南方科技大学
SOUTHERN UNIVERSITY OF SCIENCE AND TECHNOLOGY

C/C++ Program Design

Lab 14, Exceptions

廖琪梅，王大兴

Exceptions

- Exceptions
- Assertion

Exception and Exception Handling

What is an exception?

An **exception** is a situation, which occurred by the runtime error. In other words, an exception is a runtime error. An exception may result in loss of data or an abnormal execution of program. Exception handling is a mechanism that allows you to take appropriate action to avoid runtime errors.

The default behavior for unexpected is to call **terminate**, and the default behavior for terminate is to call **abort**, so the program is to halt. Local variables in active stack frames are not destroyed, because **abort** shuts down program execution without performing such cleanup.

Let’s consider a simple example: a is divided by b, if b equals to zero, what will happen?

Example of a program without exception handling

noexceptions > demo1.cpp > ...

```
1   #include <iostream>
2   using namespace std;
3
4   double Quotient(int a, int b);
5
6   int main()
7   {
8       int a, b;
9       double d;
10      a = 5;
11      b = 0;
12      d = Quotient(a,b);
13      cout << "The quotient of " << a << "/" << b << " is:" << d << endl;
14
15      return 0;
16  }
17
18  double Quotient(int a, int b)
19  {
20      return (double) a/b;
21  }
```

```
The quotient of 5/0 is:inf
```

When divisor is zero, compiler generates a special floating-point value that represents infinity; ***cout*** displays this value as ***Inf, inf or INF***.

Example of a program with if statement to judge whether the divisor is zero. If the divisor is zero, call **abort()** function to terminate the program.

noexceptions > demo2.cpp > ...

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

double Quotient(int a, int b);

int main()
{
    int a, b;
    double d;
    a = 5;
    b = 0;
    d = Quotient(a,b);
    cout << "The quotient of " << a << "/" << b << " is:" << d << endl;

    return 0;
}

double Quotient(int a, int b)
{
    if(b == 0)
    {
        cout << "Divisor can not be zero!" << endl;
        abort();
    }
    return (double) a/b;
}
```

```
Divisor can not be zero!
Aborted
```

You can use exit() function without a message displayed.

Example of a program with the return a boolean value to judge the condition, add another argument to store the answer.

noexceptions > demo3.cpp > ...

```cpp
#include <iostream>

using namespace std;

bool Quotient(int a, int b, int &c);

int main()
{
    int a, b, c;
    double d;
    a = 5;
    b = 0;

    if (Quotient(a, b, c))
        cout << "The quotient of " << a << "/" << b << " is:" << c << endl;
    else
        cout << "The divisor can not be zero!" << endl;

    return 0;
}
bool Quotient(int a, int b, int &c)
{
    if (b == 0)
        return false;
    else
    {
        c = a / b;
        return true;
    }
}
```

To judge whether the divisor is zero by return value

```
The divisor can not be zero!
```

Exception handling

Exceptions provide a way to react to exceptional circumstances (like runtime errors) in programs by transferring control to special functions called ***handlers***. C++ provides **three keywords** to support exception handling.

- **try**: The **try** block contain statements which may generate exceptions. When an exceptional circumstance arises within that block, an exception is thrown that transfers the control to the exception handler. If no exception is thrown, the code continues normally and all handlers are ignored.
- **throw**: When an exception occur in **try** block, it is thrown to the **catch** block using **throw** keyword.
- **catch**: The **catch** block defines the action to be taken when an exception occur. Exception handlers are declared with the keyword **catch**, which must be placed immediately after the **try** block.

The syntax for using **try/catch** as follows:

