

## Exception safety

## Lecture 16

### Exceptions and templates: parts 2

In the last lecture on C++, we will look more deeply at the use of exceptions and templates.

References:

- Textbook chapter 9: pp 449–451, 496–510.
- Textbook chapter 10: pp 552–568.

PMOOP(0396A)

PMOOP(0396A)-16.1

### Example

We have already seen one example where memory leak occurs when an exception is thrown. We will see another which cause **object corruption**.

```
TIntStack::Push(int data) {
    if (_cowObj._numShare > 1) { // Copy!
        --_cowObj._numShare;
        int *newObj = new TCowObj; // What if this throws bad_alloc??
        newObj._numShare = 1;
        newObj._stackData = new int [++_capacity]; // What if this throws?
        for (int i = 0; i < _count; ++i)
            newObj._stackData[i] = _cowObj._stackData[i];
        _cowObj = newObj;
        _cowObj._stackData[_count++] = data;
    } else ... // Similar to original version
}
```

Note that if the first new throws, the *numShare* element becomes incorrect because the object is still using the old COW object.

What if the second new throws?

PMOOP(0396A)-16.2

PMOOP(0396A)-16.3

### In our case...

Our first exception safe version...

```
TIntStack::Push(int data) {
    if (_cowObj._numShare > 1) { // Copy!
        int *newObj = new TCowObj; // If this throw, nothing happens
        newObj._numShare = 1;
        try { // if throw, delete newObj
            newObj._stackData = new int [_capacity+1];
        } catch (...) {
            delete newObj;
            throw; // rethrow: we can't do the push.
        }
        ++_capacity;
        --_cowObj._numShare; /* Do this only when no more exception*/
        for (int i = 0; i < _count; ++i)
            newObj._stackData[i] = _cowObj._stackData[i];
        _cowObj = newObj;
        _cowObj._stackData[_count++] = data;
    } else ... // Similar to original version
}
```

PMOOP(0396A)-16.4

PMOOP(0396A)-16.5

### Exception safety: strategies

Sometimes we **don't need exception safety**. E.g., in a local library, the **memory leak might be too small** for the programming effort.

In case you really need or want exception safety:

- At many times, you can **reorder** the code right so that when an exception occurs, nothing happens to the object.
- If you want to create an object that is always deleted at the end of a function call, you may **use auto allocation** instead. E.g., use a auto vector instead of an array as we see last time.
- If you want to **allocate** resources to be **deallocated at the end of a function**, you can make the resources an object and use auto allocation. (The “resources acquisition is initialization” technique.)
- You can use a **catch** clause to deal with it.

### Comment about the code

- For the first allocation, exception safety comes **automatically**: if the allocation throws, nothing bad happens.
- For the second allocation, exception safety must be done **manually**: if allocation failed, the allocation above it has already been done, so it must be manually removed.
- Is it possible to **avoid having to manually catch** the exception and free the allocated memory?
- In this case, there is a solution: **make the allocation within the constructor of TCowObj**.
- So everything needed by an object is allocated **when the object is constructed**, and now we have no problem of exception causing memory leak.

### A better exception safe version

```

struct TCowObj {
    ...
    TCowObj(int size): _stackData(new int[size]) {} // allocation in init
};
TIntStack::Push(int data) {
    if (_cowObj._numShare > 1) { // Copy!
        int *newObj = new TCowObj(_capacity+1); // Everything that can throw
        ++_capacity;
        newObj._numShare = 1;
        --_cowObj._numShare;
        for (int i = 0; i < _count; ++i)
            newObj._stackData[i] = _cowObj._stackData[i];
        _cowObj = newObj;
        _cowObj._stackData[_count++] = data;
    } else ... // Similar to original version
}

```

In general, if there is **only one resource to allocate**, and there is **no way to have exception** after that, then the code is exception-safe.

