fit@hcmus

Object-Oriented Programming

Exception Handling

Tran Duy Hoang

What will be discussed?

- Introduction
- try-throw-catch
- RAII

Some ways to handle errors

- Terminate the program immediately
- Return a special value to represent that the program got some errors
- Return a normal value but change the state of the whole program to “error state”
- Invoke a certain function when there is any error

[1] terminate immediately

- It is not a good way to do because most of the times, we can handle the error and continue the program instead of just simply terminate the running program

[2] return a **special** value

- The **special** value is not always possible to represent. In some cases, the function might take all the range of the possible values. Thus, there is no special value to represent it.
- Also, you need to check it every time you invoke the function
- Or, the function may not have a return
  - E.g. constructors

An example

- You have to check every time → makes the program bigger and harder to maintain

```
int main()
{
   //...
   fd=open("file",O_RDWR);
   if(fd==-1)
      ...
}
```

[3] return a normal value but change the state of the program to “error state”

- The caller might not notice the program has been put into “error state”
- In C language, many libraries have used this method and change the global variable **errno** to a special value. It is hard to keep checking this value to know if there is an error.
- It is also not suitable for parallel processing applications

Exception handling

- It is a simple but powerful technique in C++ to help you handle errors.
- Exception handling allows you to separate the error handling section from the normal program

