# Effective C++

Unit 5 – 10

Third Edition

Knowledge Discovery & Database Research Lab

# ITEM 5 : Know what functions C++ silently writes and calls

## Copy constructor, copy assignment operator, destructor

▲ If you don't declare no constructors,compilers will declare a default constructor.

```
class Empty{};

class Empty{
public:
  Empty(){. . . }                                  // default constructor
  Empty(const Empty&rhs) {. . . }                  // copy constructor
  ~Empty(){. . . }                                 // destructor
  Empty&operator = (const Empty&rhs) {. . . }      // copy assignment operator
};
```

▲ The following code will cause each function to be generated

```
Empty e1;          // default constructor
                   // destructor
Empty e2(e1);      // copy constructor
e2 = e1;           // copy assignment operator
```

# ITEM 5 : Know what functions C++ silently writes and calls

- What do the functions do?
  - give compilers a place to put “behind the scenes” code
- Generated destructor is non-virtual (see Item 7)
- Copy constructor, copy assignment operator
  - copy each non-static data member of the source object over to the target object
- Example
  - Constructor
    : declared
  - Copy constructor
    : not declared
  - Copy assignment
    : not declared

```
template <typename T>
class NamedObject {
public:
  NamedObject(const char*name, const T&value);
  NamedObject(const std::string&name, const T&value);
  . . .
private:
  std::string nameValue;
  T objectValue;
};
```

```
NamedObject<int>no1( “Smallest Prime Number” ,2);
NamedObject<int>no2(no1);          // calls copy constructor
```

# ITEM 5 : Know what functions C++ silently writes and calls

▲ Code

```
template <class T>
class NamedObject {
public:
  NamedObject(std::string& name, const T& value);
  . . .
private:
  std::string& nameValue;
  const T objectValue;
};
```

▲ What should happen here ?

```
std::string newDog( "Persephone" );
std::string oldDog( "Satch" );
NamedObject<int>p(newDog,2);
NamedObject<int>s(oldDog,36);
p=s;
```

➡ **refuse to compile the code**

Things to remember

- Compilers may implicitly generate a class' s default constructor, copy constructor, copy assignment operator, and destructor.

# ITEM 6 : Explicitly disallow the use of compiler-generated functions you do not want

## You'd like attempt to copy HomeForSale objects to not compile

```
class HomeForSale{ . . . };

HomeForSale h1;
HomeForSale h2;
HomeForSale h3(h1);
h1 = h2;
```

**attempt to copy h1, h2 – should not compile!**

**the goal here is to prevent copying!**

### ▲ Solution

- Declare the copy constructor and the copy assignment operator private
- Do not define

```
class HomeForSale{
public:
  . . .
private:
 . . .
  HomeForSale(const HomeForSale&);          // declarations only
  HomeForSale& operator=(const HomeForSale&);
};
```

# ITEM 6 : Explicitly disallow the use of compiler-generated functions you do not want

▲ Move the link-time error up to compile time

● By declaring the copy constructor and copy assignment operator private not in HomeForSale itself

```
class Uncopyable{
protected:
  Uncopyable(){}                    // allow construction and destruction of derived objects
  ~Uncopyable(){}
private:
  Uncopyable(const Uncopyable&);                    // but, prevent copying
  Uncopyable&operator=(const Uncopyable&);
};

class HomeForSale: private Uncopyable{
 . . .
};
```

Things to remember

■ To disallow functionality automatically provided by compilers, declare the corresponding member functions private and give no implementations. Using a base class like Uncopyable is one way to do this

# ITEM 7 : Declare destructors virtual in polymorphic base classes

## Non-virtual vs virtual destructor (1)

- Avoid leaking memory and other resources

```
class TimeKeeper{
public:
  TimeKeeper();
  ~TimeKeeper();
  . . .
};
```

```
class TimeKeeper{
public:
  TimeKeeper();
  ~virtual TimeKeeper();
  . . .
};
```

```
TimeKeepr *ptk = getTimeKeeper();
. . .
delete ptk;
```

