by Leedehai

Chapter 15 Nested classes, exceptions, and RTTI

15.1 Nested classes

- In C++, you can place a class declaration inside another class.
- A *nested class* is NOT composition. Composition means having a class object as a member of another class, whereas nesting a class does NOT create a class member. Instead, it defines a type of class that in known ONLY to the class in which it is defined.

```
class Queue {
  // Node is a nested class, local to Queue's class scope
  class Node {public: int data; Node *next;};
  ...
}
```

- Two kinds of access pertain to nested classes.
  - First, where a nested class is declared controls the scope of the nested class; that is, it establishes which parts of a program can create objects of that class. To be more specific:

Where declared	Available to nesting class	Available to classes inheriting the nesting class	Available to the outside world
Private section	Yes	No	No
Protected section	Yes	Yes	No
Public section	Yes	Yes	Yes, with class qualifier

  - Second, as with any other classes, the public, protected, and private sections of a nested class provide access control to class members.

15.2 Exceptions

- Programs sometimes encounter runtime errors, such as trying to read a nonexistent file, or requesting more memory than is available, or reading in a value that is not valid, or trying to divide by zero. *Exception*s are tools for programmers to prepare for such cases.
- Small tool #1: calling `abort()` or `exit()`
  - The `abort()` and `exit()` function have their prototypes in the `<cstdlib>` (or `stdlib.h` for C), and their names are a part of the `std` namespace, of course.

  - If it is called, it typically sends a message such as "abnormal program termination" to the *standard error stream* (the same as one used by `std:cerr`) and terminates the program. It also returns an implementation-dependent value that indicates failure the the operating system, or, if the program was initiated by another program, to the parent process.
  - `exit()` has similar functions as `abort()`, but `exit()` does not display a message when it is called, and flushes file buffers.

    ```
    ...
    if(a == -b) {std::cout << "untenable argument\n"; std::abort();}
    ...
    ```

- Small tool #2: returning an error code
  - A more flexible approach than aborting is to use a function's return value to indicate a problem, so that the parent program or the operating system may take further steps other than aborting right away.

    ```
    bool push(Queue *q, int &x){
      if(isFull(q)) return false; // the queue is full, cannot push
      ...
      return true;                // successful operation
    }
    ```

    ```
    short render(...) {
      if(/*some failure occures*/) return ERROR_VAL1; // a constant
      if(/*some failure occures*/) return ERROR_VAL2; // a constant
      ...
    }
    ```

- The big tool: the exception mechanism.
  - A C++ *exception* is a response to an exceptional circumstance that arises during runtime. Exceptions provide a way to transfer control from one part of a program to another.
  - Handling an exception has three components:
    - Throwing an exception. A program throws an exception when a problem shows up, with the keyword `throw`. A *throw statement*, in essence, is a jump; that is, it instructs the program to jump to statements at another location.
    - Catching an exception with a *handler*. A program catches an exception with an exception handler, with the keyword `catch`, at the place in the program where you want to handle the problem. A handler, also known as a *catch block*, begins with the keyword `catch`, followed by a type declaration in parentheses that indicates the type of exception to which it responds. And then a block of code is followed, which contains the actions to take. There can be multiple `catach` statements; the first `catch` that has the matching type catches the exception, and the rest is ignored.
    - Using a `try` block. A *try block* identifies a block of code for which particular exceptions will be activated. It is followed by one or more `catch` blocks. The `try` block itself is indicated by the keyword `try`, followed by a code block indicating the code for which exceptions will be noticed. If a program completes executing a `try` block without any exceptions being thrown, it skips the `catch` block and goes on.

- A short example

```cpp
int main() {
  double x, y, z; std::cout << "Enter two numbers: ";
  std::cin >> x >> y;
  try { z = harmonicMean(x,y); } // try block
  catch (const char *s) { // exception handler, a.k.a. catch block
    std::cout << s << std::endl;
    std::cout << "Enter a new pair of number"; return 0;
  }
  std::cout << "Heamonic mean is " << z << std::endl;
  std::cout << "Enter next set of numbers (q to quit): ";
  return 0;
}
double harmonicMean(double a, double b) {
  if (a == -b) throw "bad arguments: a = -b is invalid.\n";
  return 2.0 * a * b / (a + b);
}
```

  Here, the thrown exception is a string `"bad arguments: a = -b is invalid."`, so the exception type here is `const char *`. It can be other types, even class types.

