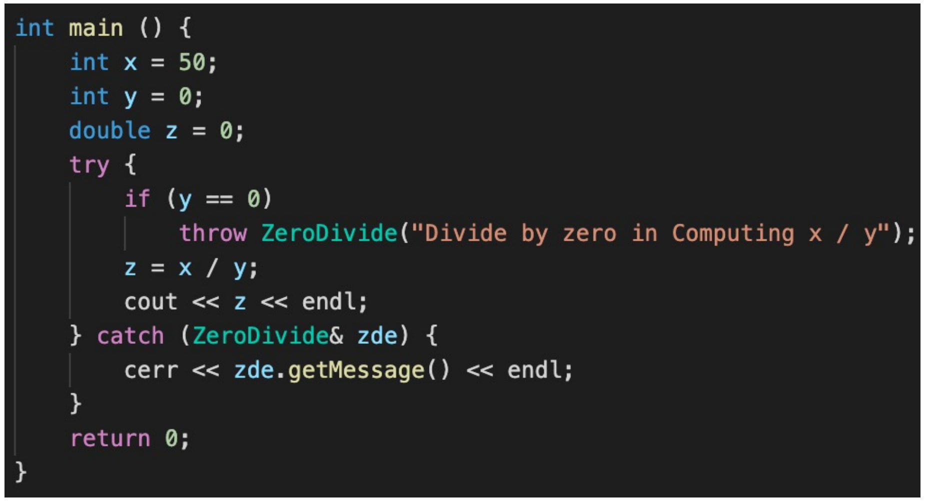
# Exception Class

- Exception handling in functions



```cpp
class RuntimeException {  // generic run-time Exception
    private:
        string errorMsg;  // error message
    public:
        RuntimeException(const string& err)  //constructor
        {
            errorMsg = err;
        }
        string getMessage() const {   //access error message
            return errorMsg;
        }
};
```

```cpp
class ZeroDivide : public RuntimeException{  // specific Exception
    public:
        ZeroDivide(const string& err)  //constructor
            : RuntimeException(err)
            { }
};
```



```cpp
int main () {
    int x = 50;
    int y = 0;
    double z = 0;
    try {
        if (y == 0)
            throw ZeroDivide("Divide by zero in Computing x / y");
        z = x / y;
        cout << z << endl;
    } catch (ZeroDivide& zde) {
        cerr << zde.getMessage() << endl;
    }
    return 0;
}
```

Output:

```
Divide by zero in Computing x / y
```

# Exception Specification

- In declaring a function, we should also specify the exceptions it might throw
  - Lets users know what to expect

```
void calculator() throw(ZeroDivide, NegativeRoot) {
    // function body ...
}
```

  - The function calculator (and any other functions it calls) can throw two exceptions or exceptions derived from these types
- Exceptions can be "passed through"

```
void getReadyForClass() throw(ShoppingListTooSmallException,
                                            OutOfMoneyException) {
   goShopping();   // I don't have to try or catch the exceptions
                   // which goShopping() might throw because
                   // getReadyForClass() will just pass these along.
   makeCookiesForTA();
}
```

## Any Exception and No Exception

- To be compatible with previous version of C++:
  - If a function does not provide a throw specification, then it may throw **any** exception
- To indicate that a function throws no exceptions, provide the throw specifier with an empty list of exceptions.

```
void func1();                    // can throw any exception
void func2() throw();            // can throw no exceptions
```

## Review Progression Code

- Demo!

| | |
|---|---|
| • Arithmetic progression (increment 1) | 0,1,2,3,4,5,... |
| • Arithmetic progression (increment 3) | 0,3,6,9,12,... |
| • Geometric progression (base 2) | 1,2,4,8,16,32,... |
| • Geometric progression (base 3) | 1,3,9,27,81,... |
| • Fibonacci progression (first = 0, second = 1) | 0,1,1,2,3,5,8,... |

# Questions?