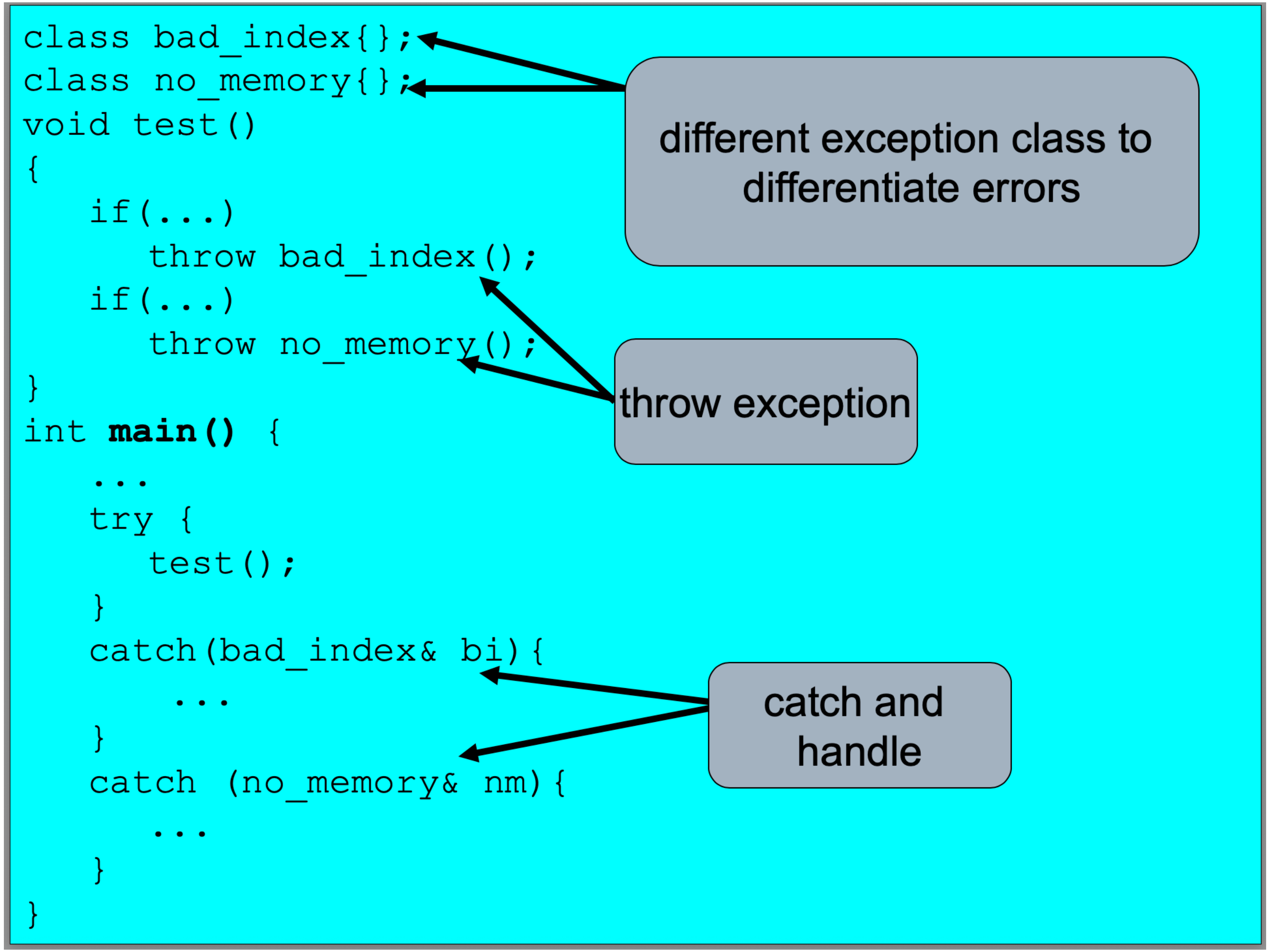
Exception handling

- C++ provides a mechanism via try-throw-catch to handle exception

```
void f1() {
   if(...)
      throw “something wrong”;
};
int main(){
   try {
      f1();
   }
   catch(char* s) {
      cout << “Error: "<< s << endl;
   }
   return 0;
};
```

An example: x*y/(x-y)

```
double tinh(double x, double y) {
   if(x == y)
      throw “divide by zero”;
   return x*y/(x-y);
}
int main(){
   double a, b;
   ...
   try {
      a = tinh(a, b);
   }
   catch(char* s) {
      cout << “Error: "<< s << endl;
   }
   return 0;
}
```



```
class bad_index{};
class no_memory{};
void test()
{
    if(...)
        throw bad_index();
    if(...)
        throw no_memory();
}
int main() {
    ...
    try {
        test();
    }
    catch(bad_index& bi){
         ...
    }
    catch (no_memory& nm){
        ...
    }
}
```

