CSCI251
Advanced Programming
Creating Library in C++

UNIVERSITY OF WOLLONGONG AUSTRALIA

Outline

Object Oriented Programming in C++

Static library

1. What is it?
2. Code - Practice

Shared library

3. Why shared?
4. Code - Practice

Exception handling

5. Why exception?
6. Two demo

Enumerations

7. Concept of enum
8. Code-demo

C++ Library - Static

What are .dll and .so files?

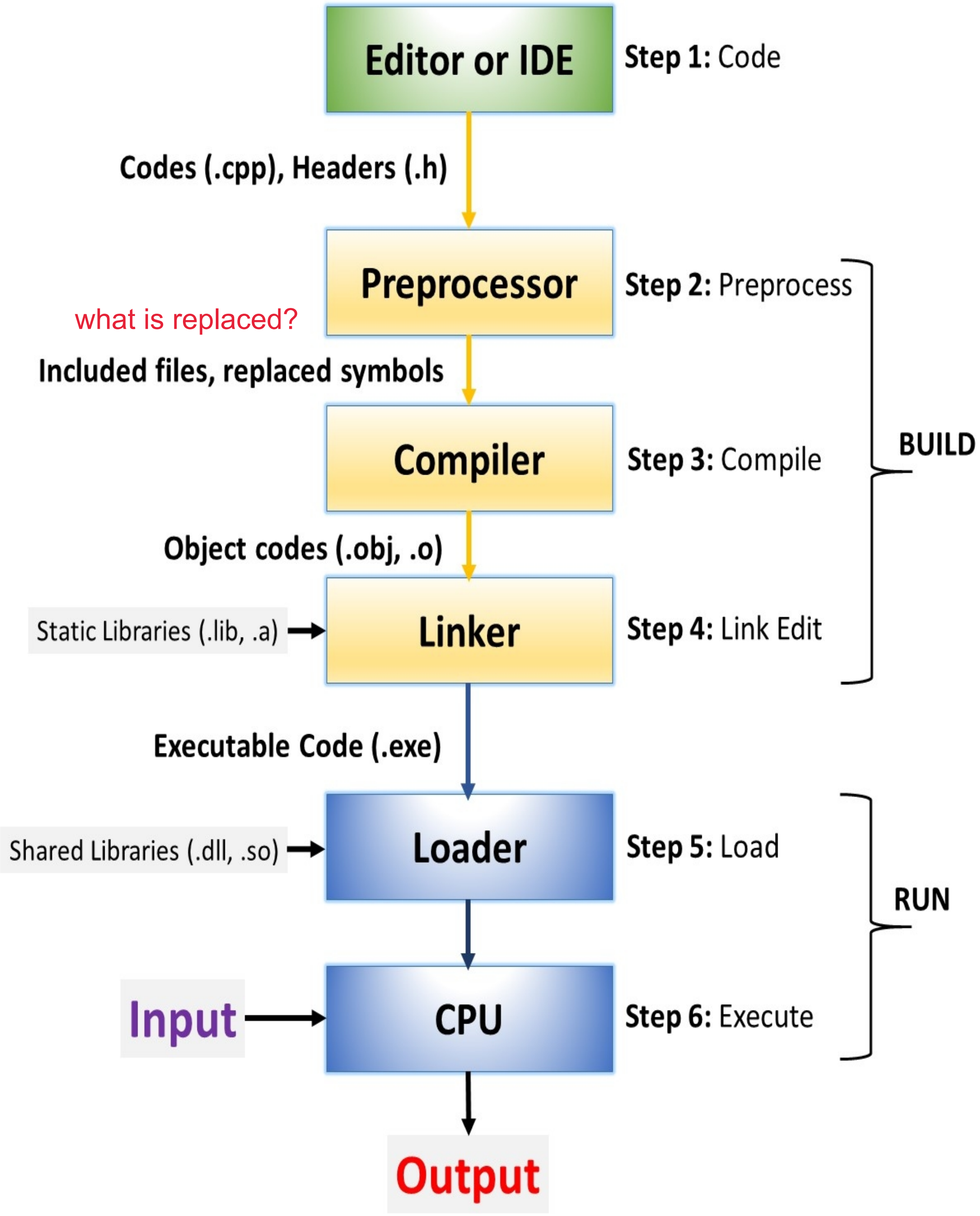
Where are they located?

