Object-Oriented Programming: Advanced Topics

Lectured by Ming-Te Chi 紀明德

First Semester, 2022

Computer Science Department
National Chengchi University

Slides credited from 李蔡彥 and 廖峻鋒

Advanced Topics

- Error handling
  - Assertion
  - Exceptions

- Run-time type identification (RTTI)
  - Dynamic_cast
  - typeid

Assertion (c macro)

- An assertion is a statement that must be true for the function to be correct.
- The program exits if the assertion fails.

```
#include <cassert>
double Divide(double numerator, double denominator){
    assert(denominator!=0);
    return numerator/denominator;
}
int main() {
    cout << Divide(4, 0);
}
```

```
a.out: main.cpp:15: double Divide(double, double):
Assertion `denominator!=0' failed.
```

- Assertions are good debugging tools and can be turned off at compile time.

```
#define NDEBUG
#include <cassert>
```

static_assert (c++11)

[code]

- Because assert is a function-like macro, commas anywhere in condition that are not protected by parentheses are interpreted as macro argument separators. Such commas are often found in template argument lists and list-initialization:

```
//std::is_same_v c++17
assert(std::is_same_v<int, int>); // error: assert does not take two arguments
assert((std::is_same_v<int, int>)); // OK: one argument
static_assert(std::is_same_v<int, int>); // OK: not a macro

std::complex<double> c;
assert(c == std::complex<double>{0, 0}); // error
assert((c == std::complex<double>{0, 0})); // OK
```

- static_assert(a < 10, "a is too small");

Exceptions

- An assertion terminates the program but you may not want to give up at all or at least not right away.
- A function indicates that an error has occurred by *throwing* an exception and control passes back to the calling function which *catch*es the error.

```
void StackT::push(int element) {
    if (isFull())
        throw element;
    top++;
    array[top]=element;
}
int main() {
    StackT stack(2); // a stack of size 2
    try {
        stack.push(1);
        stack.push(2);
        stack.push(3);
    } catch (int element) {
        cout << "Stack is full for " << element << ".\n";
    }
}
```

Output:
Stack is full for 3.

Throwing an Exception

- No catch -> termination.
- Throwing an exception returns control to the nearest catch block.

```
void StackT::push(int element) {
    if (isFull())
        throw element;
    top++;
    array[top]=element;
}
void foo() {
    StackT stack(2); // a stack of size 2
    stack.push(1);
    stack.push(2);
    stack.push(3);
}
int main() {
    try {
        foo();
    } catch (int element) {
        cout << "Stack is full for " << element << ".\n";
    }
}
```

Throwing Multiple Exceptions

```
void StackT::push(int element) {
    if (isFull())
        throw element;
    array[top]=element;
    if (isEmpty())
        throw "The stack is still empty.\n";
}
int main() {
    StackT stack(1);
    try {
        stack.push(1);
        stack.push(2);  // will be ignored
    } catch (int element) {
        cout << "Stack is full for " << element << ".\n";
    } catch (char *error) {
        cout << error;
    } catch (...) {
        cout << "Unknown error\n";
    }
}
```

generic handler comes last

Re-throwing the Exception

```
void StackT::push(int element) {
    if (isFull())
        throw element;
    array[++top]=element;
}
void Foo() {
    try { ...
    } catch(int element) {
        if (size < 10000) {
            expandStack();
        } else
            throw;
    }
}
int main() {
    StackT stack(1);
    try {
        Foo();
    } catch (int element) {
        cout << "Stack is full for " << element << ".\n";
    }
}
```

Exception Specification (until C++17)

- Specifying the list after function defintion to restrict the exception types of a function at run time.

```
void temperature(int t) throw(char *, int){
    if (t==100) throw "Boiling";
    else if (t==0) throw “Freezing”;
    else throw t;
}

try {
    try {
        temperature(13);
    } catch (int x) {
        cout << "Temperature = " << x << "deg C" << endl;
    }
    temperature(0);
} catch (char *str) {
    cout << str << endl;
}
```

Unexpected Exception (until C++11)

- If an exception type is not expected in a function, an unexpected() function is called.
  (terminate by default)
- A user can override the default behavior by specifying a handling function.

```
void myUnexpected() {
    …
}
int main() {
    set_unexpected(myUnexpected);
    …
}
```