different exception class to differentiate errors

throw exception

catch and handle

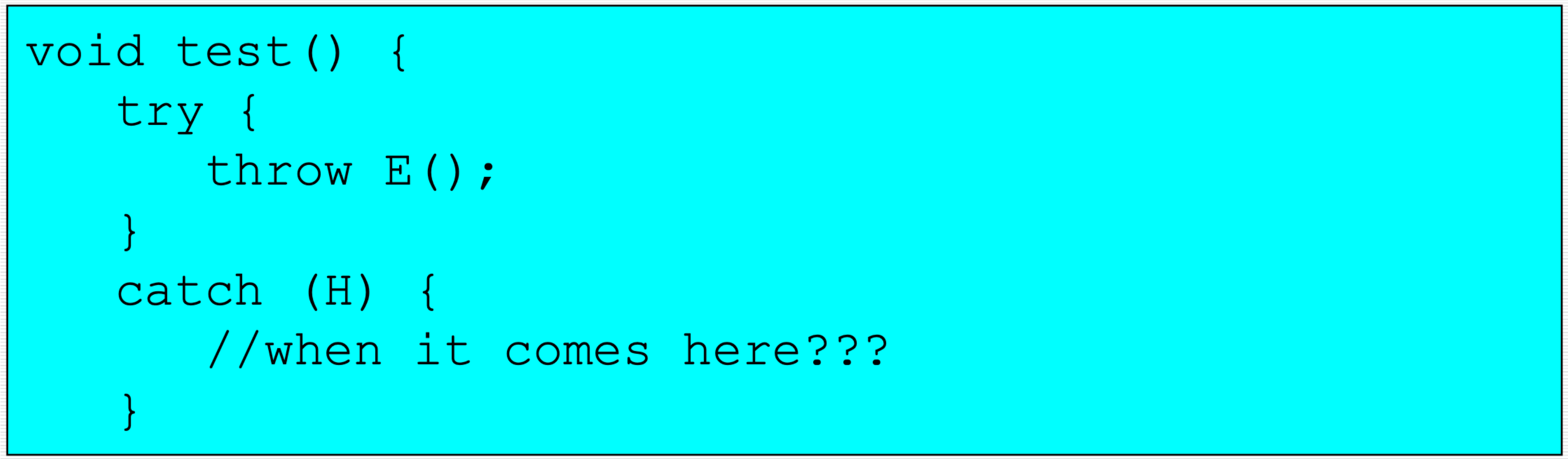
catch

- **catch** can access and change the value of the exception variables but all changes are just local within exception blocks (even passed by references)
- If **throw** in the **try{}** block doesn't return any value, the **catch** block will not be processed. Instead, the program will be terminated.

catch

- There must be at least 1 **catch** block right after each **try{}**
- **catch** has many arguments with their data types to receive the return values of **throw** from **try{}**.
- **catch** is only executed only when there is a **throw** with return value from **try{}**.

catch: matching algorithms



```
void test() {
   try {
      throw E();
   }
   catch (H) {
      //when it comes here???
   }
```

1. H has the same type as E
2. H is a base class of E
3. H & E are pointers and (1) or (2) satisfies
4. H is a reference and (1) or (2) satisfies

catch(...)

- **catch(…)** will catch any return values of throw
- It is often used as the last **catch** block to capture remaining exceptions.

catch

- Within the **catch** block, we can throw the exception to higher levels:
  - Throw with new operands with their data types
  - Throw with no operand. It means the catch throw the exception it received again to higher level.

After being throw

- If it couldn’t find a match catch block to the throw operand, the unwinding stack will be executed until there is a match catch block.
- If it still couldn’t find any match catch block, the program will be terminated.

throw declaration for a function

- By default: a function can throw anything
- To specify certain types of **throw** for a function, it is declared at the end of the function declaration
- For example:

```
int foo(int x) throw(char, int);
```

- If we declare `int foo(int x) throw();` the function is NOT expected to throw anything

