University of Essex

C++ Programming

11 – Type Conversions, Exceptions

Type Conversion

- We have seen that it is possible to convert between types by creating new objects using type conversion constructors.
- However we often wish to treat a pointer or a reference to an object of one type as a pointer or reference to an object of a subtype.
- In Java we might use a loop of the form

```
for (Person p:myList)
if (p instanceof Student)
        System.out.println(((Student)p).regNo());
```

- to print the registration numbers of all students in a list of persons; we use down-casting to treat a reference to a person as a reference to a student.

Type Conversion

- In C++ we can use a similar syntax; this is known as ***C-style casting***. If we know that all of the members of **myList** are students (where **myList** is of type **List<Person*>**) we could print their registration numbers using

```
1  for (List<Person*>::Iterator it = myList.begin();
2       it != myList.end(); it++)
3  {
4       Person *p = *it;
5       cout << (Student*)p->regNo() << endl;
6  }
```

- Note that this type of casting only works with pointers and references; also we had to use a list of pointers since a list of objects of type **Person** cannot hold objects of derived classes.

Type Conversion

- When C-style casting is used no type-checking whatsoever is performed either by the compiler or at runtime, so if any of the objects in the list in the code on the previous slide was not a student the behaviour of the program would be unpredictable and almost certainly incorrect.
- Because of the total lack of type-checking the use of C-style casting in C++ is discouraged;
- C++ instead provides type- casting operators that do perform some type-checking;
- these include **static_cast** and **dynamic_cast**.

static_cast

- The **static_cast** operator can be used to convert between types in situations where the compiler regards it as reasonable to do so.
- Permitted static casts include down-casting from pointers or references to base class objects to pointers or references to derived class objects, and also conversions between numeric types
- The syntax for a static cast is **static_cast<*T*>(*e*)**, where ***T*** is a type and ***e*** is an expression.
- We could replace the output line of the code fragment on slide 3 with

```
cout << static_cast<Student*>(p)->regNo() << endl;
```

static_cast

- When **static_cast** is used no runtime checking is performed so, as before, if the pointer **p** on the previous slide did not point to a student the behaviour of the program would be unpredictable and almost certainly incorrect.
- The compiler will however check that the type conversion is reasonable; for example, an attempt to cast a pointer to a person to a pointer to a date would be rejected.
- To obtain a real-number average from an integer total and an integer count we could use

  double average = static_cast<double>(sum)/count;
- The compiler will generate code to convert between integer and real-number representations.

dynamic_cast

- The **dynamic_cast** operator can be used to treat a pointer or reference to a base class object as a pointer or reference to a derived class object, with run-time type checking being performed.
- In the case of pointer conversion the result of the operation will be a null pointer if the cast is invalid;
- In the case of reference conversion an exception of type **bad_cast** would be thrown.
- C++ has no **instanceof** operator, but we can check if a person is a student by examining the outcome of an attempted dynamic cast.
- Two versions of C++ code to perform the same task as the Java code on slide 2 are presented on the next slide; one uses pointers and the other uses references.

dynamic_cast

```
1  // first version
2  for (List<Person*>::Iterator it = myList.begin();
3      it != myList.end(); it++)
4  {
5      Student* s = dynamic_cast<Student*>(*it);
6      if (s != 0)
7          cout << s->regNo() << endl;
8  }
9
10 // second version
11 for (List<Person*>::Iterator it = myList.begin();
12     it != myList.end(); it++)
13 {
14     try
15     { cout << dynamic_cast<Student&>(*(*it)).regNo()
16         << endl;
17     }
18     catch (bad_cast b) {}
19 }
```