C++ specifies that when a derived class object is deleted through a pointer to a base class with non-virtual destructor, results are undefined

# ITEM 7 : Declare destructors virtual in polymorphic base classes

## Non-virtual vs virtual destructor (2)

▲ When a class is not intended to be a base class, making the destructor virtual is usually a bad idea

```
class Point{
public:
  point (int xCoord, int Ycoord);
  ~point();      // ~virtual point();
private:
  int x,y;
};
```

→ **Non-virtual : 64bit**

**virtual : 96bit or 128bit**

```
class SpecialString: public std::string{
. . .
}
SpecialString *pss = new SpecialString( "Impending Doom" );
std::string *ps;
ps = pss;
delete ps;
```

→ **SpecialString destructor won't be called**

# ITEM 7 : Declare destructors virtual in polymorphic base classes

## Pure virtual destructor

```
class AWOV{
public:
  virtual ~AWOV() = 0;              // declare
}
AWOV::~AWOV() {}                    // definition of pure virtual destructor
```

- Pure virtual functions result in abstract classes.
- The rule for giving base classes virtual destructors applied only to polymorphic base classes

Things to remember

- Polymorphic base classes should declare virtual destructors.
  If a class has any virtual functions, it should have a virtual destructor.
- Classes not designed to be base classes or not designed to be used polymorphically should not declare virtual destructors.

# ITEM 8 : Prevent exceptions from leaving destructors

## Consider:

- Suppose v has ten Widgets in it.
- During destruction of the first one: exception
- The second one : exception
- Program : undefined behavior

▲ Cause

- Destructors emitting exceptions

```
class Widget{
public:
 . . .
  ~Widget() { . . . }  // might emit an exception
};

void doSomething()
{
  std::vector<Widget> v;
  . . .
} // v is automatically destroyed here
```

# ITEM 8 : Prevent exceptions from leaving destructors

## What should you do ?

▲ Create a resource-managing classes

```
class DBConnection {
public:
 . . .
  static DBConnection create();
  void close();
}
```

→ close connection; throw an exception if closing fails

● Destructor

```
class DBConn {
public:
 . . .
  ~DBConn()
  {
   db.close();
  }
private:
  DBConnection db;
};
```

→ make sure database connections are always closed

# ITEM 8 : Prevent exceptions from leaving destructors

## Propagation of exception

- ▲ Destructor throwing exceptions means trouble
- ▲ Two primary ways to avoid the trouble

(1) Terminate the program

```
DBConn::~DBConn()
{
  try {db.close();}
  catch(..){
    make log entry that the call to close
     failed;
    std::abort();
  }
}
```

(2) Swallow the exception

```
DBConn::~DBConn()
{
  try {db.close();}
  catch(..){
    make log entry that the call to close
     failed;
  }
}
```

- In general, swallowing exceptions is a bad idea.
- The program must be able to reliably continue execution even after an error has been encountered and ignored.

# ITEM 8 : Prevent exceptions from leaving destructors

(3) Better strategy

```
DBConn::~DBConn() {
public:
 . . .
  void close() {          // new function for
    db.close();           //  client use
    close = true;
  }
  ~DBConn() {
    if (!closed) {        // close the connection
      try {               // if the client didn't
        db.close();
      }
      catch( . . .) {     // if closing fails,
       make log entry that the call to close failed;
      }
    }
  }
```

```
private:
  DBConnection db;
  bool closed;
};
```

**The exception has to come from some non-destructor function**

# ITEM 8 : Prevent exceptions from leaving destructors

Things to remember

- Destructors should never emit exceptions. If functions called in a destructor may throw, the destructor should catch any exceptions, then swallow them or terminate the program.
- If class clients need to be able to react to exceptions thrown during an operation, the class should provide a regular (i.e., non-destructor) function that performs the operation.