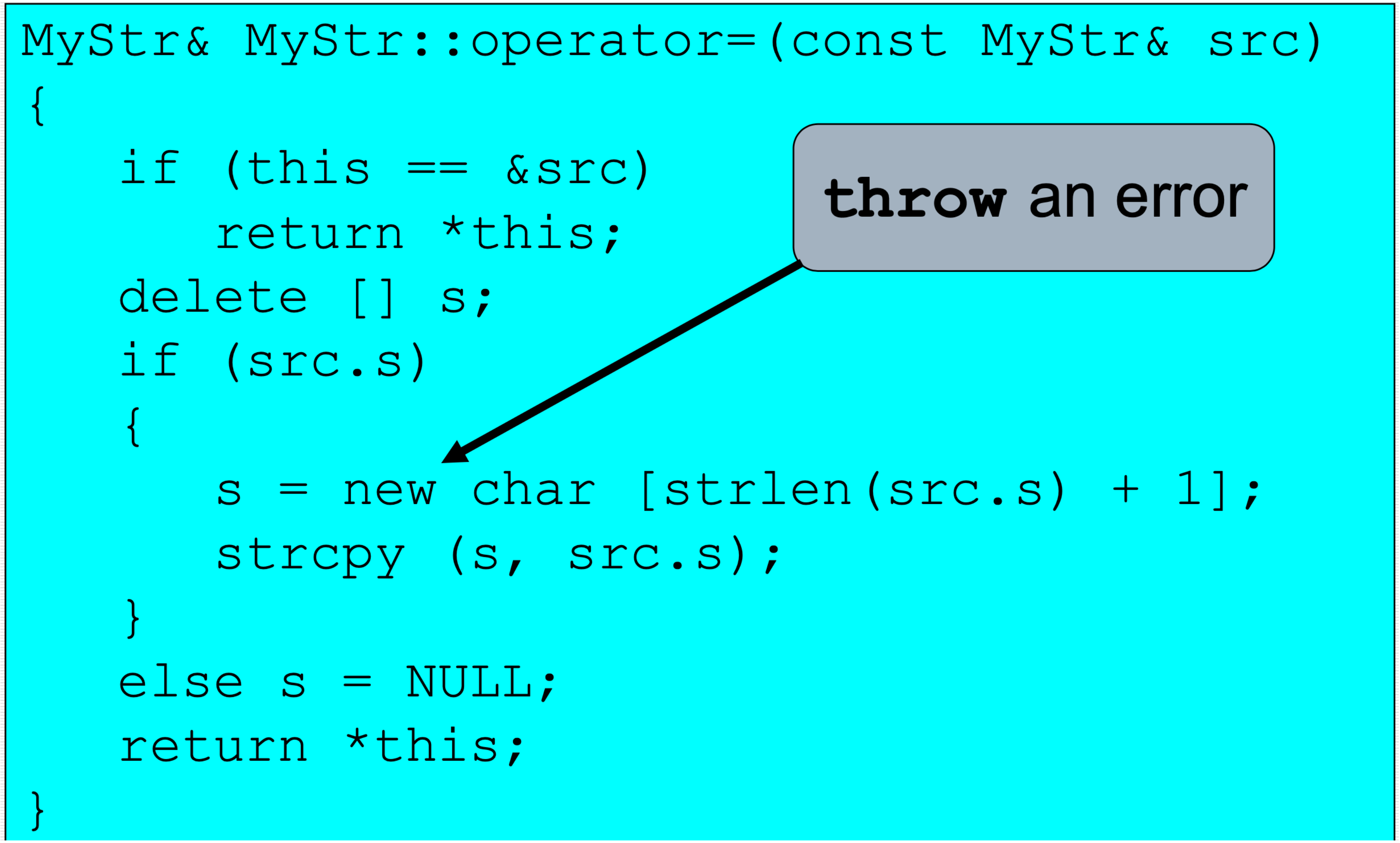
Some issues of exception handling

- Memory leak if we couldn’t handle resources properly.
- Exception handling does NOT work well with templates because template function might throw different exceptions based on different type parameters.

An example of memory leaking

```
int doSomething(int size)
{
   int* arrTest;
   arrTest = new int[size];
   ...
   if (condition)
     throw bad_exception();
   ...
   delete [] arrTest;
   return 0;
}
```

Another example



```
MyStr& MyStr::operator=(const MyStr& src)
{
   if (this == &src)
      return *this;
   delete [] s;
   if (src.s)
   {
      s = new char [strlen(src.s) + 1];
      strcpy (s, src.s);
   }
   else s = NULL;
   return *this;
}
```

`throw` an error

A fix for it

```
MyStr& MyStr::operator=(const MyStr& src) {
   if (this == &src)
      return *this;
   char* tmpS;
   if (src.s) {
      tmpS = new char [strlen(src.s) + 1];
      strcpy (tmpS, src.s);
   }
   else tmpS = NULL;
   delete [] s;
   s = tmpS;
   return *this;
}
```

Some questions!!!

- How can we handle if the constructors have errors/exceptions?
- How can we catch exceptions from initialization list?
- Nested **try{}** block
- Inheritance and polymorphism of exception classes?
- Why do we have **void pop()** for a stack?

