VE280 Recitation Class Notes

VE280 SU19 TA Group

UM-SJTU Joint Institute

July 10, 2019

Source code available at `https://github.com/ve280/VE280-Notes`
Slides modified from the version of YAO Yue's, with permission

Table Of Contents I

RC Week 9

new Operator, Deep Copying, RAII and Resource Management

Overview

This chapter we dealt with a bunch of all related cocepts, for a better under standing we would first clarify the following:

- **Resource Management** is the problem we are tackling. Resources includes "Memory", files, IO Devices (printers, Internet connections) and Locks, Mutexes, Semaphores when you learned VE482/EECS482. Often there is a limited amount of them and programs share them.
- **RAII**, stands for **R**esource **A**cquisition **I**s **I**nstantiation. It's the idea behind all these messy rules. The key problem in RAII is to identify **ownership**.
- **Deep Copying** is the technique is used to implement RAII.
- **`new` and `delete`** simply represents the one most common resource used in programs.

Need for the `new` and `delete`

We once again examine the necessity for the two operators:

- Programs may require an statically unknown number objects. For examples, for a database (think of fancy spreed sheet!) application number of entries in a table is unknown when the program is designed.
- The precise time for the creation (allocation) and release of these objects are unknown in compile-time.

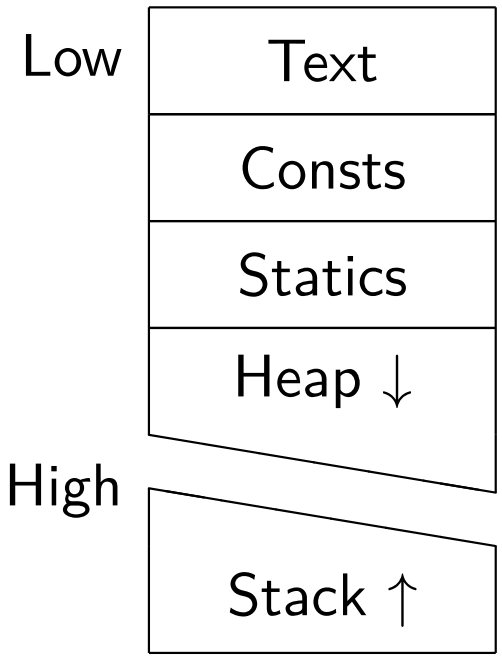
This calls for a mechanism that

- Allocates **the object** on-demand.
- Destroys **the object** on-demand.

It just happens that these objects requires memory. One should always understand that `new` is different than `malloc` in that it does not just allocate memory. It allocates objects, it just happens that these objects requires memory!.

The Heap

This graph is up-side-down, This is the same graph in page 103.



- Since we cannot decide when to allocate and when to destroy the object we surely CANNOT allocate the objects from the stack. The only way is to allocate them from a separate region, which is the Heap.
- The name is strange in that there is no commonly agreed upon origin. According to TAOCP by Knuth several authors starts to using it and everybody follows through.
- Heap is usually much bigger than stack. You should allocate large amount of memory only on heap. But that's not the decisive reason.

new and new[]

new and new[] does the following:

- Allocates space in heap (for one or a number of objects).
- Constructs object in-place (including, but not limited to ctor).
- Returns the "first" address.

The syntax for new operator are very simple

```
Type* obj0 = new Type;    // Default construction
Type* obj1 = new Type(); // Default construction
Type* obj2 = new Type(arg1, arg2);
Type* objA0 = new Type[size]; // Default cons each elt
Type* objA1 = new Type[size](); // Same as obj A0
```

For array allocation has no easy to use "construct with argument" option. In fact array allocation is seldom useful. You should always consider std::vector instead when you learned STL.

delete and delete[]

Since new-expressions allocates memory from the heap, they essentially requested (and occupies) resources from the system. For long running programs resources must always be returned (or released) when the program is finished with them, otherwise the program will end up draining all system resources, in our case running out of memory.
delete and delete[] releases the objects allocated from new and new[] respectively. They does the following:

- Destroy the object (each object in the array) being released (by calling the *destructor* of the object).
- Returns the memory to the system.

We must emphasize that delete an object more than once, or delete an array allocated using new[] by delete instead of delete[] cause undefined behavior!