PMOOP(0396A)-16.6

PMOOP(0396A)-16.7

### Another powerful technique

- Notice that all the complications of dynamic variables come **only when we have a pointer somewhere**.
- If we **use an auto variable** instead, we have no such problem.
- We have seen one example: if we **avoid a dynamic array** in a function and **use a vector** instead, we avoid having to handle exception.
- Similarly, if **we do other resource allocations via an auto variable**, we avoid having to write exception catching code for them.
- This is called **resources acquisition is initialization** in C++ Jargon.
- This works for an array. But how about a **simple pointer**?

### An example problem, and its hand-crafted solution

```

// Version with ugly fix
int f() {
    TPerson* p = new TPerson(...);
    try {
        // use the pointer p
        // but may throw exception
        delete p;
    } catch (...) {
        // Must deallocate it
        delete p;
    }
}

class TPersonPtr {
public:
    TPersonPtr(TPerson* p):
        _p(p) {}
    ~TPersonPtr() { delete _p; }
    TPerson* get() { return _p; }
private:
    TPerson* _p;
};

int f() {
    TPersonPtr p(new TPerson(...));
    // use the pointer via p.get()
    // if exception thrown
    // PersonPtr is destroyed, so
    // the TPerson object is cleared
    // no need to delete!
}

```

So the *TPersonPtr* object helps by destroying the pointer **whenever the function *f* leaves its scope**.

PMOOP(0396A)-16.8

PMOOP(0396A)-16.9

### Auto-pointers

But this is very general. We can **create a template class** that does these things, so that we don't need to write it everytime.

And, in fact we don't need to write the template class either, because **the C++ standard library does that for you already!**

The class is called *auto\_ptr<Type>* (not *auto\_ptr<Type\*>*!) E.g.,

```

#include <memory>
int f() {
    auto_ptr<TPerson*> p(new TPerson(...));
    // use p.get() as we did before
    // don't delete p.get(). It is owned by p
}

```

Auto-pointers also support **assignment** and **copy-construction**: if an auto-pointer is assigned to another, it becomes a NULL pointer.

Its memory is adopted by the new auto-pointer.

### More interesting thing about auto-pointers

- Auto-pointers also allow you to **directly access the methods of the object**, without first getting the pointer by *get()*.
- E.g., if you have a *auto\_ptr<TPerson\*> p*, you can do *p->GetName()* instead of *p.get()->GetName()*.
- This is allowed because **auto\_ptr overloads the  $\rightarrow$  operator**. E.g., if we want our *TPersonPtr* to have similar behaviour, we can add:

```

TPerson* operator->(TPersonPtr& pp) {
    return pp.get();
}

```

- The C++ rule says that if the  $\rightarrow$  operator for a class is overloaded like this, *pp->XXX* will become *pp.operator->()->XXX*. Really strange syntax. I said that C++ is not designed so that code looks good to the implementer.

PMOOP(0396A)-16.10

### One more example

Just to make sure that everybody knows what we talk about: And, make sure that everybody knows the trap of *auto\_ptr*...

```

#include <memory>
#include <string>
using namespace std;
int g(auto_ptr<string*& ar, auto_ptr<string> av) { // Adopt av!
    cout << *ar << endl;
    cout << *av << endl;
}

int main() {
    auto_ptr<string> p1(new string("hello"));
    auto_ptr<string> p2(new string("world"));
    *p1 += " world"; // Overloaded *. autoptr's works like pointers
    p2->append("-wide"); // Or like this. It works really like pointers
    g(p1, p2); // p2 now becomes NULL!
    p2 = p1; // p1 is NULL now, p2 is not
    cout << p1.get() << "---" << p2.get() << endl;
}

```

PMOOP(0396A)-16.11

### Another use of auto-pointers

Usually, we need to **store a pointer** as a data member within a class, and the pointer needs to be deleted when we delete the object.

