CSCI251/CSCI851 Spring-2021 Advanced Programming (**S5c**)

Some miscellaneous topics

Outline

- Design patterns.
- Cohesion and coupling in brief.
- Exception handling Part 2.

Design patterns

- Design patterns are a means of exploiting recurring themes in code development.
  - Design patterns correspond to small conceptual ideas which are used as building blocks in the larger scale design process.
- They are not code themselves, rather they are concepts which can be deployed in a chosen language.

- We are just going to look at a couple of design patterns, and an implementation of each in C++.
  - Mainly for the purpose of illustrating the relevance of keywords and access specifiers.

Singletons

- The name is somewhat suggestive of the meaning.
- Typically there is no upper bound on the number of instances of a class, that is the number of objects of some ADT.
- But, we can only have one instance or object of a singleton class at a time.
- This class is realised by making the constructor private or protected.

```
class Singleton {
   private:
      static Singleton* instance;
   public:
      static Singleton* setInstance();
      static void Show(){cout << instance << endl;}
      static void TidyUp();
   protected:
      Singleton(){};
};
Singleton* Singleton :: instance = nullptr;
Singleton* Singleton :: setInstance(){
  if ( instance == nullptr )
    instance = new Singleton;
  return instance;
}
void Singleton :: TidyUp(){
  delete instance;
  instance = nullptr;
}
```

Allow only one instance.

Invokes the private constructor

Delete the sole instance,
and sets the pointer to `nullptr`.

```
int main()
{
  Singleton::Show();
  Singleton *eg = Singleton::setInstance();
//eg->setInstance();
  eg->Show();
  eg->TidyUp();
  eg = nullptr;
  Singleton::Show();
}
```

Monostate

- The name follows since all instances of the class share the same state.
- A monostate class contains **only** private static data members and public non-static member functions.
- The significance is that another class can be used to control changes to the monostate class, through any instance, and know that those changes will spread to all instances of the monostate class.
  - The different objects of the monostate class may correspond to use of the same information in different situations/locations.
    - High score table for a game…

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

class Admin;       // Forward referencing

class Lab{
    friend class Admin;
  private:
    static int labCapacity;
  public:
    int getLabCapacity() const
    {
        return labCapacity;
    }
};
int Lab :: labCapacity = 25;
```

```
class Admin
{
   private:
      bool permit;
      string passwd;
   public:
      Admin( string password );
      bool checkPassword( string password );
      void setLabCapacity( int cap );
};
```

It’s not a good idea storing passwords in this way …

```
Admin :: Admin( string password )
{
   if( passwd == "" )
     passwd = password;
}

bool Admin :: checkPassword( string password )
{
   if( passwd == password ) {
      cout << "Correct password" << endl;
      permit = true;
      return (true);
   }
   else {
      cout << "Incorrect password." << endl;
      return (false);
      permit = false;
   }
}

void Admin :: setLabCapacity(int numOfStudents){
  if ( permit )
    Lab::labCapacity = numOfStudents;
}
```

```
int main()
{
  Lab lab1, lab2;
  Admin admin1( "psd1234" );
  Admin admin2( "mypas" );

  cout << lab1.getLabCapacity() << endl;
  cout << lab2.getLabCapacity() << endl;

  if( admin1.checkPassword("psd1234") )
    admin1.setLabCapacity(30);


  cout << lab1.getLabCapacity() << endl;
  cout << lab2.getLabCapacity() << endl;


  if( admin2.checkPassword("abc111") )
    admin2.setLabCapacity(40);

  cout << lab1.getLabCapacity() << endl;
  cout << lab2.getLabCapacity() << endl;
}
```

```
25
25
Correct password
30
30
Incorrect password
30
30
```

Monostate vs singleton

- Both have only one set of data.
  - The monostate can have multiple references to that data though.
- They differ in transparency.
  - The monostate pattern is transparent, in the sense the user doesn’t need to modify their behaviour, they still create monostate class objects in the usual way.
  - Creating the singleton requires non-standard behaviour.

Coupling between functions/modules

- **Coupling** is the strength of the connection, or the level of dependency, between two modules.
  - **Tight coupling:** This means there is a lot of dependence between modules.
    - Tends to make programs more error prone.
    - Makes them more difficult to write, maintain, and reuse.
    - Data passing problems. More complicated, bad data.
    - Chance for functions to alter information others require.
  - **Loose coupling:** This means functions do not depend on others as much as in tight coupling.
    - We can change functions independently with greater ease, effectively saving time and money.

- While loose coupling can save money and time when changes occur, this needs to be balanced against the advantages obtained by implementing specific relationships.
- Wikipedia describes many types:

https://en.wikipedia.org/wiki/Coupling_(computer_programming)#Coupling_versus_cohesion

Cohesion within functions/modules

- **Cohesion:** Describes how well the operations in a function relate to one another.
- The best kind of cohesion is **Functional cohesion**, where all function operations contribute to the performance of only one task.