.dll and .so

- What is dll: dll stands for Dynamic Link Library. It is a dynamic library file
- Where you can find it: in Windows OS (in program folders)

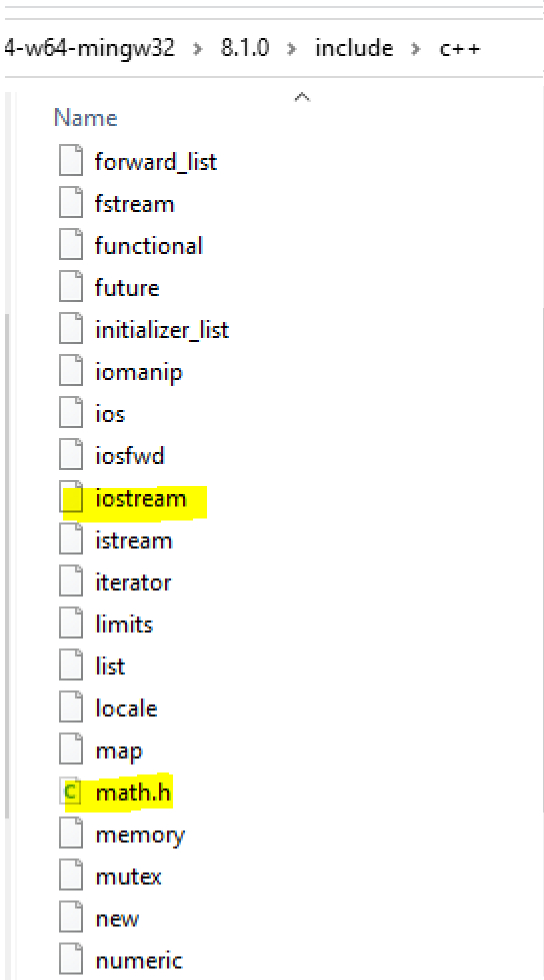
- What is so: so stands for Shared Object. It is a dynamic library file
- Where you can find it: in Linux or Mac OS