In such case, we want a pointer because we want polymorphic behaviour (rather than different lifetime of the two objects). Then it is possible to use auto-pointers:

```

class TCapability {
public:
    TCapability(istream &is) { // Our capability class in assignment 3
        ...
        _action = auto_ptr<TAction>(new TAction(is));
    }
private:
    // Automatically destroyed when TCapability is destroyed
    // so no need to write ~TCapability, but be careful on copy and assignment
    auto_ptr<TAction> _action;
};

```

PMOOP(0396A)-16.12

### How costly is it?

If auto-pointers are so good, how costly is it?

- Surprisingly, it nearly cost nothing.
- All functions are simple inline functions, so they go directly into the compiled function. **No out-of-line compiled code** is generated.
- The only data member stored by the auto-pointer is the pointer itself, so there is **no waste of memory**.
- On the other hand, it doesn't do a real lot of things. In particular, it **doesn't do reference counting**.
- It also might be easy to get trapped into the problem we have shown, **accidentally let another auto-pointer to adopt the memory**.

PMOOP(0396A)-16.13

### Exceptions in constructors

- In constructors, we can't return error code. To communicate error, it either **throw exception** or mark the object as failed using failure flag.
- What will happen to the object? If the constructor throws an exception, **the object is discarded**: allocated members are destroyed by calling their destructors and freeing their memory.
- What happen if an **initializer throws an exception**? E.g.,

```

class X {
public:
    X(): x(), y(), z() {} // What happen if y() throws?
private:
    MyClass x, y, z; // can throw on construction
};

```

- Answer: the members **before** it will be destroyed, and members **after** it won't. So things that are not constructed will not be destroyed.

PMOOP(0396A)-16.14

### Exceptions in destructors

What about destructor?

- It is a really bad idea to throw an exception in a destructor, because it can happen at a **really inconvenient time**.
- In particular, destructors can be executed **due to an exception**, when the auto variables are being destroyed.
- At that time, **the exception is not yet handled**. Throwing an exception at that time will results in a **"double fault"**.
- The **result** of a double fault is that the *terminate()* function being executed, i.e., the **program is terminated**.
- So destructors should **not** throw an exception. Luckily, most of the time it need not.  
If this advice has to be violated, one should call *uncaught\_exception()* to see whether exception handling is in progress, and only throw only when it returns false.

PMOOP(0396A)-16.15

### A few more interesting things about templates

- The TList **template class** uses templates to implement a typed list.
- This has the benefit of **complete compiler type checking**: you can't do silly things, and you don't need to do type-casting, etc.
- We have also seen a TList class that **uses void pointers** to implement the "anything". It has the benefit that **multiple instances need not be generated**.
- Is it possible to **combine their benefits**, so that we can have full type-checking, but at the same time we **don't need multiple copies of the same member functions**?
- It is indeed **possible—to an extent**. We can have a void-list implementation, and a light-weight template implementation that redirect to the void list.  
Unluckily, our STL list implementation didn't do this.

PMOOP(0396A)-16.16

### How to write this?

How to implement the idea? Something like this:

```

template<class T>
class TList {
public:
    void Append(T* data) { _lst.Append(data); }
    void Prepend(T* data) { _lst.Prepend(data); }
    void Remove(TListNode<T>* node) { _lst.Remove(node->_node); }
    // Call GetFirst of the internal TVoidList.
    // And use it to create a "typed" list node
    TListNode<T>* GetFirst() const { return TListNode<T>(_lst.GetFirst()); }
private:
    TVoidList _lst;
};

```

Nothing special is done, everything is redirected to the void list.

We still have a typed list-node, so we will need to write more...