Memory Leaks

Now the problem of memory leak seems obvious. If an object is allocated, but not released after the program is done with it, the system would assume the resource is still being used (since it won't examine the program), but the program will never use it. Thus resource is "leaked", i.e. no longer available for using. In our case the leaked resource is memory.

```
void foo() {int* p = new int(0); /* Code */}
```

The function `foo` causes memory leak each time it is being called. When the function returned the handle of the resource, i.e. the address (stored in `p`), is lost, thus the program is unable to release the memory. Thus we have a memory leak.
Technically memory leak represents all situations where resources are not released once the program is done with it. The following program also "leak" resources in the technical sense.

```
IntPointerSet s;
void foo() {int* p = new int(0); s.add(p); /* Code */}
```

Fixing Memory Leaks

Fixing memory leaks is by no means as simple as you might thought! You might thought the following code fixes the `foo` once and for good.

```
void foo() {int* p = new int(0); /* Code */ delete p;}
```

But what if an exception is thrown in `/* Code */`? The function will return immediately at the point of exception, skipping the final `delete` statement. You could

```
try{/* Code */}catch(...){delete p; throw;}
```

But I trust you know the draw back!. Another problem is what if `foo` has multiple return point? Will you be able to remember to release everything whenever you `return`?. What if above two situation are mixed?
All these problem suggest we need a better way of handling the problem! We need a strategy other than patches!

Digression: Diagnosing memory related problem

`valgrind` is not a tool that only looks for memory leaks. It actually looks for for all sorts of memory related problems, including:

- Memory Leaks.
- Invalid accesses, array out-of-range, use of freed memory, etc.
- Double free problems.

These problem are most likely UBs so bugs related to them are very likely to be elusive and random. Always use `valgrind` to check your program. It significantly improves your success rate on OJ.
Valgrind can also act as a *profiler*. A profiler measures how much time is spent on each portion of your code. It let you to know which portion is taking up the most time. Profiling is always the first step towards performance optimization.
Valgrind does all these by tap into (hijacks) EVERY function call of your program, which comes with a truely significant performance cost.
For more information, RTFM, thank you.

Lifetime of objects

We now take a little digression. The *lifetime of an object*, sometimes "life cycle", is the period starting from creation of object, to the point the object is destroyed. We typically has (roughly) 3 of them

- Global objects. They are created when the program starts and destroyed when the program is terminated. Including global variables and `static` variables.
- Automatic objects. Basically local variables.
- Dynamic objects. Objects created/deleted on-demand through `new` and `delete`.

From the lifetime view resource leaks are simply dynamic objects whose lifetime should have been terminated when they are no longer useful. It should be clear that the lifetime of the objects is tightly related to the *scope* of the variable. This is natural. Since global objects can be access by any portion of the program they have to exist throughout the program. Local objects are only visible in the enclosing brackets thus they are destroyed when they go out of scope.

Binding resource to the scope

The solution to our previous problem is to bind the resource to a scope where it is needed. The detailed way of doing so is:

```
// intset.h
const int MAXELTS = 100;
class IntSet {
    int *elts, sizeElts;
    int numElts;
public:
    IntSet(int size = MAXELTS)
        : elts(new int[size]),
          sizeElts(size),
          numElts(0) {}

    ~IntSet() { delete[] elts;}
    // Other methods unchanged
};
```

```
// driver.cpp
#include "ex.h"
class T {
    IntSet s1, s2;
// ommit other
}
// Nothing leaks
void foo() {
    IntSet s;
    while (...) {
        T t; if (...) throw -1;
    }
    if (...) return;
    else ...;
}
```

RAII

Stands for *Resource Acquisition Is Instantiation*. RAII is perhaps the one most famous rule specific to C++, unfortunately with an extremely terrible name! Recently people start referring to the rule by *Scope-based Resource Management*. It's a little bit clearer, but by no means summarizes it's full power. RAII requires the following:

Holding an resource is a class invariant!

With a little (?) bit explanation:

- Resource is allocated in ctors and ctors only.
- Resource is released in destructors (dtors).
- Object "owns" the resources. The resource is managed by the object and the object only.
- The resources would share it's life cycle with object. As long as there is no object leak there is no resource leak. Fortunately we know automatic objects are destroyed by the compiler when they go out of scope, impossible to have object leak. Valgrind is only

Destructors

Often denoted as `dtors` for short. Destructors should:

- Be named as `~ClassName`.
- Takes no argument and returns nothing (Not even `void`).
- If one expect the class to be inherited the dtor should be declared as virtual.
- Release resource allocated only in this class (don't release base class resources!).