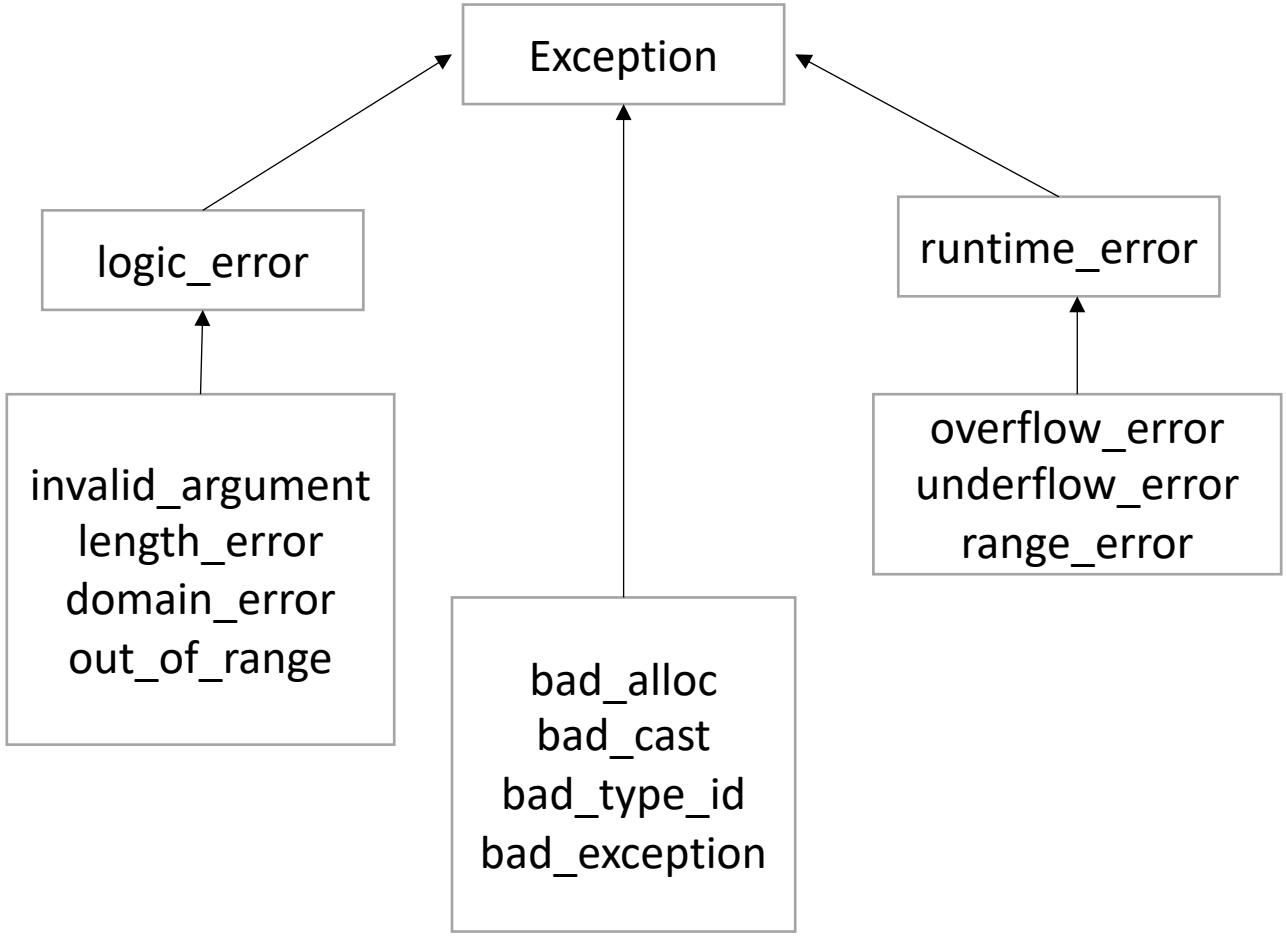
Exceptions

- It is often the case that the handling of an error needs to be done in a different part of the program than the place where it is detected.
- For example if something goes wrong when a function is called the caller may want to display an appropriate message.
- The detector of the error needs to indicate in some way that an error has occurred.
- This could be done using a global flag (e.g. **if (x<0) error = true;**) or by returning a special value (e.g. **if (x<0) return -99999;**), but the preferred approach is to throw an exception (e.g. **if (x<0) throw tantrum();**)

Exception

- C++'s approach to exception handling uses **try** and **catch** blocks in the same way as Java.
- An exception should normally be an object whose type is either **exception** (as defined in the header file **<stdexcept>**) or some subclass of **exception**, but (unlike Java) it is permissible to use a class that does not inherit from **exception**.
- The **exception** class has a public member function called **what** that will return a C-string containing a description of the exception.
- The header file **<stdexcept>** defines several exception classes that may be thrown by standard library functions; the hierarchy shown on the next slide shows most of these classes.