- Executing a `throw` statement is a bit like executing a `return` statement in that it terminates function execution. However, instead of returning control to the calling program, a `throw` causes a program to back up through the sequence of current function calls in the program stack until it finds the function that contains a `try-catch` combination that catches the exception.
  - If a `try-catch` combination cannot catch the thrown exception due to type mismatch, the exception will continue to be thrown back through the sequence of call until it encounters a matched `catch`.
  - If a function throws an exception but there is no `try` block or no matching `cathch`, the program will typically call the `abort` function, but you can modify that behavior, as discussed later.
  - A `catch` block can re-throw the exception it has caught, by using the plain statement `throw;` as its last statement.
  - Going back through the sequence of current function call is termed *stack unwinding*. When the program jumps back to a function call in the stack, the data on top of it are destroyed (FILO rule), including any class objects.
- Using objects as exceptions
  - In practice, exception type is class type, instead of a simple `const char *`. One important advantage is that you can use different exception types to distinguish among different exceptions, and an object can carry information with it, which helps you to identify the problem and enables the program to choose which course of action to pursue.
  - Here is a possible design for an exception to be thrown by the `harmonicMean()` function in the example above.

```cpp
class bad_hMean{  // a possible exception class
private: double v1, v2;
public: bad_hMean(int a = 0, int b = 0) : v1(a), v2(b) {}
        void message(); void show() {...}
};
inline void bad_hMean::message() {
  std::cout << "bad arguments: a = -b is invalid.\n";
}
```

```cpp
// new version of harmonicMean(), using "bad_hMean" class
double harmonicMean(double a, double b) {
  if (a == -b) throw bad_hMean(a,b); // calls the constructor
  return 2.0 * a * b / (a + b);
}
// new version of the handler, using "bad_hMean" class
catch(bad_hMean &b) {...}
```

If you have multiple types of exceptions to prepare for, then you should design multiple exception classes, and have multiple `catch` blocks to detect and catch each exception.

```cpp
catch(bad_hMean &b) {...}
catch(bad_gMean &b) {...}
...
```

- You can design an inheritance hierarchy of exception classes. When you does so, you should arrange the order of the `catch` blocks so that the exception of the most-derived class is caught first, and the base-class exception is called last. This feature utilizes class reference/pointer's implicit upcasting (see 13.2). Or, you can use the base-class handler to take care of them altogether.

```cpp
class bad_1 {...};
class bad_2 : public bad_1 {...};
class bad_3 : public bad_2 {...};
...
try {...}
catch (bad_3 &b) {...} // most derived class of exception
catch (bad_2 &b) {...}
catch (bad_1 &b) {...} // base class of exception
```

- You can explicitly indicate whether a function might throw an exception, by postfixing `noexcept` (no-exception) at the function prototype and function definition. There also is a `noexcept()` operator that reports on whether or not its operand could throw an exception.
- A caution: because a `throw` statement acts like a `return` statement in that it terminates a function, it is possible that memory blocks that are allocated dynamically in this function is not freed yet when an exception is thrown.

```cpp
void func(std::string &str){
  std::string *ps = new string(str); // dynamic memory allocation
  ...
  if (/*something weird*/) throw exception();
  ...
  delete ps;
  return;
}
```

- You have to take care of it, either by catching the exception in the same function and free the memory in that `catch` block, or by using smart pointer templates, as discussed in Chapter 16.

15.3 More on exceptions

- A `throw` ALWAYS generates a new copy of the thrown data, be it a string, a class object, or something else. That is because the original copy is destroyed when the called function, which contains the `throw` statement, terminates.
  - However, the `catch` block that is responsible for handling the exception prefers a reference-type formal argument rather than the copy itself, because
    - A class reference has the useful feature of implicit upcasting (see 13.2) so it can take derived-class objects, and
    - you can assign a variable to a reference-type formal argument as well as to a regular formal argument.
- The `exception` class: declared as part of `std` namespace in the `<exception>`
  - C++ uses the `exception` as the base class for other standard exception classes.
  - Your code, too, can throw an `exception` object or use it as a base class.
  - One `virtual` member function of this class is named `what()`, which returns a string. Moreover, because this methods is `virtual`, you can override it in a class derived from `exception`.

```cpp
#include <exception>
class bad_hMean : public std::exception { ...
public: const char * what() { return "bad arguments,\n";} // override
};
class bad_gMean : public std::exception { ...
public: const char * what() { return "bad arguments,\n";} // override
};
```

  Of course, if you do not want to handle these derived exceptions separately, you can catch them with the base-class handler:

```cpp
try {...}
catch(std::exception &e) {std::cout << e.what() << std::endl; ...}
```

- The `<stdexcept>` header: declared more exception classes in the `std` namespace

- `logic_error` class and `runtime_error` class are both publicly derived from `exeception`, and are prepared for typical scenarios a program might encounter. Moreover, these two's constructors respectively take a string as their initializer, and return the string by the method `what()` inherited from `exception`. For example, the `logic_error` is declared as:

```
class logic_error : public exception { // the same as "domain_error"
public: explicit logic_error(const std::string &what_arg);
        explicit logic_error(const char* what_arg);
};
```

- These two classes serve as base classes for two family of derived classes.
  - The `logic_error` family describe typical logic errors that should be prepared for when programming, including `domain_error`, `invalid_argument`, `length_error`, `out_of_bounds`.
  - The `runtime_error` family describes errors that might show up during runtime but could not easily be predicted and prevented, including `range_error`, `overflow_error`, `underflow_error`.