The process of destroying an object is as follows:

- It calls the dtor of the class.
- Calls the dtors for each member of current class.
- Calls dtor of the base class.
- Does above recursively until no more dtors to invoke. Finally it releases the memory.

Dtor examples: Memory Leak

The following code causes memory leak problem.

```cpp
//classes.h
class Base {
protected:
    int *p;
public:
    Base() : p(new int(10)) {}
    ~Base() {delete p;}
};

class Derived : public Base {
    int *q;
public:
    Derived() :
     Base(), q(new int(20)) {}
    ~Derived() {delete q;}
};
```

```cpp
//driver.cpp
#include "classes.h"

void foo() {
    Base* ptrA = new Derived;
    delete ptrA; // Memroy Leak!
}

void foo() {
    Derived* ptrB = new Derived;
    delete ptrB; // Safe
}
```

Dtor examples: Double Free

The following code causes double free problem.

```
//ex2.h
class Base {
protected:
    int *p;
public:
    Base() : p(new int(10)) {}
    virtual ~Base(){delete p;}
};

class Derived : public Base {
    int *q;
public:
    Derived() :
     Base(), q(new int(20)) {}
    ~Derived()
    { delete q; delete p;}
```

```
//ex2.cpp
#include "classes.h"

void foo() {
    Base* ptrA = new Derived;
    delete ptrA; // Double Free!
}

void foo() {
    Derived* ptrB = new Derived;
    delete ptrB; // Double Free!
}
```

Dtor examples: Free only what you "own"

This is our correct solution.

```
//ex3.h
class Base {
protected:
    int *p;
public:
    Base() : p(new int(10)) {}
    virtual ~Base(){delete p;}
};

class Derived : public Base {
    int *q;
public:
    Derived() :
     Base(), q(new int(20)) {}
    ~Derived() {delete q;}
};
```

```
//ex3.cpp
#include "classes.h"

void foo() {
    Base* ptrA = new Derived;
    delete ptrA; // Safe!
}

void foo() {
    Derived* ptrB = new Derived;
    delete ptrB; // Safe!
}
```

Linked List example

To support our future discussion with a concrete example we must take the singly linked list example from the instructor. We would like to make a comment. Objects like the linked list are sometimes called "containers". The major purpose of containers is to container other objects, store and more importantly arrange them in a manner suitable for specific access. For example:

- Fast random access with few or no deletion at all? go for an array.
- Only access front? Frequent insertion at front? linked list.

However there are in general 2 types of containers:

- Container of value. The container only store the "value" of inserted object. The container does not give you back the exact same object that you gives it. It only gives back a object that has the same "value" you provided. It does NOT own the object.
- Container of pointer. Think of a value container whose value are pointers. It gives you back the actual object. When you insert an object into a container of pointer, you are transfering ownership!

Linked List Interface

```
class listIsEmpty {}; // An exception class
struct node{ node *next; int value; };
class IntList {
    node *first;
    void removeAll();
    void copyList(node *list);
public:
    bool isEmpty() const;
    void insert(int v);    // inserts v into the front of the lis
    int remove(); // Pops the first element
    void print() const; // print the int list
    IntList();                          // default constructor
    IntList(const IntList& l);  // copy constructor
    ~IntList();                         // destructor
    IntList &operator=(const IntList &l); // assignment operator
};
```

Linked List Implementation: ctor and dtor

```
bool IntList::isEmpty() const {
    return first == nullptr;
}

IntList::IntList(): first(nullptr) {}

IntList::~IntList() { removeAll(); }

void IntList::removeAll() {
     while(!isEmpty()) remove();
}
```

Resource management: An analysis

Keep in mind that an object "owns" the resources. Keep in mind as well that holding a resource is a class invariant. Whenever class invariant is in play we must question whether each operation breaks / preserves the invariant.

- Ctors and Dtors are checked already :=).
- `insert()` and `remove()` might break invariant.
- `print()` and `isEmpty()` are `const`.

But there are two cases that are more tricky.