A Sample Base Exception Class

- Derive user-customized exception from this class.
- Do not throw exception by itself.
- Override `what()` virtual function.

```
//c++11
class exception {
public:
  exception () noexcept;
  exception (const exception&) noexcept;
  exception& operator= (const exception&) noexcept;
  virtual ~exception();
  virtual const char* what() const noexcept;
};
```

`noexcept` (since C++11)

- Specifies whether a function could throw exceptions.
  - the function is declared not to throw any exceptions
- Whenever an exception is thrown and the search for a handler encounters the outermost block of a non-throwing function, the function std::terminate is called:

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)

runtime_error
- range_error
- overflow_error
- underflow_error
- regex_error(C++11)
- system_error(C++11)
  - ios_base::failure(C++11)
  - ios_base::failure(until C++11)

bad_typeid

bad_cast

bad_alloc
- bad_array_new_length(C++11)

bad_exception

bad_weak_ptr(C++11)

bad_function_call(C++11)

Example

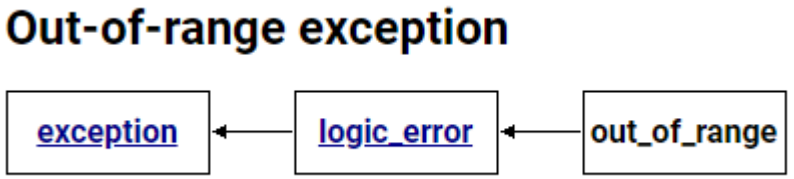
```
// exception example
#include <iostream>       // std::cerr
#include <typeinfo>       // operator typeid
#include <exception>      // std::exception

class Polymorphic {virtual void member(){}};

int main () {
  try
  {
    Polymorphic * pb = 0;
    typeid(*pb);  // throws a bad_typeid exception
  }
  catch (std::exception& e)
  {
    std::cerr << "exception caught: " << e.what() << '\n';
  }
  return 0;
}
```

```
exception caught: St10bad_typeid
```

Example




```
// out_of_range example
#include <iostream> // std::cerr
#include <stdexcept> // std::out_of_range
#include <vector> // std::vector

int main (void) {
        std::vector<int> myvector(10);
      try {
         myvector.at(20)=100; // vector::at throws an out-of-range
       }
      catch (const std::out_of_range& oor) {
         std::cerr << "Out of Range error: " << oor.what() << '\n';
      }
      return 0;
}
```

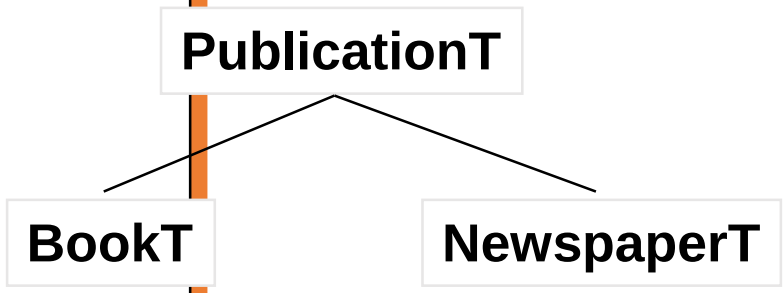
Run-time type identification (RTTI)

Using Exceptions Carefully

- Exception should be exceptional.
  - Getting a `EOF` when reading data from a file is an exception only if the `EOF` comes unexpectedly.
- Avoid unnecessary exceptions.
  - The syntax is cumbersome for the client.
  - Exception violate normal flow of control (like `goto`).
  - Exceptions require care with respect to dynamically allocated memory.
- From the client's point of view, the desirable solution is to prevent exceptions, not to catch them.
- Do not use exceptions to cover flaws in the class design.