PMOOP(0396A)-16.17

### The TListNode

TListNode is just similar, redirecting all calls to the void list implementation.

```

template<class T> class TList;

template<class T>
class TListNode {
public:
    friend class TList<T>;
    TListNode(TVoidListNode* n) _node(n) {}
    TListNode* GetNext() const { return _node.GetNext(); }
    T* GetData() const { return reinterpret_cast<T*>(_node.GetData()); }
    void ChangeData(T* data) { _node.ChangeData(data); }
private:
    TVoidListNode _node;
};

```

So we still need `reinterpret_cast`, but is deeply **hidden in the implementation**. User code does not need it, and has full type checking.

The template thus performs type checking for us.

PMOOP(0396A)-16.18

PMOOP(0396A)-16.19

### The need of special cases...

- Sometimes we want a set of classes which behaves the same for most types, **except for some special classes**.
- E.g., we might want a list of type T, so that it uses the void\* implementation if **T is a pointer type**, and creates a copy of complete code if T is any other type.
- Or we might want a vector of type T which uses an array of type T, unless T turns out to be **bool**, and in such case use each char to represent 8 elements.
- What we want: **most of the time** we want one implementation, but if the **template parameters matches some criteria**, we use another implementation.
- C++ support this idea by **template specialization**. The idea is, we **define the general case** as normal, and then **add specific cases** where the compiler shouldn't generate code normally.

### Special case for one type

The **bool** vector special case is the easier case, because **it works for a specific type only**.

```

template<class T>
class TVector {
public:
    TVector(int size) {
        _impl = new T[size];
    }
    ~TVector() {
        delete[] _impl;
    }
    T& operator[](int n) {
        return _impl[n];
    }
    ...
private:
    T* _impl;
};

template<> // Still need this to define template!
class TVector<bool> {
public:
    TVector(int size) {
        int numChar = (size + 1) / 8;
        _impl = new (unsigned char)[numChar];
    }
    // Destructor still needed
    // Return a special object which will set
    // the right bit when assigned
    TBoolRef operator[](int n) {
        return TBoolRef(_impl[n/8], 1 << (n%8));
    }
private:
    unsigned char* _impl;
};

```

PMOOP(0396A)-16.20

PMOOP(0396A)-16.21

### Some comments

- To the user, it doesn't seem that anything special happened. **They just use a bool vector like a normal vector**.
- Internally, a bool vector has a **completely different implementation** from a normal vector.
- For this to work, the **compiler has to read both** template classes. So usually they are defined in the same header file.
- Once the "bool" type is fixed, there is **nothing that varies** in the template. So we need no template parameter. But we **still have to define it as a template**, since it is a template specialization.
- And... STL has this. If you creates a `vector<bool>`, **its data is packed into ints**, so it is a space-saving bool vector. Unlike an array of bools, which are actually an array of ints. (It uses 32 bits to store just 1 bit of information!)

### Specializing one class of types

Finally, we can also perform specialization by a **pattern**, e.g., use special code if the template parameter is a pointer.

E.g., if we want to use a **void\*** vector for vector of pointers, then...

```

// Special case for pointers
template<class T>
class TVector<T*> {
public:
    TVector(int size): _impl(size) {}
    T& operator[](int n) {
        return reinterpret_cast<T*>
            (_impl[n]);
    }
    ...
private:
    TVector<void*> _impl;
};

template<>
class TVector<void*> {
public:
    TVector(int size) {
        _impl = new (void*)[size];
    }
    ~TVector() { delete [] _impl; }
    void& operator[](int n) {
        return _impl[n];
    }
private:
    void* _impl; // array of pointers
};

```

PMOOP(0396A)-16.22

PMOOP(0396A)-16.23

### Final comments

- Note that we did **two** specializations here: one for void pointers, one for any other pointers.
- The latter is called a **partial specialization**.
- So if we create a `TVector<int>`, the `T*` specialization will be used. If we create a `TVector<void*>`, the `void*` specialization is used.
- This allows us to **avoid many copies of long functions if possible**, while at the same time allows for a quick vector if needed.
- And this **can work with the TVector<bool> specialization**, so the compiler is quite intelligent in choosing a specialization to instantiate.