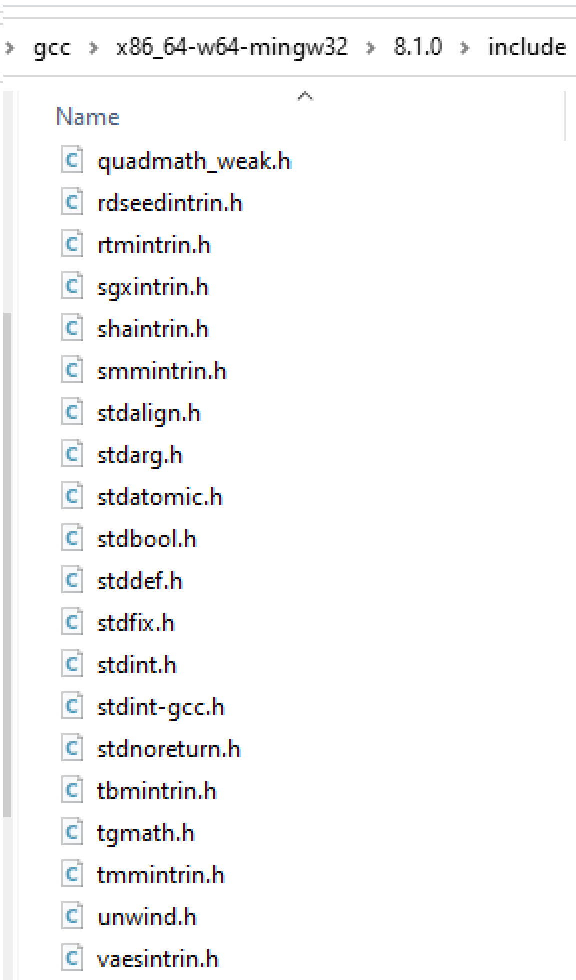
Writing Process



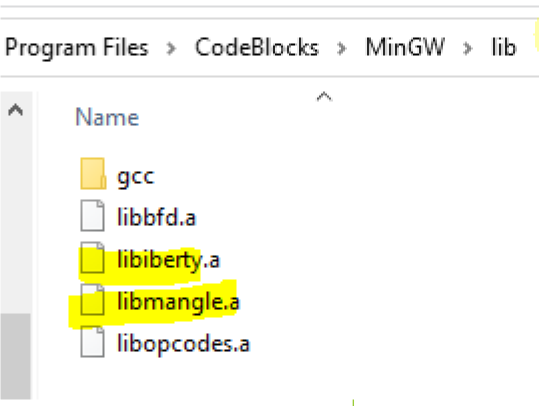

```cpp
/*
 * Ask the user for the length and width of a retangle
 * and compute its perimeter and area.
 * (retangle.cpp)
 */
#include <iostream>
using namespace std;

int main() {
   int length;                   // Declare 1 integer variable
   int width, perimeter, area;   // Declare 2 integer variables in one state

   cout << "Compute the Perimeter and Area of Rectangle" << endl;
   cout << "Please enter the length: ";   // Prompting message
   cin >> length;                         // Read input into variable length
   cout << "Please enter the width:  ";   // Prompting message
   cin >> width;                          // Read input into variable width

   // Compute perimeter and area
   perimeter = 2*(length + width);
   area = length * width;

   // Print the results to the screen
   cout << "\nThe result:" << endl;
   cout << "The perimeter is: " << perimeter << "m" << endl;
   cout << "The area is:      " << area << "m2" << endl;

   return 0;
}
```

C++ Library: Inside Code:Blocks code

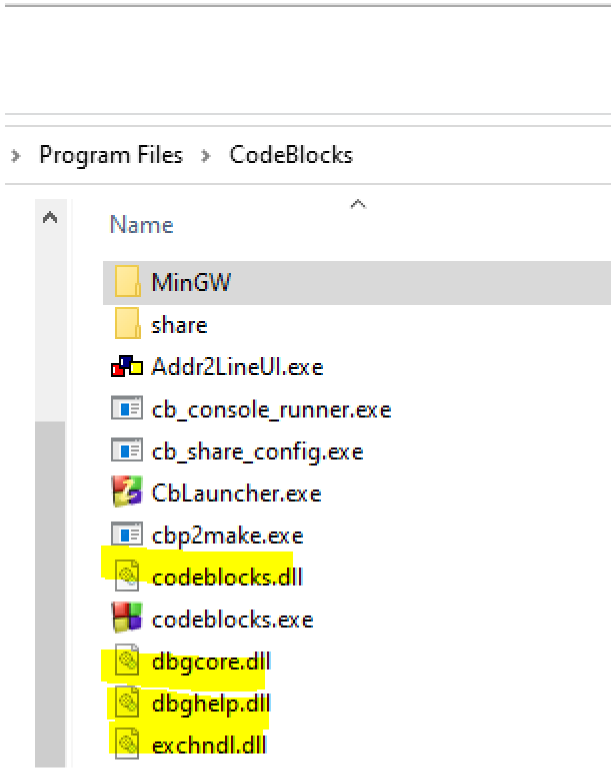




Preprocess-header



Static-.lib, .a



Shared-.dll, .so

What is C++ library

A library is a package of code that is meant to be reused by many programs. Typically, a C++ library comes in two pieces:

- A header file that defines the functionality the library is exposing (offering) to the programs using it.
- A precompiled binary that contains the implementation of that functionality pre-compiled into machine language.

Why are libraries pre-compiled?

First, save resources: libraries rarely change, they do not need to be recompiled often.

Second, improve security: precompiled objects are in machine language, it prevents people from accessing or changing the source code.

Types of Library

- There are two major types of library, and the type influences linkage:
- Static Libraries:
  - A static library is just a collection of .o files concatenated together.
  - They are compiled and linked directly into your program. When you compile a program that uses a static library, all the functionality of the static library that your program uses becomes part of your executable.

Types of Library

- On Windows, static libraries typically have a .lib extension, whereas on linux, static libraries typically have an .a (archive) extension. One advantage of static libraries is that you only have to distribute the executable in order for users to run your program.
- A copy of the library becomes part of every executable that uses it, this can cause a lot of wasted space. Static libraries also cannot be upgraded easily -- to update the library, the entire executable needs to be replaced.

Types of Library

- Dynamic or Shared Libraries:
  - ✓ They are loaded into your application at run time. It does not become part of your executable -- it remains as a separate unit.
  - ✓ On Windows, dynamic libraries typically have a .dll (dynamic link library) extension, whereas on Linux, dynamic libraries typically have a .so (shared object) extension.

Types of Library

- Advantages of Shared Library
  - Many programs can share one copy, which saves space.
  - Dynamic library can be upgraded to a newer version without replacing all of the executables that use it.

Separation: h vs cpp

- We need to tell “clients” about how to use the functions and classes we write, but they don’t see the details of our implementations.
- So we can put the definition in the header and leave the implementation to the cpp file, and the cpp code can be pre-compiled to save time.

