Fu-Yin Cherng

National Taiwan University

Template and Exception Handling

Outline

- **Template**
- Exception Handling

What is Template?

- 模板
- Use the same template for different ingredients
- Templates can help to increase the flexibility and generalizability of our codes.

Redundant function

- similar functions with different **type** of parameters
- is it possible that we only need to **implement one** add() function for both double and int parameters?

```
double addDouble(double a, double b){
     return a + b;
}
int addInt(int a, int b){
     return a + b;
}
```

```
? add(? a, ? b){
     return a + b;
}
```

Yes! use Template

- **types** of parameters/variables can be **assigned later**
- it can be applied to **functions** and **classes**
- foundation of **generic** programming
  - writing code in a way that is independent of any particular type

Template + Function

- add template<typename T>
  - T: coder defined
- use T as the type of a, b, and function
- assign the type (int/double) to T in main()

```
template <typename T>
T add(T a, T b) {
   // T can be used as type for this function
   return a + b;
}

int main(){
     add<int>(1 ,2); //T as int, so return 3
     add<double>(1.1,1.1); //T as double, so
return 2.2
     add<int>(1.1,1.1); //return 2
}
```

Template + Class

- add template<typename T>
  - T: coder defined
- use T as the type for the class members
- T is the placeholder for the type name, which will be specified when ClassName is instantiated.

```
template <typename T>
class ClassName {
      // T can be used as type for this class
      public:
            T a;
            T add(T a, T b);
};

template <typename T>
T ClassName<T>::add(T a, T b){
      return a + b;
};
```

Template + Class

- all T in a is replaced by int
  - T in a.add() is int, so return (int) 3
- all T in b is replaced by double
  - T in b.add() is double, so return (double)

```
template <typename T>
class ClassName {
      // T can be used as type for this class
      public:
            T a;
            T add(T a, T b);
};

template <typename T>
T ClassName<T>::add(T a, T b){
      return a + b;
};

int main() {
      ClassName<int> a; // assign int to T
      ClassName<double> b; // assign double to T
      //ClassName<AnotherClassName> c;
      cout << a.add(1, 2) <<endl; // a's T is int,
      so print 3
      cout << b.add(1.1, 1.1) <<endl; //b's T is
      double, so print 2.2
};
```

Multiple type parameters

- two type parameters A & B for the function
- assign different types to A and B in main()
- pay attention to how the assigned type in main and how it influences the results

```
template<typename A, typename B>
void multi_template(A a, B b) {
     cout << a + b << endl; }

int main() {
     multi_template<double, int>(1.1, 2);
     // 3.1
     multi_template<int, int>(1.1, 2);
     // 3
     return 0;
}
```

Summary

- Template is useful!
- can make function and class more flexible and general
- When more flexible and general, need to consider more conditions when implementation and coding
- carefully checking and use **Exception Handling**

Outline

- Template
- **Exception Handling**

Exception Handling

- 例外/異常處理
- What is exception in C++?
- Why handle exception?
- How to handle exception?

What is exception in C++?

- exception = when an error occurs; bug
- different types of error
  - coding errors made by the programmer, errors due to wrong input, or other unforeseeable things
- When exceptions happen, C++ will normally **stop** and **generate an error message**
- then **throw** an exception (error message) to **describe** and indicate the error

Why handle exceptions?

- can get more information and control over the threw back exceptions
- increase the **robustness** of your program
- **cath** the **threw** back exceptions = exception handling

How to handle exceptions?

- try
  - define a block of code to be **tested** for errors while it is being executed.
- throw
  - **throws** an exception when a problem is **detected**; **create** a custom **error**
- catch
  - define a block of code to be executed, if an **error occurs in the try block**

```
try {
  // Block of code to try
  throw exception; // Throw an exception when a
problem arise
}
catch (ExceptionName e1) {
  // Block of code to handle errors
}
catch (ExceptionName e2) {
  // Block of code to handle errors
}
catch (...) {
  // code to handle any exception
}
```

How to handle exceptions?