- The `bad_alloc` exception and `new`: declared in `<new>` in the `std` namespace
  - This exception class is publicly derived from the `exception` class and is meant to be thrown by `new` if memory allocation fails. In the days of yore, failed memory request returns a null pointer, but now it throws an exception.

```
try {
  std::cout << "Trying to request a chunk of memory:\n";
  double *pm = new double[10000]; std::cout << "Got it!\n";
} // free the memory after using if the allocation is succeed
catch (std::bad_alloc &ba) {
  std::cout << "Failed operation." << ba.what() << std::endl;
}
```

- The null pointer and `new`: the old version `new` returns a null pointer instead of throwing an exception. C++11 let you explicitly specify which version you are using:

```
int *pn1 = new (std::nothrow) int[10000]; // old version
```

- When exceptions go astray: what if there are uncaught exceptions?
  - If an exception goes astray, the program calls a function named `terminate()`, which by default, called the `abort()` function.
  - You can modify the behavior of `terminate()` by *registering* a function that `terminate()` should call instead of `abort()`. To do this, you call the `set_terminate()` function.
  - Both `set_terminate()` and `terminate()` are declared in `<exception>`:

```
typedef void (*terminate_handler)(); // typedefine a function pointer
terminate_handler set_terminate(terminate_handler f) throw(); //C++98
terminate_handler set_terminate(terminate_handler f) noexcept;//C++11
void terminate(); /* C++98 */  void terminate() noexcept; /* C++11 */
```

Here the typedef makes terminate_handler the type name for a pointer to a function that has no arguments and no return value.The `set_terminate()` function takes, as its argument, the name of a function (that is, its address) that has no arguments and the `void` return type. It returns the address of the previously registered function. Calling the `set_terminate()` function with your designated function's address (the name), makes `terminate()`, when triggered by the program, calls your function instead of `abort()`.

```
void myQuit() {...} // your designated function
set_terminate(myQuit);
```

- Exception cautions
  - The exception handling mechanism should be designed and integrated into a program rather than tacked on. Doing this slows your speed of developing and complicates the code, but it enhances the opportunities for preventing errors.
  - In modern libraries, exception handling can appear to reach new levels of complexity - much of it due to undocumented or poorly documented exception-handling routines. Programmers usually struggle to understand what exceptions are thrown, why and when they occur, and how to handle them.
  - Therefore, the importance of robust testing and good documentation is never exaggerated. It is a good habit in developing libraries and other software.

15.4 Runtime Type Identification (RTTI)

- The intent of *runtime type identification* (RTTI) is to provide a standard way for a program to determine the type of object during runtime. Many class libraries have already provided ways to do so for their own class objects, but in the absence of built-in support in C++.
- RTTI is used to detect the type of object which is bound to by a base-class reference/pointer. This could be useful if you want to call the object's some method that is not a `virtual` function possessed by all members of the class hierarchy, or if you want to keep track of which kinds of object are bound for debugging purposes.
- You can use RTTI ONLY with a class hierarchy that has `virtual` functions.

  > The reason for this is that these are the only class hierarchies for which you should be assigning the addresses for derived objects to base-class pointers.

- C++ has three components supporting RTTI:
  - The `dynamic_cast` operator generates a pointer to a derived class from a base-class pointer, if possible. Otherwise, the operator returns the null pointer.
  - The `typeid` operator returns a value identifying the exact type of an object.
  - A `type_info` structure holds information about a particular type.
- The `dynamic_cast` operator: the most heavily used RTTI component.
  - It does NOT answer the question of what type of object a pointer points to. Instead, it answers the question of whether you can safely assign the address of an object to a pointer of a particular type.
  - `dynamic_cast<Type *>(pt)` converts the pointer `pt` to a pointer of type `Type *` if the pointed-to object (`*pt`) is of type `Type` or else derived directly or indirectly from type `Type`. Otherwise, the expression evaluates to `0`, the null pointer.

- The `typeid` operatpr and `type_info` class
    - The `typeid` operator lets you determine whether two objects are the same type. It accepts two kind of operands (like `sizeof`): the name of a class, OR an expression that evaluates to an object.
    - The `typeid` operator returns a reference to a `type_info` object, where `type_info` is a class defined in the `<typeinfo>` header. The `type_info` class overloads the `==` and `!=` operators so that you can use these operators to compare types.
        - For example, this expression evaluates to a `bool` value: `true` if `p` points to a `Dog` object and `flase` otherwise: `typeid(Dog) == typeid(*p)`. If `p` happens to be a null pointer, the program throws a `bad_typeid` exception, which is derived from the `exception` class and is declared in `<typeinfo>`.
        - Moreover, a `type_info` object has a `name()` method that returns an implementation-dependent string that is typically the name of the class: `std::cout << "Now the type is " << typeid(*p).name() << std::endl;`.

- [ ]