Run-Time Type Identification

```
class PublicationT {
  public:
    PublicationT(char *title);
    virtual void purchasingInfo() const;
    bool same(char *inTitle) const;
  private:
    char *title;
};
class BookT: public PublicationT {
  public:
    BookT(char *inTitle, char *inAuthor);
    void printAuthor() const;
    void purchasingInfo() const;
  private:
    char *auther;
};
class NewspaperT:public PublicationT {
  public:
    NewspaperT(char *title, long inCirculation);
    void printCirculation() const;
    void purchasingInfo() const;
 private:
    long circulation;
}
```



Run-Time Type Identification

```
class InventoryT { // keep a list of publications
  public:
    ..
    void insert(PublicationT *element, int slot);
    void query(char *title);
    PublicationT *next();
  private:
    PublicationT **data; //linklist
    ...
};
void InventoryT::query(char *title) {
    PublicationT *publicationPointer;
    while((publicationPointer = next())!=NULL)
       if (publicationPointer->same(title))
          publicationPointer->purchasingInfo();
}
int main() {
    InventoryT database(2);
    BookT *novel=new BookT("Romeo and Juliet", "Will");
    NewspaperT *journal=new NewspaperT("US Today",10);
    database.insert(novel, 0);
    database.insert(journal, 1);
    database.query("Romeo and Juliet");
}
```

Only purchasing information got printed.

dynamic_cast < type-id > (expression)

- We would like to identify the type of the object in order to print more specific information on the query.

```
void InventoryT::query(char *title) {
    PublicationT *publicationPtr;
    BookT *bookPtr;
    NewspaperT *newspaperPtr;
    while((publicationPtr = next())!=NULL)
        if (publicationPtr->same(title) {
            bookPtr = dynamic_cast<BookT *>(publicationPtr);
            newspaperPtr = dynamic_cast<NewspaperT *>(publicationPtr);
            if (bookPtr)
                bookPtr->printAuthor();
            if (newspaperPtr)
                newspaperPtr->printCirculation();

            publicationPointer->purchasingInfo();
        }
}
```

`dynamic_cast` will make the conversion *if it is valid*; otherwise, `NULL` will be returned.

Alternative Solution

- When do you need to use RTTI?
  - When you are deriving an object from a compiled library.
  - If you have access to the base class, redesign it with polymorphism.

```
class PublicationT {
  public:
    ...
    virtual void display() const;
    ...
};
void PublicationT::display() const {
}
void BookT::display() const {
    printAuthor();
    purchasingInfo();
}
void NewspaperT::display() const {
    printCirculation();
    purchasingInfo();
}
// in Inventory::query, simply call
// publicationPtr->display()
```

typeid in C++

- An “operator” used to retrieve the runtime or dynamic type information of an object
- The type information is stored and returned with a typeinfo object.
- Syntax:
  - `typeid(expression)`
  - `typeid(type)`
- Expression: first evaluated and then determined
- Type: variable or the object
- Return: `const type_info&`

Example of typeid

```
#include <typeinfo>
int main()
{
    int i, j;
    float f;
    char c, *d;
    double e;
    const type_info& ti1 = typeid(i);
    const type_info& ti2 = typeid(j);
    const type_info& ti3 = typeid(f*c); // expression
    const type_info& ti4 = typeid(c);
    const type_info& ti5 = typeid(d);
    const type_info& ti6 = typeid(*d);
    // Printing the types of the variables of different
      data type on the console
    cout << ti1.name() << endl;
    cout << ti2.name() << endl; // ti1 == t12
    cout << ti3.name() << endl;
    cout << ti4.name() << endl;
    cout << ti5.name() << endl;
    cout << ti6.name() << endl; // ti5 != t16
```

```
i
i
f
c
Pc
c
```

Another Example of typeid

```
#include <typeinfo>
class B1 {// polymorphic base class
public:
    virtual void fun() {}
};
class B2 {}; // non-polymorphic base class
class D1 : public B1 {};
class D2 : public B2 {};
int main()
{
    D1* d1 = new D1;
    B1* b1 = d1;
    D2* d2 = new D2;
    B2* b2 = d2;
    cout << typeid( d1 ).name() << endl;
    cout << typeid( b1 ).name() << endl;
    cout << typeid( *d1 ).name() << endl;
    cout << typeid( *b1 ).name() << endl;
    cout << typeid( d2 ).name() << endl;
    cout << typeid( b2 ).name() << endl;
    cout << typeid( *d2 ).name() << endl;
    cout << typeid( *b2 ).name() << endl;
```

```
P2D1
P2B1
2D1
2D1
P2D2
P2B2
2D2
2B2
```