Exception


Exception
logic_error
runtime_error
invalid_argument
length_error
domain_error
out_of_range
bad_alloc
bad_cast
bad_type_id
bad_exception
overflow_error
underflow_error
range_error

Exception

- The two subclasses of the **exception** class, **runtime_error** and **logic_error**, have constructors that take a constant reference to a string as an argument;
- The inherited **what** function will return, as a C-string, the string which was supplied when the object was created.
- When a user wishes to create his own exception classes the usual practice is to use one of these as the base class.
- The former should be used when the error is of a nature that could not be prevented by the programmer (e.g. failure to open a file) whereas the former should be used if the programmer could have prevented it (e.g. index out of range).

Exception

- If we wish to write a named exception class that behaves in the same way as one of these classes we would simply use something like

```
1  class MyException: public runtime_error
2  {
3      public:
4          MyException(const string &s): runtime_exception(s) {}
5  }
```

- This allows users to write **catch** blocks that just catch occurrences of **MyException**, for example.

```
1  try
2  {
3      .........
4  }
5  catch (MyException e)
6  {
7      cout << "Caught exception: " << e.what();
8  }
```

Exception

- In many applications we may wish to generate message strings containing some fixed text and additional text supplied as an argument to the exception constructor.

```
1  class StudentException: public logic_error
2  {
3      public:
4          StudentException(const string& s):
5          logic_error("Student problem: " + s)
6          {
7
8          }
9  }
```

Exception

- If we need to generate a message string that cannot be created in an expression that can be used in an initialiser list we should instead store the string as a private data member and redefine the **what** function to return it.

```
1  class StudentException: public logic_error
2  {
3      private char mess[100];
4      public:
5          StudentException(const string &s):
6          {
7              // generate message in mess
8          }
9          const char *what() const
10         {
11             return mess;
12         }
13 }
```

Exception

- We can specify which classes of exceptions could potentially be thrown by a function. C++, however, has no **throws** keyword, so **throw** is used with the following syntax:

```
1  int myFunc(int n) throw (MyException, HisEx)
2  {
3      if (n==0)
4          throw MyException("n is zero");
5      if (n<0)
6          throw HisEx(n);
7      ......
8  }
```

- Observe that parentheses are always used, even if there is only one exception class in the list.
- Unlike Java there is no concept of checked and unchecked exceptions so the choice of whether to use an exception-list in a declaration is left to the programmer.

Exception

- Since a function with a throw list can call another function defined in a separate file that has no throw list the compiler is unable to check that the function throws only the listed exceptions.
- Hence the use of throw lists is now generally discouraged.

Exception

- We can use **throw()** in the declaration to indicate that the function will not throw an exception.
- Again this does not however guarantee that an exception will not be thrown since the compiler is unable to check whether any calls made inside the function body to functions written in a separate file can potentially throw exceptions.
- If a function does try to propagate an exception that was thrown by a called function but is not listed in its throw-clause an exception of type **bad_exception** will be thrown instead.

Exception

- We can use **catch(exception e)** to catch all exceptions that derive from **exception**;
- This will not however catch any exceptions that are not objects derived from this class.
- The variable **e** would just hold the inherited part of the class, so if the exception is of a class that has, for example, overwritten
- the **what** member of **exception** a call to **e.what()** would use the base class version, not the derived class version.
- A catch block headed **catch(...)** will catch anything that is thrown, but since this may be an object of any type, or even a primitive or pointer, we cannot examine it within the block. (The **...** is part of the syntax, i.e. exactly three dots without any spaces.)

University of Essex

CONTACT YOUR LECTURER

m.barros@essex.ac.uk

University of Essex