- abnormal program **termination** occurs when n**o applicable catch** for an exception
- try and catch are paired
  - **try/catch** block
- try/catch block is placed around the code that might generate an exception.
  - **protected** code

```
try {
  // Block of code to try
  throw exception; // Throw an exception when a
problem arise
}
catch (ExceptionName e1) {
  // Block of code to handle errors
}
catch (ExceptionName e2) {
  // Block of code to handle errors
}
catch (...) {
  // code to handle any exception
}
```

How to handle exceptions?

- multiple catch to catch different types of exceptions
  - in case try block raises more than one exception
- use catch(...) to handle any type of exceptions

```
try {
  // Block of code to try
  throw exception; // Throw an exception when a problem arise
}
catch (ExceptionName e1) {
  // Block of code to handle errors
}
catch (ExceptionName e2) {
  // Block of code to handle errors
}
catch (...) {
  // code to handle any exception
}
```

Example

- divided by zero
- check if b==0 in division()
- throw error message (exception) when b==0

```
double division(int a, int b) {
   if( b == 0 ) {
      throw "Division by zero condition!";
      //throw exception (error message)
   }
   return (a/b);
}

int main () {
   int x = 50;
   int y = 0;
   double z = 0;

   try {
      z = division(x, y);
      cout << z << endl;
   } catch (const char* msg) {
     cerr << msg << endl; //print Division by zero
condition!
   }

   return 0;
};
```

Example

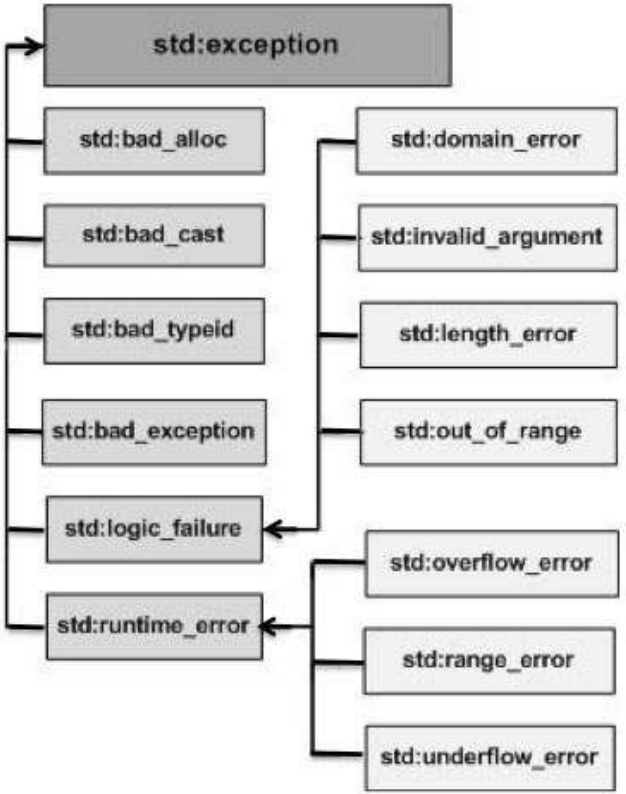
- catch/try block surrounding division() in main()
- catch error message by char* msg
- print out the error message in catch

```
double division(int a, int b) {
   if( b == 0 ) {
      throw "Division by zero condition!";
      //throw exception (error message)
   }
   return (a/b);
}

int main () {
   int x = 50;
   int y = 0;
   double z = 0;

   try {
      z = division(x, y);
      cout << z << endl;
   } catch (const char* msg) {
     cerr << msg << endl; //print Division by zero
condition!
   }

   return 0;
};
```

Standard Exceptions

- C++ provides a list of standard exceptions defined in <exception> which we can use in our programs.
- directly use them in your code

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



Summary

- Template
  - use type parameter; assign types later; generic programming
  - pay attention to how to use template in function and class
- Exception Handling
  - What, why, and how about exception handling
  - example of division()
  - pay attention to how to use catch/try block and throw