```
double square(double number)
    {
        return (number * number);
    }
```

- There are other weaker kinds of cohesion, the worst being coincidental.
- Wikipedia has a list:

https://en.wikipedia.org/wiki/Cohesion_(computer_science)

Exception handling: Part 2

- There are two uses of the term exception:
  - An *exception* is a situation that the code is unable to deal with.
    - This is in the sense of something occurring which is outside of normal expectations.
    - The exception needs to be communicated and dealt with.
    - These generally correspond to runtime errors, for example when the user enters invalid data.
  - An *exception* is also an object that is passed from the problem location to the location where the problem with be dealt with, or “handled.
    - This is the aspect we are going to focus on now.

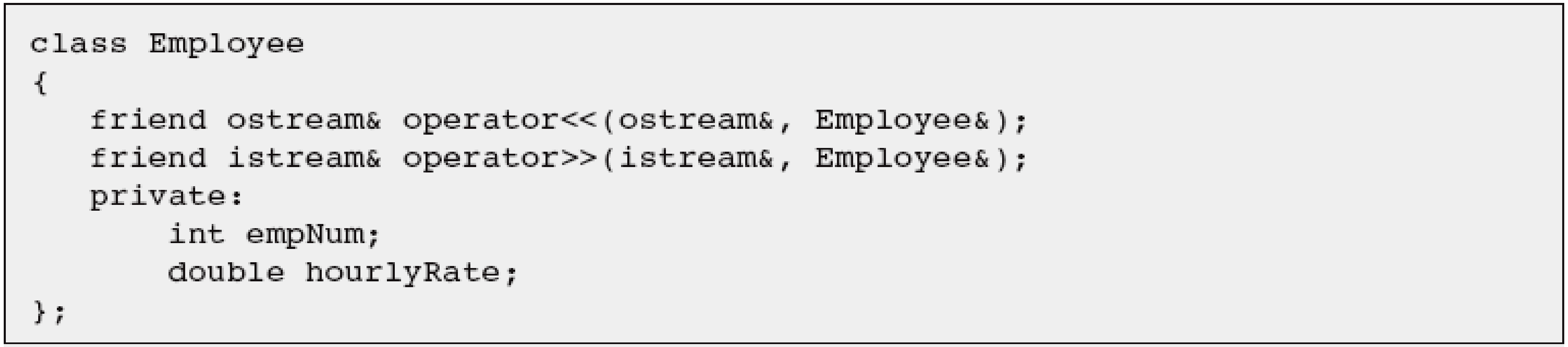
- A program is composed of separate modules, such as functions which may come from libraries.
- Error handling can be separated into two parts:
    - The reporting of error conditions that cannot be resolved locally.
    - The handling of errors detected elsewhere.
- For example:
  - The author of a library can detect runtime errors, but will not know what to do with them in the context of your program!
  - The user of the library must know how to cope with such errors, but cannot detect them easily.
- Exceptions in C++ are a means of separating error reporting from error handling, it's not supposed to replace ordinary in-function error checking/recovery.

Throwing Objects

- Just as simple variables such as `doubles`, `ints`, and `strings` can be thrown via exception-handling techniques, programmer-defined class objects can also be thrown.
  - These objects are called exception objects.
- This is particularly useful in two situations:
  - If a class object contains errors, you may want to throw the entire object, rather than just one data member or a `string` message.
  - When you want to throw two or more values, you can encapsulate them into a class object so that they can be thrown together.

Throwing Standard Class Objects

- The extraction operator for the following class has been overloaded to throw an exception.



```
class Employee
{
   friend ostream& operator<<(ostream&, Employee&);
   friend istream& operator>>(istream&, Employee&);
   private:
       int empNum;
       double hourlyRate;
};
```

Figure 12-16 The `Employee` class

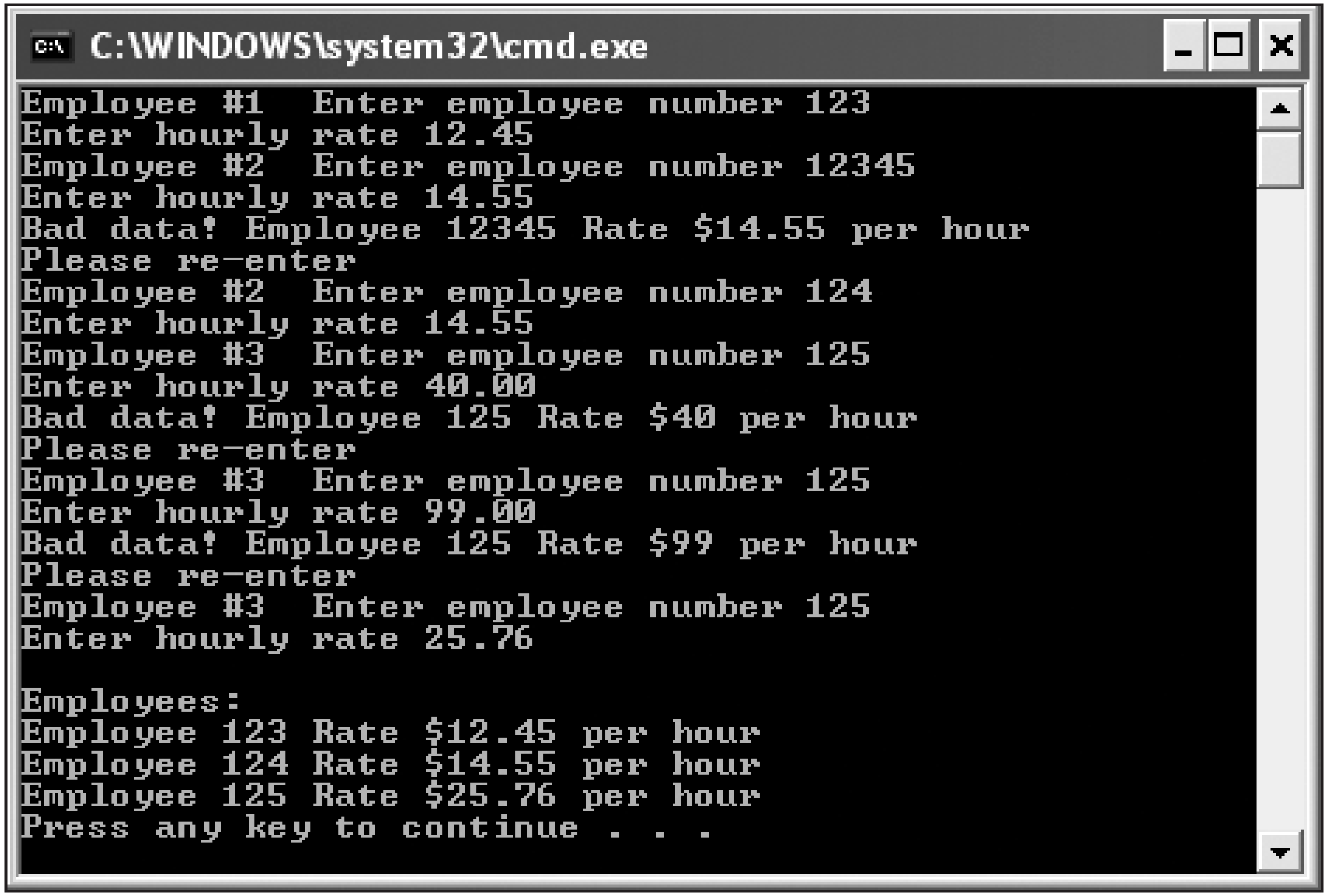
```
ostream& operator<<(ostream &out, Employee &emp)
{
   out<<”Employee “<<emp.empNum<<” Rate $”<<
      emp.hourlyRate<<” per hour”;
   return out;
}
istream& operator>>(istream &in, Employee &emp)
{
   const int LOWNUM = 100;
   const int HIGHNUM = 999;
   const double LOWPAY = 5.65;
   const double HIGHPAY = 39.99;
   cout<<”Enter employee number “;
   in>>emp.empNum;
   cout<<”Enter hourly rate “;
   in>>emp.hourlyRate;
   if(emp.empNum < LOWNUM || emp.empNum > HIGHNUM ||
      emp.hourlyRate < LOWPAY || emp.hourlyRate > HIGHPAY)
         throw(emp);
   return in;
}
```

Figure 12-16 The `Employee` class (continued)

- If either `empNum` or `hourlyRate` is too high or too low, the entire `Employee` object is thrown.