```
try {
    // protected code
}catch(ExceptionType1 e1){
    // handle e1
}catch(ExceptionType2 e2){
    // handle e2
}catch(ExceptionType3 e3){
    // handle e3
}catch(...){
    // default handle
}
```

Catches any type exception

You can list down multiple **catch** statements to catch different type of exceptions in case your **try** block raises more than one exception in different situations.

How does exception handling work?

- When a problem is detected during the computation, an exception is raised by using **keyword throw.**
- The raised exceptions are handled by the **catch block**. This exception handler is indicated by the **keyword catch**.
- **try** block is responsible for testing the existence of exceptions.
- After an exception has been handled, the program execution resumes after the **try-catch** block, not after the **throw** statement.

The following figure explains more about this:

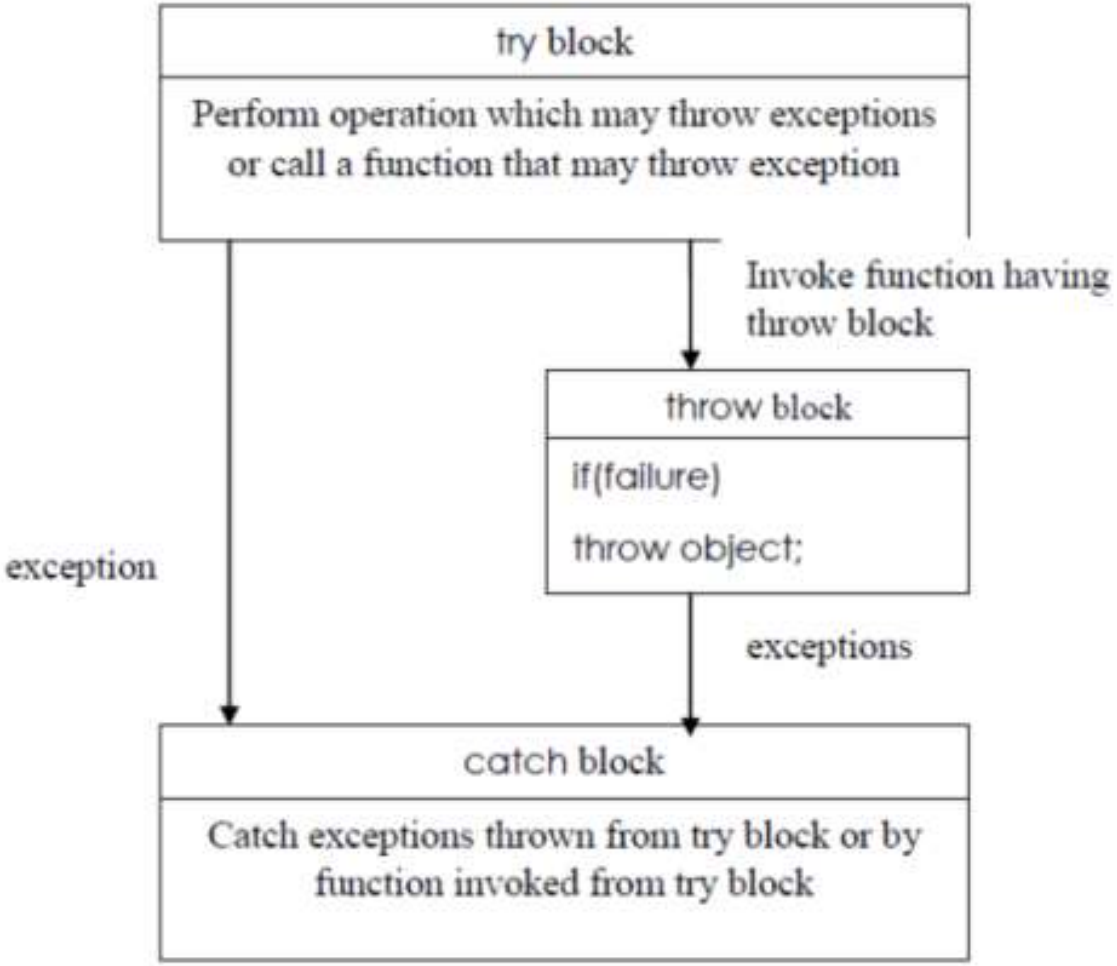


Figure: exception handling mechanism in C++

Steps taken during exception handling:

1. **Hit the exception**(detect the problem causing exception)

2. **Throw the exception**(inform that an error has occurred)

3. **Catch the exception**(receive the appropriate actions)

4. **Handle the exception**(take appropriate actions)

Example of a program with exception handling using try and catch, throw an exception in **try** block in main()

exceptions > exceptiondemo1.cpp > ...

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

int main()
{
    int a, b;
    double d;
    a = 5;
    b = 0;

    try{
        if(b == 0)
            throw "The divisor can not be zero!";
        d = (double)a/b;
        cout << "The quotient of " << a << "/" << b << " is:" << d << endl;
    }catch(const char* perror){
        cout << perror << endl;
    }catch(int code){
        cout << "Exception code:" << code << endl;
    }

    return 0;
}
```