# ITEM 9 : Never call virtual functions during construction or destruction

## Example : a class of hierarchy for modeling stock transactions

▲ important point : auditable

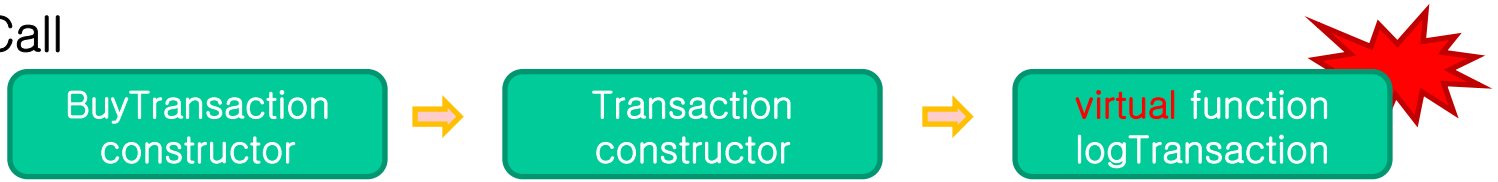
```
class Transaction{                              // base class for all transactions
public:
  Transaction();
  virtual void logTransaction() const = 0;      // make type-dependent log entry
  …
}
Transation::Transaction()                       // implementation of bass class
{
  …
  logTransaction();                             // as final action, log this transaction
}
class BuyTransaction:public Transaction{        // derived class
public:
  virtual void logTransaction() const;          // how to log transactions of this type
  …
}
```

# ITEM 9 : Never call virtual functions during construction or destruction

```
class SellTransaction:public Transaction{          // derived class
public:
  virtual void logTransaction() const;             // how to log transactions of this type
  …
}

BuyTransaction b;
```

▲ Call




- Virtual functions never go down into derived classes
- Instead, the object behaves as if it were of the base type.
- Base class parts of derived class objects are constructed before derived class parts are
- Derived class data members have not been initialized when base class constructors run
- The logTransaction function is pure virtual in Transaction.
  ☞ Unless it had been defined, the program wouldn't link.

# ITEM 9 : Never call virtual functions during construction or destruction

▲ Example 2

```
class Transaction{                          // base class for all transactions
public:
  Transaction();
  { init();}                                // call to non-virtual
  virtual void logTransaction() const = 0;
  …
private:
  void init()
  {
  …
  logTransaction();                         // that calls a virtual!
  }
};
```

- more insidious code: compile and link without complaint
- Most runtime systems will abort the program when the pure virtual is called.

# ITEM 9 : Never call virtual functions during construction or destruction

## Approach to this problem

▲ Turn logTransaction into a non-virtual function in Transaction

```
class Transaction{
public:
  explicit Transaction(const std::string& logInfo);
  void logTransaction(const std::string& logInfo) const;          // now a non-virtual func
  …
};
Transaction::Transaction(const std::string& logInfo)
{
  …
  logTransaction(logInfo);
}
```

**Now, a non-virtual call**

# ITEM 9 : Never call virtual functions during construction or destruction

```
class BuyTransaction: public Transaction{
public:
  BuyTransaction { parameters }
    : Transaction(createLogString (parameters))        // pass log info to base class constructor
  {. . . }
  . . .
private:
  static std::string createLogString (parpameters) ;
};
```

- compensate by having derived classes pass necessary construction information up to base class constructors

**Things to remember**

- Don't call virtual functions during construction or destruction, because such calls will never go to a more derived class than that of the currently executing constructor or destructor.

# ITEM 10 : Have assignment operators return a reference to *this

## Chain of assignment

### ▲ Right-associative assignment

```
int x,y,z;
x=y=z=15;

x=(y=(z=15));
```

### ▲ Example

```
class Widget{
public:
…
  Widget& operator = (const Widget& rhs)   // return type is a reference to the current class
  {
  …
  return *this;                            // return the left-hand object
  }
  …
};
```

# ITEM 10 : Have assignment operators return a reference to *this

▲ This convention applies to all assignment operators (+=, -=, *=, etc)

```
class Widget{
public:
…
  Widget& operator += (const Widget& rhs)
  {
  …
  return *this;
  }
  Widget& operator = (int rhs)    // it applies even if the operator's parameter type is
  {                               // unconventional
   …
   return *this;
  }
  …
};
```

Things to remember

- Have assignment operators return a reference to *this