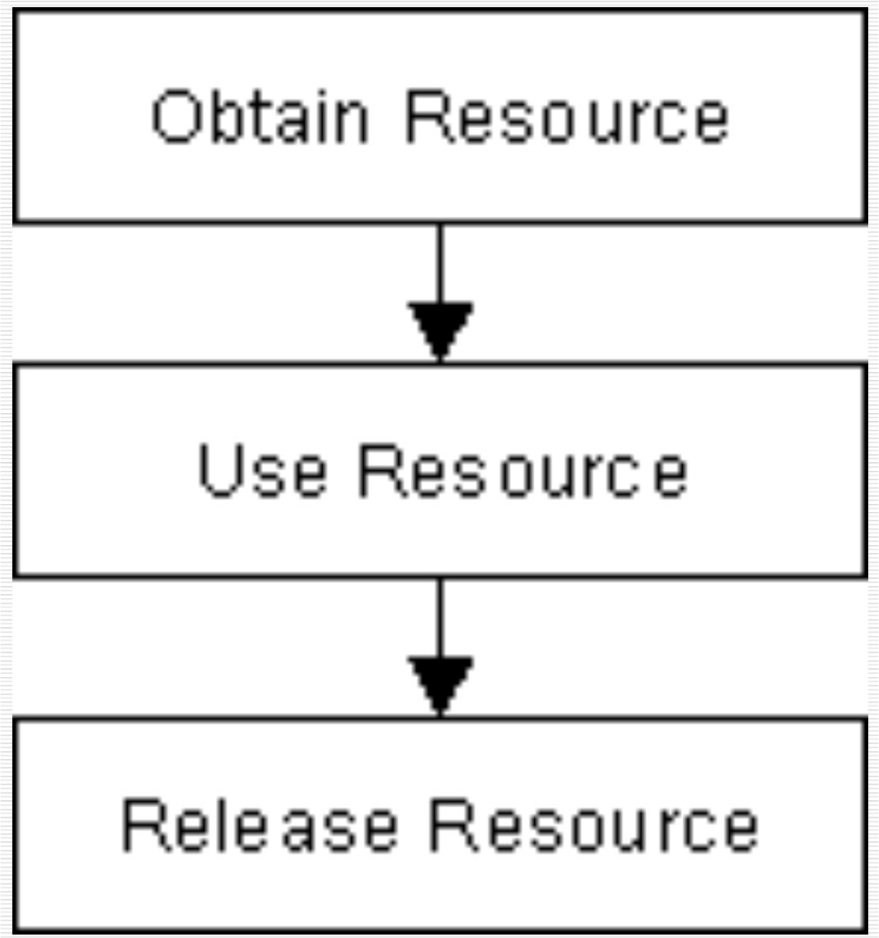
RAII: Resource Acquisition Is Initialization

- Invented by Bjarne Stroustrup to ensure that if a resource is used, it is released properly by attaching it into the life cycle of the object.
- RAII helps to write exception-safe code easier.