match

```
The divisor can not be zero!
```

Example of a program with exception handling using try and catch, throw an exception in other function, handlers are in main()

exceptions > exceptiondemo2.cpp > ...

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

double Quotient(int a, int b);

int main()
{
    int a, b;
    double d;
    a = 5;
    b = 0;

    try{
        d = Quotient(a,b);
        cout << "The quotient of " << a << "/" << b << " is:" << d << endl;
    }catch(const char* perror){
        cout << perror << endl;
    }catch(int code){
        cout << "Exception code:" << code << endl;
    }

    return 0;
}

double Quotient(int a, int b)
{
    if(b == 0)
        throw 404;

    return (double)a/b;
}
```

match

```
Exception code:404
```

Note: In general, no conversions are applied when matching exceptions to catch clauses.

exceptions > exceptiondemo2.cpp > ...

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

double Quotient(int a, int b);

int main()
{
    int a, b;
    double d;
    a = 5;
    b = 0;

    try{
        d = Quotient(a,b);
        cout << "The quotient of " << a << "/" << b << " is:" << d << endl;
    }catch(const char* perror){
        cout << perror << endl;
    }catch(double code){
        cout << "Exception code:" << code << endl;
    }

    return 0;
}

double Quotient(int a, int b)
{
    if(b == 0)
        throw 404;

    return (double)a/b;
}
```

does not match

```
terminate called after throwing an instance of 'int'
Aborted
```

Define and using exception class

exceptions > exceptiondemo3.cpp > ...

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

// difine your exception class
class RangeError{
private:
    int iVal;
public:
    RangeError(int _iVal) {iVal = _iVal;}
    int getVal() {return iVal;}
};

char to_char(int n)
{
    if( n < numeric_limits<char>::min() || n > numeric_limits<char>::max())
        throw RangeError(n);

    return (char)n;
}

void gain(int n)
{
    try{
        char c = to_char(n);
        cout << n << " is character " << c << endl;
    }catch(RangeError &re){
        cerr << "Cannot convert " << re.getVal() << " to char\n" << endl;
        cerr << "Range is " << (int)numeric_limits<char>::min();
        cerr << " to " << (int)numeric_limits<char>::max() << endl;
    }
}

int main()
{
    gain(-130);

    return 0;
}
```

Define your exception class

Throw the exception and invoke the constructor

Catch and handle the exception

```
Cannot convert -130 to char

Range is -128 to 127
```

Handling exceptions from an inheritance hierarchy

exceptions > exceptiondemo4.cpp > ...

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

class MathException { };
class OverflowException : public MathException{ };
class UnderflowException : public MathException{ };
class ZeroDivideException : public MathException { };

double divide(int numerator, int denominator)
{
    if(denominator == 0)
        throw ZeroDivideException();

    double d = (double) numerator/denominator;
    return d;
}

int main()
{
    try{
        cout << divide(6,0) << endl;
    }catch(ZeroDivideException& zd){
        cerr << "Zero Divide Error" << endl;
    }catch(OverflowException& oe){
        cerr << "Overflow Error" << endl;
    }catch(UnderflowException& ue){
        cerr << "Underflow Error" << endl;
    }catch(MathException& me){
        cerr << "Math Error" << endl;
    }

    return 0;
}
```

Note: A kind of conversion is applied when matching exceptions to **catch clauses**. That is inheritance-based conversions. A catch clause for base class exceptions is allowed to handle exceptions of derived class types, too.

```
Zero Divide Error
```

exceptions > exceptiondemo4.cpp > ...

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

class MathException { };
class OverflowException : public MathException{ };
class UnderflowException : public MathException{ };
class ZeroDivideException : public MathException { };

double divide(int numerator, int denominator)
{
    if(denominator == 0)
        throw ZeroDivideException();

    double d = (double) numerator/denominator;
    return d;
}

int main()
{
    try{
        cout << divide(6,0) << endl;
    }catch(MathException& me){
        cerr << "Math Error" << endl;
    }catch(ZeroDivideException& zd){
        cerr << "Zero Divide Error" << endl;
    }catch(OverflowException& oe){
        cerr << "Overflow Error" << endl;
    }catch(UnderflowException& ue){
        cerr << "Underflow Error" << endl;
    }

    return 0;
}
```

Note: **catch** clauses are always tried in the order of their appearance. Hence, it is possible for an exception of a derived class type to be handled by a catch clause for one of its base class types.

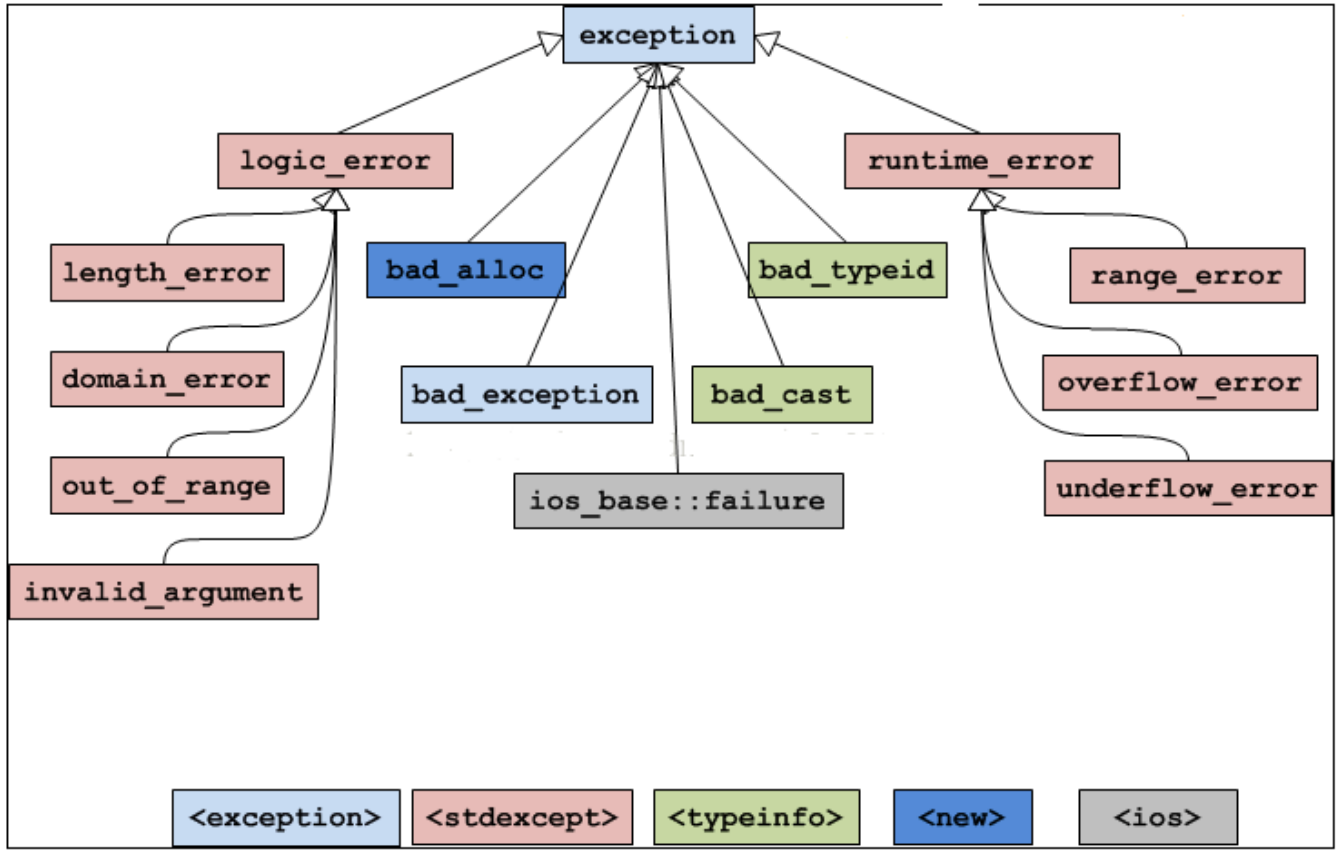
Compilers may warn you if a catch clause for a derived class comes after one for a base class.

```
exceptiondemo4.cpp: In function ‘int main()’:
exceptiondemo4.cpp:24:6: warning: exception of type ‘ZeroDivideException’ will be caught
   24 |     }catch(ZeroDivideException& zd){
      |      ^~~~~
exceptiondemo4.cpp:22:6: warning:    by earlier handler for ‘MathException’
   22 |     }catch(MathException& me){
      |      ^~~~~
```

```
Math Error
```

C++ Standard Exceptions

C++ provides a list of standard exceptions defined in which we can use in our programs.



Exception	Description
std::exception	An exception and parent class of all the standard C++ exceptions.
std::bad_alloc	This can be thrown by **new**.
std::bad_cast	This can be thrown by **dynamic_cast**.
std::bad_exception	This is useful device to handle unexpected exceptions in a C++ program
std::bad_typeid	This can be thrown by typeid.
std::logic_error	An exception that theoretically can be detected by reading the code.
std::domain_error	This is an exception thrown when a mathematically invalid domain is used
std::invalid_argument	This is thrown due to invalid arguments.
std::length_error	This is thrown when a too big std::string is created
std::out_of_range	This can be thrown by the at method from for example a std::vector and std::bitset<>::operator.
std::runtime_error	An exception that theoretically can not be detected by reading the code.
std::overflow_error	This is thrown if a mathematical overflow occurs.
std::range_error	This is occured when you try to store a value which is out of range.
std::underflow_error	This is thrown if a mathematical underflow occurs.

```
class exception{
    public:
        exception () throw();  //constructor
        exception (const exception&) throw();  //copy constructor
        exception& operator= (const exception&) throw();  //assignment operator
        virtual ~exception() throw();  //destructor
        virtual const char* what() const throw();  //virtual function
}
```

Exception specification used in function declaration, with no argument indicates that the function is not allowed to throw any exceptions.

what() is a public method provided by **exception class** which returns a string and it has been overridden by all the child exception classes.

Define your own exception class derived from exception class and override **what()** method

exceptions > exceptiondemo5.cpp > ...

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

class MyException : public exception
{
public:
    const char* what() const throw()
    {
        return "C++ Exception.";
    }
};

int main()
{
    try{
        throw MyException();
    }catch(MyException& me){
        cout << "MyException is caught." << endl;
        cout << me.what() << endl;
    }catch(exception& e){
        cout << "Base class exception is cautch." << endl;
        cout << e.what() << endl;
    }
    return 0;

}
```

```
MyException is caught.
C++ Exception.
```

Note: **use catch-by-reference for exception objects**

catch-by-value:Derived class exception objects caught as base class exceptions have their derivedness "sliced off." Such "sliced" objects are base class objects: they lack derived class data members, and when virtual functions are called on them, they resolve to virtual functions of the base class. So use **catch-by-reference** for exception objects and invoke the virtual function of the derived class.

exceptions > exceptiondemo6.cpp > ...

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

class MyException : public exception
{
public:
    virtual const char* what() const noexcept
    {
        return "C++ Exception.";
    }
};

int main()
{
    try{
        throw MyException();
    }catch(exception e){
        cout << "Base class exception is cautch." << endl;
        cerr << e.what() << endl;
    }
    return 0;

}
```

It will not throw any exception

Catch the exception by value

```
Base class exception is cautch.
std::exception
```

Invoke what() of the exception class rather than the MyException class.

exceptions > exceptiondemo6.cpp > ...

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

class MyException : public exception
{
public:
    virtual const char* what() const noexcept override
    {
        return "C++ Exception.";
    }
};

int main()
{
    try{
        throw MyException();
    }catch(exception& e){
        cout << "Base class exception is cautch." << endl;
        cerr << e.what() << endl;
    }
    return 0;

}
```

It is overriding a virtual method of the base class.

Catch the exception by reference

```
Base class exception is cautch.
C++ Exception.
```

Invoke what() of the MyException class not the exception class.

Assertions in C/C++

Assertions are statements used to test assumptions made by programmers. It is designed as a macro in C/C++. Following is the syntax for assertion:

void assert(int expression);

If the expression evaluates to 0 (false), then the expression, source code filename, and line number are sent to the standard error, and then **abort()** function is called.

noexceptions > testassert.cpp > ...

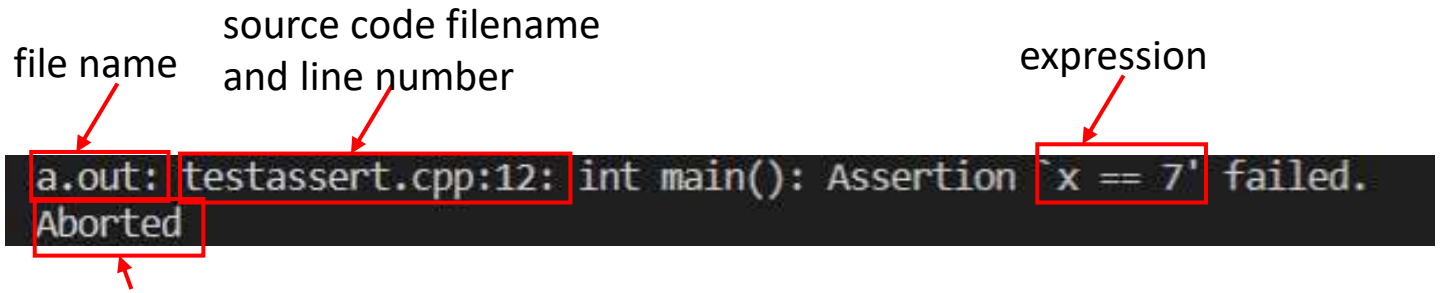
```cpp
#include <assert.h>
#include <iostream>
using namespace std;

int main()
{
    int x = 7;
    //x is accidentally changed to 9  */
    x = 9;

    // Programmer assumes x to be 7 in rest of the code
    assert(x == 7);

    // Rest of the code
    cout << "The original value of x is 7" << endl;

    return 0;
}
```



Assertions are mainly used to check logically impossible situations. For example, they can be used to check the state of a code which is expected before it starts running, or the state after it finishes running.

- Verify the validity of the passed argument at the beginning of the function.

```
int resetBufferSize(int nNewSize)
{
      assert(nNewSize >= 0);
      assert(nNewSize <= MAX_BUFFER_SIZE);
    ...
}
```

```
// is not recommended
assert(nOffset>=0 && nOffset+nSize<=m_nInfomationSize);

// is recommended, each assert test only on condition
assert(nOffset >= 0);
assert(nOffset+nSize <= m_nInfomationSize);
```

- Each assert tests only one condition, because when multiple conditions are tested at the same time, it is not intuitive to determine which condition failed if the assertion failed.
- Ignores assertions. We can completely remove assertions at compile time using the preprocessor NDEBUG. Put **#define NDEBUG** at the beginning of the code, before inclusion of <assert.h>

Therefore, this macro is designed to capture programming errors, not user or run-time errors, since it is generally disabled after a program exits its debugging phase.

Standard output and standard error

Apart from the cout you have used before, there is another output stream you can use call stardad error.

The cout put data into “standard output”, while the cerr put data into “standard error”.

```
#include <iostream>
using std::cout;
using std::cerr;
using std::endl;

int main() {

   cout<<"Hello world 1."<<endl;
   cerr<<"Hello world 2."<<endl;

return 0;
}
```

```
wo@DESKTOP-JSRF9TV:/mnt/d/CPP$ g++ main.cpp && ./a.out
Hello world 1.
Hello world 2.
```

Standard output and standard error

At first it does not make a difference. But you can redirect the streams.