Structure of a library

- A library has two parts:
  - The library itself.
    - Depending on whether it is a static library or shared library it will have a different extension, typically .a or .so.
  - The header file describing the entry points/ function calls and variables in the library which are publicly accessible.
- A library does not have a main function, it's not a complete program.

Building a static library

Building a static library ...

One way to build a static library is to generate the object file and use the archiver.

The archiver will combine object files to form a library OR an archive:

```
$ g++ -c code.cpp
$ ar -crv libcode.a code.o
```

The resultant library is libcode.a.

Libraries are generally prefixed with lib.

To check what a library contains, in terms of the object files inside it, use the –t flag.

```
$ ar -t libcode.a
code.o
```

Using the static library

- ✓ To use `libcode.a` in other programs you would include the header `code.h` and link against `libcode.a`.
- ✓ We can link it in using …

```
$ g++ driver.cpp libcode.a -o program
```

- ✓ The `.a` file doesn't need to be compiled, just linked in after the object files for the cpp files, here just `driver.cpp`, have been generated.
- ✓ We don't need `libcode.a` around when we run our program.

Code – Static

Time complexity?

time.cpp

```
#include <iostream>
#include <chrono>
#include "fibfun.h"

int main()
{
    auto start = std::chrono::system_clock::now();
    std::cout << "f(35) = " << fibonacci(35) << '\n';
    auto end = std::chrono::system_clock::now();

    std::chrono::duration<double> elapsed_seconds = end-start;
    std::time_t end_time = std::chrono::system_clock::to_time_t(
end);

    std::cout << "finished computation at " << std::ctime(&end_t
ime)
              << "elapsed time: " << elapsed_seconds.count() <<
"s\n";
}
```

```
$ g++ -c fibfun.cpp
$ ar -crv libfib.a fibfun.o
$ g++ time.cpp libfib.a -o time1
```

fibfun.h

```
long fibonacci(unsigned n
);
```

fibfun.cpp

```
long fibonacci(unsigned n)
{
    if (n < 2)
        return n;
    return fibonacci(n-1) + fibonacci(n-
2);
}
```

```
ls -l time1
```

Size: time1   15168

Practice 1: Static Library

- Do on capa system
- Create 3 files: `main.cpp`, `add.h`, and `add.cpp`
  - `add.h` declare an add function type int with 2 int parameter: `int add(int, int);`
  - `add.cpp` defines add function that returns the sum of 2 parameters: `int add(int a, int b){return a+b;}`
  - `main.cpp` include `add.h` file then call add function and print to the sum of your two numbers: `cout<<add(10, 14);`
- Create static library
  - `$ g++ -c add.cpp`
  - `$ ar -crv libadd.a add.o`
  - `$ g++ main.cpp libadd.a -o add1`
- Run executable file and show the size
  - `./add1`
  - `ls -l add1`

Building a shared library

Building a shared library

- Dynamic libraries are built in the similar way except we use `-shared` as the link directive - these are passed `to ld`.
- This tells the compiler we want a shared library.
- We can replace dynamic content without requiring recompilation of the rest.
- To build the shared library with the GNU compiler you must use the use `-fpic` directive which produces position independent output.

```
g++ -fpic -shared -o libcode.so code.cpp
```

Using a library...

- Let us consider a prebuilt library called `libcode.so`.
- The header `code.h` and shared object `libcode.so` reside in your current home directory.
- You have a file called `main.cpp` which calls a function in `libcode.so` called code.
- How do you compile it and link it against the library?

Using a library…

- We need to set up the library path to the current directory (.), or wherever we store our library…,

  $ LD_LIBRARY_PATH=.

  $ export LD_LIBRARY_PATH
- And then …

  $ g++ -I. -L. main.cpp -lcode
- The code is for the library short name…

Using a library...

- This directive means:

`-I` headers can be found in the current directory along with the default system directory.

`-L` libraries can be found in the current directory along with the default system directory.

`-lcode`, link against `libcode.so` which is in the current directory.

Without `-L`. it won't be found.