```
int main()
{
    const int NUM_EMPLOYEES = 3;
    Employee aWorker[NUM_EMPLOYEES];
    int x;
    for(x = 0; x < NUM_EMPLOYEES; ++x)
    {
        try
        {
            cout<<"Employee #"<<(x+1)<<"  ";
            cin>>aWorker[x];
        }
        catch(Employee emp)
        {
            cout<<"Bad data! "<<emp<<endl<<
               "Please re-enter"<<endl;
            --x;
        }
    }
    cout<<endl<<"Employees:"<<endl;
    for(x = 0; x < NUM_EMPLOYEES; ++x)
    cout<<aWorker[x]<<endl;
}
    cout<<endl<<"Employees:"<<endl;
    for(x = 0; x < NUM_EMPLOYEES; ++x)
       cout<<aWorker[x]<<endl;
}
```

Figure 12-17 A `main()` function that instantiates three `Employee` objects



```
Employee #1  Enter employee number 123
Enter hourly rate 12.45
Employee #2  Enter employee number 12345
Enter hourly rate 14.55
Bad data! Employee 12345 Rate $14.55 per hour
Please re-enter
Employee #2  Enter employee number 124
Enter hourly rate 14.55
Employee #3  Enter employee number 125
Enter hourly rate 40.00
Bad data! Employee 125 Rate $40 per hour
Please re-enter
Employee #3  Enter employee number 125
Enter hourly rate 99.00
Bad data! Employee 125 Rate $99 per hour
Please re-enter
Employee #3  Enter employee number 125
Enter hourly rate 25.76

Employees:
Employee 123 Rate $12.45 per hour
Employee 124 Rate $14.55 per hour
Employee 125 Rate $25.76 per hour
Press any key to continue . . .
```

Figure 12-18 Typical execution of the `main()` function in Figure 12-17

Bad data! statements appear in the catch block.

- The separation here allows for the calling function to distinguish between two cases:
  - An interactive problem with a present user so we can request the data be re-entered, as in the main() function a couple of slides back.
  - Input being taken from a data file.
    - Here we might just have to skip over that input line and report the problem to the source of the data.

- Here goes a class specifically for an exception, the instances are exception objects.

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

class dividebyzero {
public:
        dividebyzero();
        void printmessage();
private:
        const char* message;
};

dividebyzero::dividebyzero() : message("Divide by Zero"){}

void dividebyzero::printmessage()
{
        cout << message << endl;
}
```

```
float quotient(int num1, int num2){
if (num2 == 0)
      throw dividebyzero();
return (float) num1 / num2;
}

int main(){
      int a, b;
      cout << "Enter two numbers : ";
      cin >> a >> b;
      try
      {
            float x = quotient(a,b);
            cout << "Quotient : " << x << endl;
      }
      catch (dividebyzero error)
      {
            error.printmessage();
            return 1;
      }
      return 0;
}
```

Here we have a function which throws an exception object of type `dividebyzero`.

When the exception is thrown an instance of this object is passed to the handler.

The thrown thing must be an object. The seemingly direct call to the constructor isn't allowed for initiation of an object normally but is when you are passing it straight on → copy elision.

Exception specifications

- Any C++ function might throw any type of object.
- You can explicitly indicate the exceptions that a function cannot throw by using **exception specifications**, as follows:

```
int dataEntry() throw()
int dataEntry() noexcept
int dataEntry() noexcept( true )
```

- This can help users of your code.

Ye Olde Exception Specifications

- Pre C++11 you could have things like …

```
int dataEntry() throw(char, double, Employee)
```

- This could throw objects of type char, double, `Employee`, or an object of a child class of `Employee`.
- If a function threw an error whose type was not listed in its exception specification, then a runtime error would occur and the program will abort.
- But it was pretty much only for documentation purposes, and the compiler wouldn’t stop you if you threw different things.
- So, it’s deprecated.