- When you pass the object by value, the content of the object will be copied, byte by byte. Now both the "original" object and the "copy" both owns the same actual list.
- When you performs assignment a byte-wise copy will happen. In addition to the previous problem, the old object must give up it's ownership!

insert and remove

A kind note: draw the graph when confused. Remember boundary condition that `first` could be empty!

```
int IntList::remove() {
    if (isEmpty()) throw listIsEmpty();
    int result = first->value; // None empty, always possible
    node *victim = first;      // Why save to victim?
    first = first->next;
    delete victim;             // Giving up ownership
    return result;             // Value container
}

IntList::IntList(const IntList &l)
    : first (0) {
        copyList(l.first);
    }
```

Resource management: Copying

Keep in mind that an object "owns" the resources. A value container owns the location it used to store the values. A container to pointer owns both the location it used to store the pointers, but also the objects stored by the pointer.

These invariants are part of that objects, and must be preserved for both the copied object and the original object when passed into a function.

When a variable is passed into a function by value, the copy constructor of a function will be called. The copy constructor has the following form, argument type is critical!

```
class Type {
// Omit other methods
public:
    Type(const Type& type); // Copy constructor
}
```

Copy constructor

Just like the default constructor:

- A copy constructor is "synthesized" if not specified.
- For types like `int` copy constructors performs byte-wise copy.
- A synthesized copy constructor by default calls it's element's copy constructor.

All three rules combined provides the same behavior as in C when you pass a struct (class) by value. But the real process happened is listed above. But a copy constructor is after all a constructor and that gives us problems:

- If you only write a custom copy constructor, since there already exists one ctor, the default ctor will not be synthesized.
- If you choose to write a custom copy ctor, you must remember to call the copy-ctor on each of your element in initialize list. If you don't, your members will be default initialized.

This gets much more complicated combined with inheritance.

Implementing the copy ctor

This is very easy to understand recursive algorithm. A comment is that a copy ctor is also a ctor, which means all rules applicable to a regular ctor applies to copy ctor as well, for example the initialization list.

```
IntList::IntList(const IntList &l)
    : first (0) {
        copyList(l.first);
    }

void IntList::copyList(node *list) {
    if (!list) return; // Base case
    copyList(list->next);
    insert(list->value);
}
```

Operator Overloading

The first rule of thumb:

Operators are just functions.

And functions can be re-written. You can rewrite the effect of existing operator on your custom type. This is a very powerful tool of abstraction. For example you could overload $*$ on matracies, so that they looks as if they are regular values. It might surprise you how many operators can be overloaded:

- Arithematic logic operators: $<<$, $+$, $-$, $*$, $==$, $||$, $\&$...
- Unary logic: !,
- Special operators: [] (array), $->$, $*$ (pointers), () (function).
- Memory: `new` and `delete`
- And others ...

A few remark:

- Overloaded operators preserves their old precedence.
- Overloaded operators preserves their old associativity.

Overloaded operator =

It's better that to keep the original "behavior" of an overloaded operator. For example you might want to overload + with a function that preserves interchangeability. You might want to avoid overload operator = with a function that returns a `bool`. For the operator =

```
class Type {
public:
^^IType& operator= (const Type& rhs);
};
```

- The return type should be a reference to the left hand side.
- The keyword `operator` indicates opearator overloading. You can think the name of the function is `operator=`.
- The argument type must be a const reference to the right hand side.

Again if you don't overload the assignment operator, one will be synthesized for you, which follows similar rules as the cp-ctor.

Implementing the overloaded assignment operator

```
IntList &IntList::operator= (const IntList &l) {
    if (this != &l) {
        removeAll();
        copyList(l.first);
    }
    return *this;
}
```

In addition to the copying the overloaded assignment operator needs to first release the resources of the left hand side. Because the left hand side is letting go the ownership of old resources, and taking over new resources from the compied from list.
There is a caveat. In the copy constructor the destination is guaranteed to be different from the source (why?). However one might assign a value to itself. What problem would there be if the branch `if (this != & l)` is not there?

Move Semantic: The problem

Consider the following code, suppose we have a function combine

```
IntList combine(const IntList& l1, const IntList& l2);}
```

In a function `foo`

```
void foo() {
^^IIntList l1; // Populate l1 with much data
^^IIntLIst l2; // populate l2 again with data
^^IIntList l3 = combine(l1, l2);
}
```

We examine what will happen on the third line.