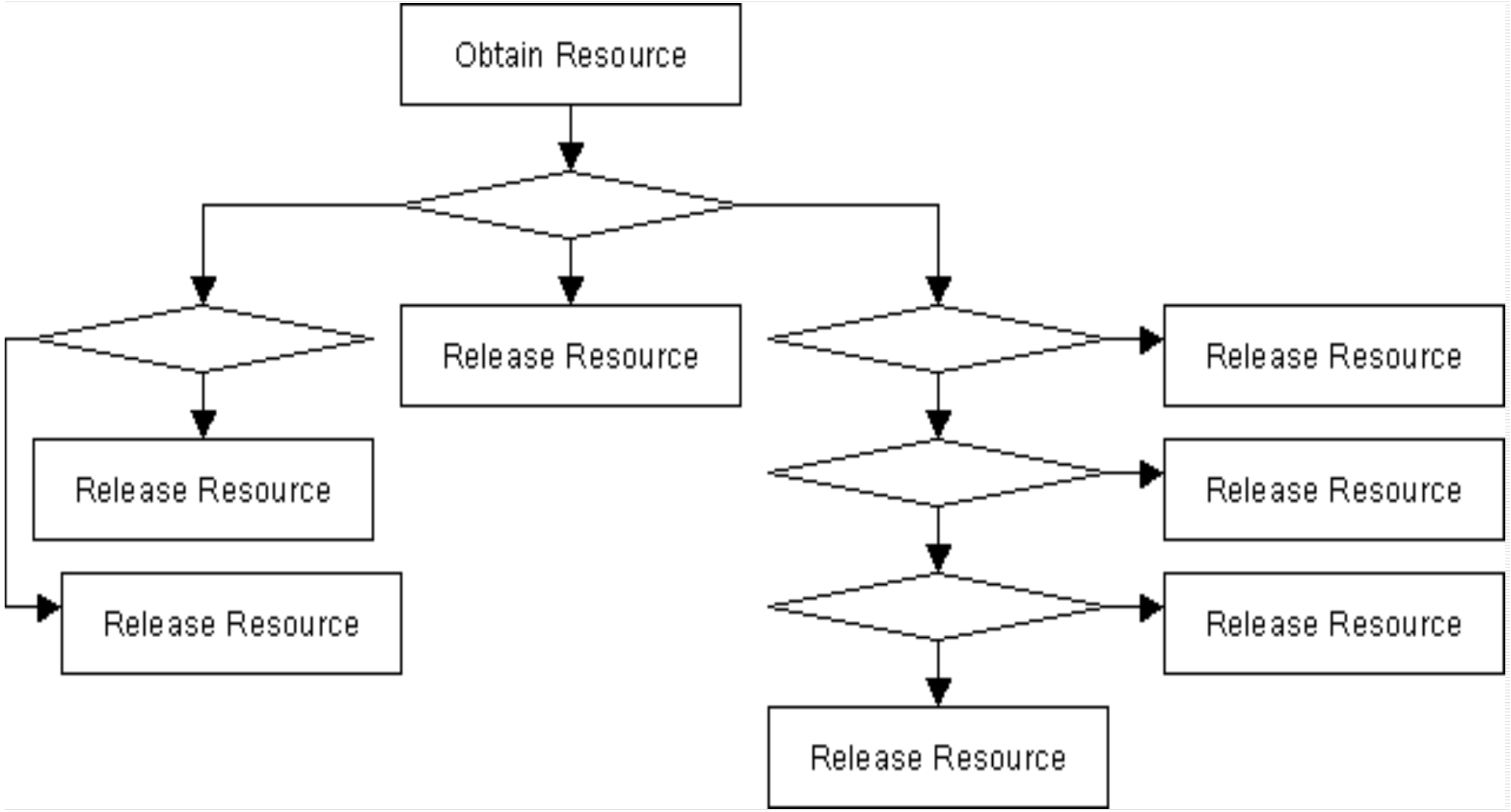
Main applications of RAII

- Often used to manage thread lock of multi-threading applications.
- Applications working with resources, such as dynamic memory allocating or file management to avoid leaking.

Problem



Problems become more complex



```
#include <cstdio>
class file {
public:
   file (const char* filename):
     f(std::fopen(filename, "w+")){
        if (!f)
         throw std::runtime_error("open failure");
   }
   ~file(){
          if (0 != std::fclose(f))
               {... } // handle it
   }
   void write (const char* str);
}
private:
   std::FILE* f;
   ...
};
```

Using the file class above

```
void example_usage()
{
    // open file (acquire resource)
    file logfile("logfile.txt");
    logfile.write("hello logfile!");
    // continue using logfile ...
    // throw exceptions or return
        // without worrying about closing the log;
    // it is closed automatically when out of scope
}
```

Exercise 9.1

Implement a UserAccount class with two attributes: username and password. Your program must validate inputs and use error handling to report problems (not silent true/false).

- Username rules: contains only alphabetic letters (A–Z, a–z) and has a maximum length of 30 characters.
- Password rules: length is 8 to 30 characters (inclusive), contains only letters and digits (A–Z, a–z, 0–9), and must include at least one uppercase letter and at least one digit.