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

void test() throw(int)
{
      cout << "Hello" << endl;
      throw 5;
      throw string("Hello");
}

int main()
{
      try {
        test();
      }
      catch (int x){
            cout << x << endl;}
      catch (string y){
            cout << y << endl;}
      cout << "Hello" << endl;
}
```

We can try this with throwing a string or int. And changing `throw(int)` to `throw()` and `noexcept`.

Note how we just continue after the exception is caught.

`throw()` and `noexcept`

- In C++11 `throw()` was deprecated so shouldn't be used with `-std=c++11`, although it will still work.
  - Since C++11, destructors are `noexcept` by default. They shouldn't throw exceptions.
- `throw()` is back in C++17 and is equivalent to `noexcept(true).` ☺
  - And you should only use `throw()` or `noexcept` now.
- See http://en.cppreference.com/w/cpp/language/noexcept_spec

The `noexcept` operator

- New to C++11.
- Returns a Boolean indicating if we can throw exceptions within a function.
- So if we have …

```
int dataEntry() noexcept
```

- … then we would get true from …

```
noexcept(dataEntry())
```

- … which we might use in …

```
int moreDataEntry noexcept(noexcept(dataEntry());
```

- If an exception not listed in the exception specification is thrown, the function `void unexpected(void);` is called, defined in `<exception>`, part of the standard namespace.
- By default, `unexpected()` calls `void terminate( void );`
- In Unix, `terminate()` calls the standard C Library `abort()`.
- You can change the behavior of `unexpected()` and `terminate()` by setting your own functions.