- On return of `combine`, a temporary object will be constructed.
- The temporary object will be used to copy construct `l3`.
- The temporary object will be deleted.

You should realize that the second step comes with a significant performance cost.

Move Semantic: An analysis

We now take a step back and think: We don't need to **copy** the content of temporary object. What we really need is to **move** the resources from the temporary object to the target object. We need a mechanism to transfer the ownership of resources from the temporary object to the target. The language features supporting this is called move semantics.

We know that the temporal object is a rvalue, but in order to do what we expected we need to modify the temporal object, i.e. pass the temporal object by reference of some sort:

```
class Type {
public:
    Type(Type&& rhs); // A move constructor
};
```

Here `Type&& rhs` declares a *rvalue-reference*.

Move Semantic: Implementation

```
IntList::IntList(IntList&& rhs)
    : first(rhs.first) {
    rhs.first = nullptr;
}
```

Remember move constructor is also a ctor. In this very simple code it exchanges resources owned by the original object (which is empty) and the temporal object. Note argument is not consted.

Another similar situation is the case of assignment operator:

```
Type& operator=(Type&& rhs); // Move assignment
```

And the implementation

```
IntList& IntList::operator=(IntList&& rhs) {
^^Iswap(this.first, rhs.first);
}
```

It is equally simple. The resources of the original object will be freed during the deconstruction of the temporal object.

The rule of big X

Where $X = 3$ traditionally and $X = 5$ after c++11.

Whenever an object owns resources, any resources, not just memory, it should implement 5 methods:
A ctor and a dtor, A copy ctor, a move ctor, a copy assignment operator, and a move assignment operator.

These are 5 typical situations where resource management and ownership is critical. You should never leave them unsaid. If you want to use the version synthesized by the compiler, use `=default`:

```
Type(const Type& type) = default;
Type& operator=(Type&& type) = default;
```

Traditionally constructor/destructor/copy assignment operator forms a rule of 3, you should know this if being questioned in the exam. Move semantics is a feature available after C++11.

Singleton Pattern

Very often some object is "global". It is used by the entire program (for example `World` class). Probably this object should not be copied, or more likely cannnot be copied. Probably this object can only be validly instantiated once.
Using a global variable is one possible solution, but hardly a good one. For instance if you have multiple global instances the order of their instantiation is not guaranteed.
A clean slate solution is called *the singleton pattern*:

- Clients use a `static` method to access the object.
- Object is created on first access, than return every time.
- Implement `private` constructor.
- Use deleted copy ctor and deleted assignment operator to explicitly signify the client that this object is not copy-able.

Singleton pattern: Implementation

```
// s.h
class S {
        static S* instance;
        S() {}                          // Constructor
    public:
        static S& getInstance() {
            if (instance == nullptr) instance = new S;
            return *s;
        }
        S(S const&)                     = delete;
        S&   operator=(S const&)  = delete;
};
// s.cpp
S* S::instance = nullptr; // initialize
// foo.cpp
void foo() { S& s = S::getInstance(); /* Use s as you wish */}
```

Problems with exception

Quotation from Google C++ Style Guide:

> *We do not use C++ exceptions. ... Our advice against using exceptions is not predicated on philosophical or moral grounds, but practical ones. ... Things would probably be different if we had to do it all over again from scratch.*

The Criticism

Again we quote from the same document:

> *When you add a throw statement to an existing function, you must examine all of its transitive callers. Either they must make at least the basic exception safety guarantee, or they must never catch the exception and be happy with the program terminating as a result.*

https://google.github.io/styleguide/cppguide.html#Exceptions

Exception Safety

There are in general 3 levels of exception safety:

No-throw Guarantee The function will not throw an exception or allow one to propagate.

Strong Guarantee If a function terminates because of an exception, it will not leak memory and no data will not be modified, as if it has never been called from the beginning.

Basic Guarantee If an exception is thrown, no resource (memory) is leaked ... though the data might have been modified.

Understand these guarantees as part of the specification for your function. Exceptions allows the function to interact with the outside world (by throwing exceptions) and it requires the outside world to change accordingly (catch exceptions).