In the example below, we redirect the standard output to “stdout.txt” and the standard error to “stderr.txt”

```
#include <iostream>
using std::cout;
using std::cerr;
using std::endl;

int main() {

   cout<<"Hello world 1."<<endl;
   cerr<<"Hello world 2."<<endl;

return 0;
}
```

```
wo@DESKTOP-JSRF9TV:/mnt/d/CPP$ g++ main.cpp && ./a.out
Hello world 1.
Hello world 2.
wo@DESKTOP-JSRF9TV:/mnt/d/CPP$ ./a.out 1> stdout.txt 2>stderr.txt
wo@DESKTOP-JSRF9TV:/mnt/d/CPP$
```

stdout.txt - 记事本

文件(F) 编辑(E) 格式(O) 查看(V) 帮助(H)

Hello world 1.

stderr.txt - 记事本

文件(F) 编辑(E) 格式(O) 查看(V) 帮助(H)

Hello world 2.

Standard output and standard error

Conventionally, we put the output of the program in standard output, and the error information, log information in standard error, even if the language does not require this.

```
#include <iostream>
using std::cout;
using std::cerr;
using std::endl;

int main() {

   cout<<"Hello world 1."<<endl;
   cerr<<"Hello world 2."<<endl;

return 0;
}
```

Standard output and standard error

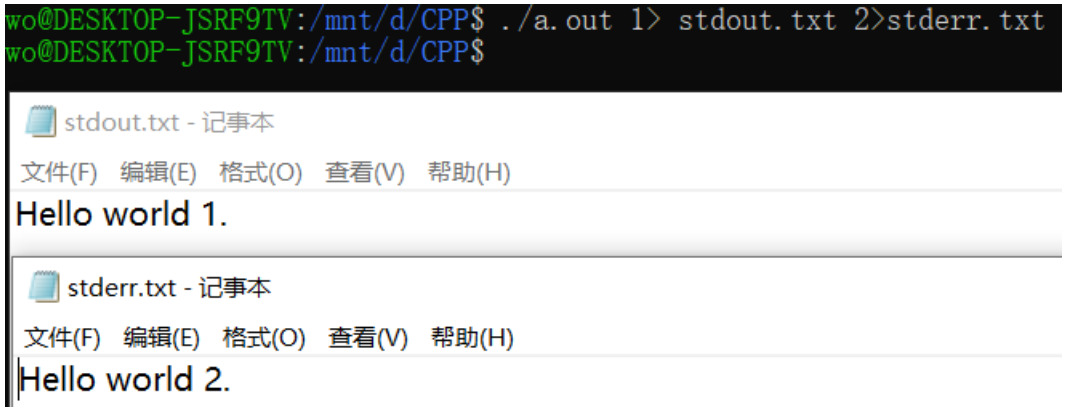
You can do the same with c language.

```
#include <cstdio>

int main() {

   fprintf(stdout, "Hello world 1.\n");
   fprintf(stderr, "Hello world 2.\n");

return 0;
}
```



You may need to search for “linux file descriptor table” to understand the basic idea behind these concepts.

Exercise 1

Write a function **calculateAverage()** which takes four int arguments which are marks for four courses in the semester and returns their average as a float.

The **calculateAverage()** function should take only valid range for marks which is between 0-100. If the marks are out of range throw an **OutOfRangeException** – define this exception as a class.

Invoke the **calculateAverage()** function in main function and get the following inputs and outputs:

```
Please enter marks for 4 courses:70 80 90 67
The average of the four courses is 76.75
Would you want to enter another marks for 4 courses(y/n)?y
Please enter marks for 4 courses:120 56 89 99
The parameter 1 is 120 which out of range(0-100).
Would you want to enter another marks for 4 courses(y/n)?y
Please enter marks for 4 courses:90 -87 67 92
The parameter 2 is -87 which out of range(0-100).
Would you want to enter another marks for 4 courses(y/n)?n
Bye, see you next time.
```

Exercise 2

Rewrite exercise 1, using assert instead of exceptions this time.