```
unexpected_function set_unexpected( unexpected_function unexp_func );
terminate_function set_terminate( terminate_function term_func );
```

`unexpected_function` and `terminate_function` are types.

The custom `unexp_func` needs to return `void`.

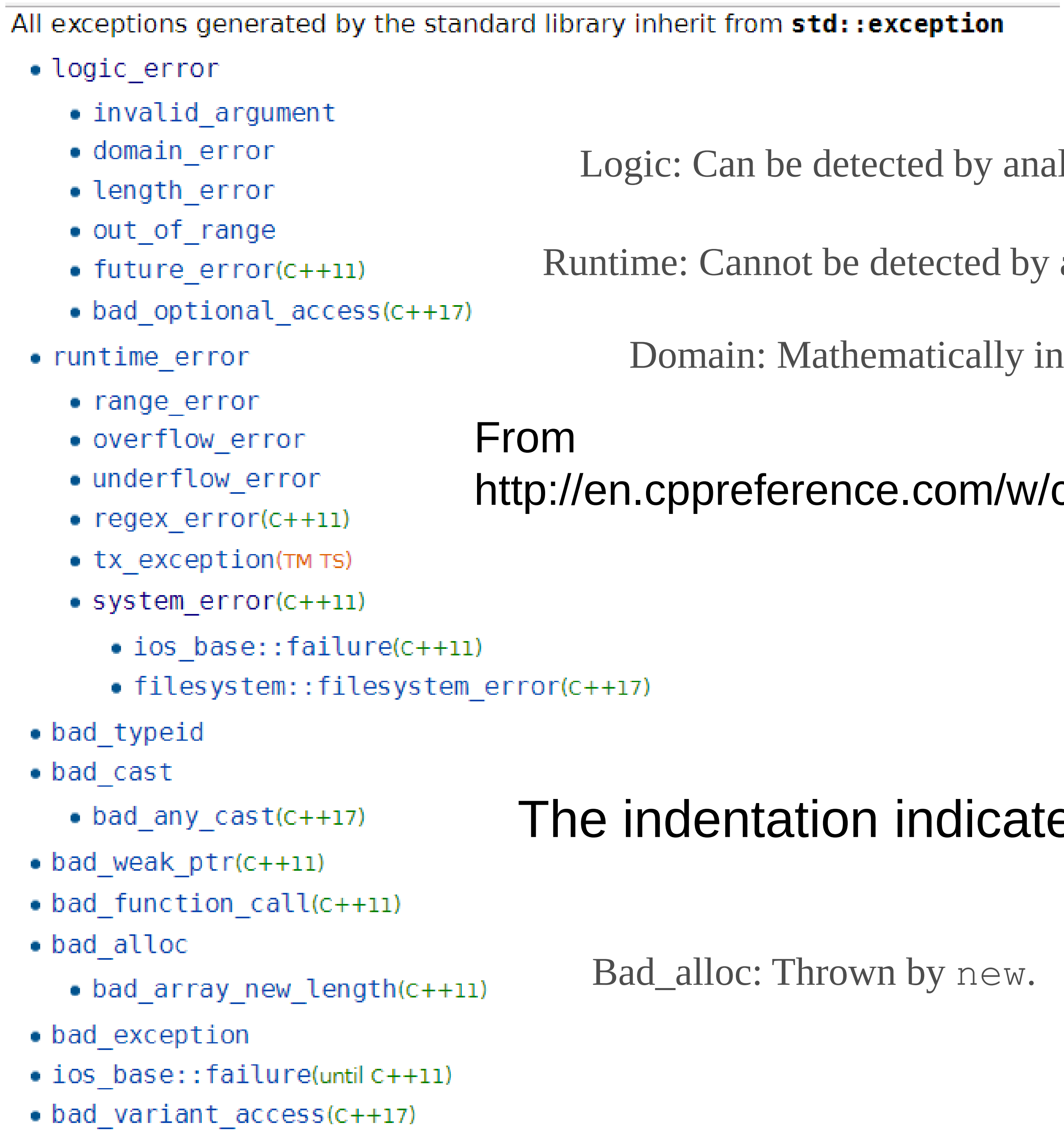
0..* catches …

- The first to match is used.
- So, if you need to throw both a base class object and an object that is a member of its derived class from the same function, and you want to carry out different operations when they are caught, then you must code the `catch` for the derived object first.
  - This is because the derived object is also a base object so would against the first case.
- The `catch` type also catches objects differing only in a `const` qualifier, a reference qualifier or both qualifiers:
  - For example, `char` can be caught by `char`, `const char`, `char&` or `const char&`.

- If you throw an argument and no `catch` block has a usable match, then the program terminates.
- To avoid termination, you can code a **default exception handler** that catches any type of object not previously caught.
  - Create a default exception handler by creating a `catch` block with an ellipsis (…) as its argument.
  - It must be the last `catch` block listed after a `try`.
  - A default `catch` block will catch any thrown object that has not been caught by an earlier `catch` block.
- You might also code a default catch block to handle several exception types the same way.

Standard exceptions…

- Some functions of the standard C++ language library send exceptions that can be captured if we include them within a **try** block.
- These exceptions are sent with a class derived from **`std::exception`**.
- The class `std::exception` is defined in the C++ standard header file `<exception>` and serves as a pattern for the standard hierarchy of exceptions.
- For C++11, so CC –std=c++11, it’s in `<stdexcept>`.



All exceptions generated by the standard library inherit from **`std::exception`**

- `logic_error`
    - `invalid_argument`
    - `domain_error`
    - `length_error`
    - `out_of_range`
    - `future_error`(C++11)
    - `bad_optional_access`(C++17)
- `runtime_error`
    - `range_error`
    - `overflow_error`
    - `underflow_error`
    - `regex_error`(C++11)
    - `tx_exception`(TM TS)
    - `system_error`(C++11)
        - `ios_base::failure`(C++11)
        - `filesystem::filesystem_error`(C++17)
- `bad_typeid`
- `bad_cast`
    - `bad_any_cast`(C++17)
- `bad_weak_ptr`(C++11)
- `bad_function_call`(C++11)
- `bad_alloc`
    - `bad_array_new_length`(C++11)
- `bad_exception`
- `ios_base::failure`(until C++11)
- `bad_variant_access`(C++17)

Logic: Can be detected by analysing the code.

Runtime: Cannot be detected by analysing the code.

Domain: Mathematically invalid parameters.

From http://en.cppreference.com/w/cpp/error/exception

The indentation indicates inheritance.

Bad_alloc: Thrown by `new`.

exception & bad_exception classes

C++11: Commented lines are removed, throw() → noexcept

```
namespace std {
    class exception {
    public:
        exception() throw() {}
//      exception(const exception&) throw() {}
//      exception& operator=(const exception&) throw()
//           {return *this;}
        virtual ~exception() throw() {}
        virtual const char* what() const throw();
    };

    class bad_exception: public exception {
    public:
        bad_exception() throw() {}
//      bad_exception(const bad_exception&) throw() {}
//      bad_exception& operator=(const bad_exception&) throw()
//           {return *this;}
//      ~bad_exception() throw() {}
        const char* what() const throw();
    };
}
```

```
#include <iostream>
#include <exception>
#include <typeinfo>
using namespace std;

class A {virtual void f() {}; };
int main ()
{
  try {
     A* a = NULL;
     typeid (*a);
  }
  catch (exception& e)
  {
     cout << "Exception: " << e.what() << endl;
  }

  return 0;
}
```

```
Exception: Bad typeid
```

Handling memory-allocation exceptions

- When you try to allocate memory, there may not be enough memory available on your computer:
  - If the `new` operator fails, the program ends abruptly.
- The **`set_new_handler()` function** is an out-of-memory exception handler; to use it:
  - Insert `#include<new>` at the top of the program file.
  - Create a function to handle the error, then pass that error-handling function's name (pointer to the function) to the `set_new_handler()` function:

```
set_new_handler(nameOfErrorHandlingFunction);
```

- Return type must be `void`
- Must not take any arguments

```
#include<iostream>
#include<new>
using namespace std;
void handleMemoryDepletion() {
  cout << "Out of memory" << endl;
  exit(1);
}
int main() {
  int num = 10000000;
  set_new_handler( handleMemoryDepletion );
  const int LIMIT=30;

  for(int i=0; i<LIMIT; i++) {
      double *dp = new double[ num ];
      cout << "i is " << i << endl;
  }
  return 0;
}
```

Common practice …

- Throw an object, catch a reference.
- C++ allows throwing exceptions of any data type, but that isn't recommended.
- It's preferable to create a subclass of `exception` for your program.
  - Easier for OO programmers to understand.
  - A commonly used way to handle errors.
  - Can use: `catch(exception& e)`
    - Better than `catch(…)`
    - Informative message can be returned from `what()` tailored for your exception class.

```
#include <iostream>
#include <exception>

class MyException : public std::exception
{
    const char * what () const throw ()
        {return "MyException instance";}
}


int main()
{
    try
    {
        throw MyException();
    }
    catch (MyException& e)
    {
        std::cout << "MyException caught" << std::endl;
         std::cout << e.what() << std::endl;
    }
    catch (std::exception& e)
    {
        // Other errors
    }
}
```

Based on http://peterforgacs.github.io/2017/06/25/Custom-C-Exceptions-For-Beginners/

Building an `exception` derived class

```
class BoundException  :  public runtime_error {
    private:
        int lower;
        int higher;
    public:
        BoundException( int l, int h );
        int getLower() const;
        int getHigher() const;
};
BoundException :: BoundException(int l, int h) :
                            runtime_error("Out of bound") {
     lower = l;
     higher = h;
}
int BoundException :: getLower() const {
     return lower;
}
int BoundException :: getHigher() const {
     return higher;
}
```

Pass a string to be stored in the parent class.

```
int inputData( int l, int h )
{
    BoundException error( l, h );
    int tmp;

    cout << "Input data in the range [" << l << ", " << h << "] : ";
    cin >> tmp;

    if( tmp < error.getLower() || tmp > error.getHigher() )
        throw( error );

    return tmp;
}
```

```
int main()
{
    int num;

    try {
        num = inputData(-50, 50);
        double sq = sqrt(num);
        cout << "sqrt of " << num << " is: " << sq << endl;
    }
    catch( logic_error& err ) {
        cout << "logic_error: " << err.what() << endl;
    }
    catch( BoundException& berr ) {
        cout << "BoundException: " << berr.what() << endl;
    }
}
```

Include `<stdexcept>` and `<cmath>`.

`what()` inherited from `runtime_error`

Specifying our own `what()`

```
class BoundException : public runtime_error {
    private:
        int lower;
        int higher;
    public:
        BoundException( int, int );
        int getLower() const;
        int getHigher() const;
        virtual const char* what() const noexcept;
};
```

```
const char* BoundException :: what() const noexcept{
     // ... operations specific to BoundException
     return ( runtime_error::what() );
}
```

Get the string stored in the parent class

- The following has some useful notes on exceptions …

https://www.acodersjourney.com/top-15-c-exception-handling-mistakes-avoid/