From MSDN 0https://msdn.microsoft.com/en-us/library/hh279653.aspx

No-throw guarantee

No-throw guarantee is very strong guarantee! You might think this is easy. You might thought as long as your function does not throw anything than you are safe. However consider the following example:

```
int g(int); int foo(int i) { return g(i); }
```

In this example, `foo` does not give no-throw guarantee. It g could throw an exception so that `foo` fowards an exception. But can we make `foo` no-throw?

The only way of doing this is to allow `foo` to catch everything. But since `foo` does not have the faintest idea what kinds of exception g might throw (since exception are not part of abstraction specification), it cannot catch them without causing problems.

In this example it is simply impossible to achieve no-throw guarantee.

Strong Exception Safety

Now we take a step back. What about strong exception safety. At first look it seems strong exception safety is easier to achieve since a function need only to look over it's own stuff. But this is very difficult as well. Consider the following code:

```
int divide(IntList& stack) {
    int num = stack.pop();
    int denum  = stack.pop();
    if (denum == 0) throw ExceptionDivZero();
    return num / denum;
}
```

At first look you might think this is perfect. But this code is not strong exception safe. This function has side-effect (still remember what are side effects?), and side effects must be reversed before exiting from `divide`. Unfortunately not all side-effects are reversible (within reasonable performance overhead).

Basic Exception Safety

Now we seek the least. Can we achieve at least basic safety. Well, the good news is we can, but by no means easy!
Note RAII takes most of the trouble away. If you follow RAII and only explicitly use local objects (so they are destroyed automatically by the compiler) you are already on the safe track.
When a exception happens, the compiler performs *stack unrolling* (see page 116 of the slides).
But there is one corner case. We start by asking what happens if a constructor has to report an error.

- If an construction fails, which indicates the invariance of an object cannot be satisfied, so that an object cannot be valid.
- Constructors do not have return value.

It seems we are cornered, but we do have choices.

Error handling in constructors

One way of handling this is to create a "dummy" object, a tombstone. This is used in `fstream` in standard library. If you construct a file string with a invalid file, you would have a `fstream` that is in a failed state. It is not bind to any resource (it owns nothing). You can use a method (`fstream::is_good()` in our case) to determine it's state.
This have the same problem of error code! (What are they?)

Another approach is to throw an exception in a constructor (yes you can do that!). However, since we are still in the constructor, this indicates the object itself has not been valid (i.e. it does not really exist). When a exception is thrown:

- Terminate the function and unroll stack as usual.
- The destructor will NOT be invoked.
- It's members are already valid object, so they will be destructed.

Exceptions in constructors

The rules can easily gets extremely complicated. First of all, since destructor of an object might be called in the process of throwing an exception (the exception is raised, but is not brought to process), it will be extremely bad if another exception is thrown. So that a destructor may not throw any exception.

Secondly we note a few complicated situations.

- If a member of class throws an exception in the process of initialization, the members that have been initialized will be destroyed.
- If one of the objects throw an exception in initializing an array of objects obtained by `new[]`, the objects that already constructed will be destroyed.

And there are more of them. They are so tricky that the only way to safely deal with is to follow RAII strictly.

Smart Pointers

The problem of ownership often comes with pointers. We use pointers for two very different reasons:

- To signify owner ship.
- To store a reference of an object.

```
Type* getNewObject(int param) {
    Type* obj = new Type; /* Calculation */ return obj;
}
```

Above code essentially transfer the ownership of the new object to it's caller. The caller must be aware that it is accepting an ownership, thus it should be in charge of releasing it.

Further more it could happen an object is shared among multiple modules. The ownership is not clear, and we might not know which module finishes using it last.

Smart Pointers

The standard library features 3 smart pointers:

```
std::unique_ptr<Type> ptr;   // Unique ownership
std::shared_ptr<Type> s_ptr; // Shared ownership
std::weak_ptr<Type> s_ptr;   // No Ownership
```

Unique pointer features a complete ownership. This pointer cannont be duplicated by copying. It can only be transfered through `move`.
Shared pointer indicates shared ownership. Multiple owner "shares" the object. Reference counting it used to indicate how many instances is using the object. When reference counting goes to zero, the object is released.
Weak pointer indicates no ownership at all. Smart pointers supports syntax just like pointers. For more information RTFM.