Code – Dynamic

time.cpp

```
#include <iostream>
#include <chrono>
#include "fibfun.h"

int main()
{
    auto start = std::chrono::system_clock::now();
    std::cout << "f(35) = " << fibonacci(35) << '\n';
    auto end = std::chrono::system_clock::now();

    std::chrono::duration<double> elapsed_seconds = end-start;
    std::time_t end_time = std::chrono::system_clock::to_time_t(
end);

    std::cout << "finished computation at " << std::ctime(&end_t
ime)
              << "elapsed time: " << elapsed_seconds.count() <<
"s\n";
}
```

fibfun.h

```
long fibonacci(unsigned n
);
```

fibfun.cpp

```
long fibonacci(unsigned n)
{
    if (n < 2)
        return n;
    return fibonacci(n-1) + fibonacci(n-
2);
}
```

A dynamic library

```
$ g++ -fpic -shared -o libfib.so
fibfun.cpp
$ g++ -I. -L. time.cpp -lfib -o time2
$ ./time2
```

Size: time2   15128

Code – Dynamic

A dynamic library – Should be like this

Size: time2 15128

```
$ LD_LIBRARY_PATH=.
$ export LD_LIBRARY_PATH
$ g++ -fpic -shared -o libfib.so
fibfun.cpp
$ g++ -I. -L. time.cpp -lfib -o time2
$ ./time2
```

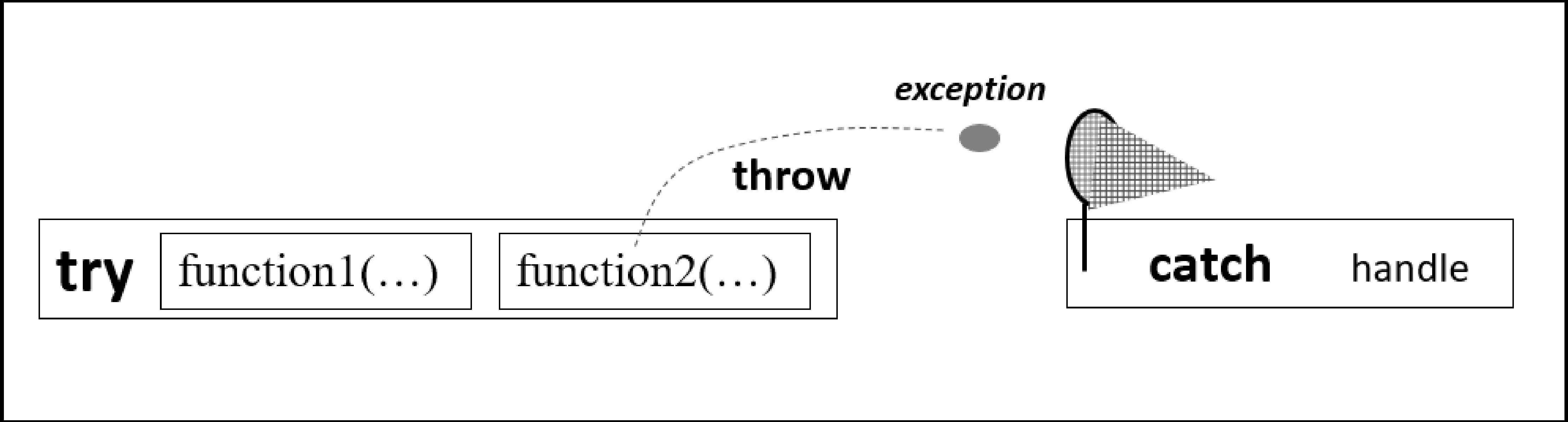
Practice 2: Dynamic Library

- Do on capa system
- (did it) Create 3 files: `main.cpp`, `add.h`, and `add.cpp`
  - `add.h` declare an add function type int with 2 int parameter: `int add(int, int);`
  - `add.cpp` defines add function that returns the sum of 2 parameters: `int add(int a, int b){return a+b;}`
  - `main.cpp` include `add.h` file then call add function and print to the sum of your two numbers: `cout<<add(10, 14);`
- Create dynamic library
  - `$ LD_LIBRARY_PATH=.`
  - `$ export LD_LIBRARY_PATH`
  - `$ g++ -fpic -shared -o libadd.so add.cpp`
  - `$ g++ -I. -L. main.cpp -ladd -o add2`
- Run executable file and show the size
  - `./add2`
  - `ls -l add2`

Summary?

Exception handling Part 2

Exception handling Part 2 - remind



Exception handling Part 2

- There are two (C++) 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.

Exception handling Part 2

- An exception is also an object that is passed from the problem location to the location where the problem will be 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.

Exception handling Part 2

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

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.

Employee.cpp

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

```cpp
ostream& operator<<(ostream &out, const Employee &emp)
{
   out << "Employee " << emp.empNum << " : Rate $";
   out << emp.hourlyRate << " per hour ";
   return out;
}
```

```cpp
istream& operator>>(istream &in, Employee &emp)
{
   const int LOWNUM = 100;
   const int HIGHNUM = 999;
   const double LOWPAY = 19.49;
   const double HIGHPAY = 59.99;
   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;
}
```

```
int main()
{
   const int NUM_EMPLOYEES = 3;
   Employee aWorker[NUM_EMPLOYEES];

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

}
```

```
03:25:49 $ ./a.out
Employee #1 Enter employee number : 102
Enter hourly rate : 3
Bad data! Employee 102 : Rate $3 per hour
Please re-enter
Employee #1 Enter employee number : 120
Enter hourly rate : 500
Bad data! Employee 120 : Rate $500 per
hour
Please re-enter
Employee #1 Enter employee number : 120
Enter hourly rate : 25
Employee #2 Enter employee number : 200
Enter hourly rate : 20
Employee #3 Enter employee number : 300
Enter hourly rate : 30

Employees:
Employee 120 : Rate $25 per hour
Employee 200 : Rate $20 per hour
Employee 300 : Rate $30 per hour
```

Practice 1

The `Bad data!` statements are from the catch block.

- Here goes a class specifically for exceptions, 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.

Enum

Enumerations: enum

- An `enum` is a means of grouping together constants.
- Each enumeration is a new literal type.
- Classical `enum`s are now referred to as unscoped enums …

```
enum colour {red, green, blue};
```

- C++11 introduces scoped `enum`s, which look more like …

```
enum class lights {red, green, blue};
```

- … and are sometimes called enum classes.

Enumerations: enum

- The idea is the names entered in the braces take on literal values.
- The default values are 0 for the first, and 1 more than the previous for other entries.
- So 0, 1, 2 for the example …

▶ Run ☑ Line N

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

int main(){
enum colour {red, green, blue};
cout << red << endl;
cout << green << endl;
cout << blue << endl;
}
```

```
0
1
2
```

```
#include <iostream>
using namespace std;
int main(){
enum colour {red, green, blue};
cout << red << endl;
cout << green << endl;
cout << blue << endl;
}
```

Unscoped generally

- The general unscoped notation is

```
enum typeName {list-of-values}
```

- And once we have done this …

```
enum colour {red, green, blue};
```

- … we can declare and use variables of that type …

```
colour hatColour;
hatColour = blue;
```

Unscoped generally

- You don’t actually need the typename to set up the values but you then won’t have corresponding types…

```
enum {a, b, c};
```

- You can set values as well, as in this example:

```
enum {floatPrec = 6, doublePrec = 10,
     double_doublePrec = 10};
```

- What if you set some?

```
enum {a = 4, b, c, d};
```

- Or set some equal …

```
enum {a = 4, b, c, d = 4};
```

Unscoped generally

- Unscoped `enum`s are accessible anywhere, scoped are not.
- The general notation for scoped ones …

```
enum class typename {list-of-values};
```

- To access these values we need to use the scope resolution operator, so `typename::value`.
- Let's look at an example.

Unscoped generally

- Set up a couple of `enum`s, one unscoped and one scoped.

```
enum colour {red, yellow, green};
enum class peppers {red, yellow, green};
```

- Which of these will work?

```
colour eyes = green;
peppers p1 = green;
colour hair = colour::red;
peppers p2 = peppers::red;
```

Unscoped generally

- Set up a couple of `enum`s, one unscoped and one scoped.

```
enum colour {red, yellow, green};
enum class peppers {red, yellow, green};
```

- Which of these will work?

```
colour eyes = green;
peppers p1 = green;
colour hair = colour::red;
peppers p2 = peppers::red;
```

- All but the second, where you try to initialise `peppers` with a `colour`.

enum type types: C++11

- By default, scoped `enum`s have `int` as the underlying type.
- ```
  enum intValues : unsigned long long
  { charType = 255, shortType = 65535,
  intType = 65535, longType = 4294967295UL,
  long_longType = 18446744073709551615ULL};
  ```
  ```

enum type types: C++11

- Unscoped `enum`s don't have a default type, just an integral large enough to hold the specified values.
- This makes a difference when we use forward declaration for `enum`s, which is allowed from C++11 on.
  - The underlying size must be specified and since unscoped doesn't have a default we must be explicit for unscoped enums.

```
enum intValues : unsigned long long;
enum class open_modes;
```

Scoped are better than unscoped

- We can represent enums in UML as follows: