COMP6771
Advanced C++ Programming

Week 4
Part One: Copy Control (continued) and Move Semantics

2016

www.cse.unsw.edu.au/˜cs6771

Inline Constructors, Accessors and Mutators

- Question (from 2015): In the week 3 examples, constructors and getters/setters were defined inside the class delcaration. However, we've been told to seperate declarations (.h) and definitions (.cpp).

Inline Constructors, Accessors and Mutators

- Answer: Remember the .h file, is the "public" interface, so everyone can see all the function declarations and data members (both public and private).
- Your class' data members are not secret, but how they are used might be.
- Therefore, the specific code implementation of methods should be in the .cpp file.
- However, simple constructors, getters, and setters that are not complex may be inlined/defined in the class declaration.
- See also: http://google-styleguide.googlecode.com/svn/trunk/cppguide.html#Inline_Functions

Copy Control and Resource Management

- Use copy-control members to management resources (e.g., memory, file and socket handles)
- Two general strategies:
  - Value-like classes (with value/copy semantics)
    - Class data members have their own state
    - When an object is copied, the copy and the original are independent of each other
  - Pointer-like classes (with reference/pointer semantics)
    - Class data members share state
    - When an object is copied, the copy and the original use the same underlying data
    - Changes made to the copy also affect the original, and vice versa

Let us write a value-like stack class.

Interface: UB_stack.h

```
#ifndef UB_STACK_H
#define UB_STACK_H

class UB_stack {
public:
  UB_stack();  // default constructor
  UB_stack(const UB_stack &s);  // copy constructor
  UB_stack(UB_stack &&s);  // move constructor
  ~UB_stack();  // destructor
  UB_stack& operator=(const UB_stack &s);  // copy assignment
  UB_stack& operator=(UB_stack &&s);  // move assignment
  void push(const int &item);
  void pop();
  int& top();
  const int& top() const;
  bool empty() const;
  bool full() const;
private:
  class Node;   // forward declaration of nested class
  void reverse(Node *);  // private method
  Node *head_;

  friend void swap(UB_stack &s1, UB_stack &s2);
};

void swap(UB_stack &s1, UB_stack &s2);

#endif /* UB_STACK_H */
```

Implementation: UB_stack.cpp I

```
#include "UB_stack.h"

// nested class
class UB_stack::Node {
  // allow UB_stack to access private data
  friend class UB_stack;
  // methods and data in this class are private by default
  Node(int i, Node *n = nullptr) : item_{i}, next_{n} { }
  ~Node() { delete next_; }  // destructor cleans up the memory

  int item_;
  Node *next_;
};

// constructor
UB_stack::UB_stack() : head_{nullptr} { }

// copy constructor
UB_stack::UB_stack(const UB_stack &s) : head_{nullptr} {
  reverse(s.head_);
}
```

Implementation: UB_stack.cpp II

```
// destructor
UB_stack::~UB_stack() { delete head_; }

// return the top of the stack
int& UB_stack::top() { return head_->item_; }
const int& UB_stack::top() const { return head_->item_; }
bool UB_stack::empty() const  return head_ == 0;
bool UB_stack::full() const  return false; ;

// method to work down a given stack
// and push items onto "this" stack correctly
void UB_stack::reverse(Node *h) {
  if (h != nullptr) {
    reverse(h->next_);
    push(h->item_);
  }
}

// method to push an int onto the stack
void UB_stack::push(const int &item) {
  head_ = new Node (item, head_);
}
```

Implementation: UB_stack.cpp III

```
// copy assignment operator
UB_stack& UB_stack::operator=(const UB_stack &s) {
  // if not already the same stack
  if (this != &s) {
    delete head_;  // delete this stack
    head_ = nullptr;
    reverse(s.head_); // copy data from other stack
  }
  return *this;
}

// pop off the top of the stack
void UB_stack::pop() {
  Node *t = head_;
  head_ = head_->next_;
  t->next_ = nullptr;
  delete t;
}
```

Implementation: UB_stack.cpp IV

```
// return the top of the stack
int& UB_stack::top() { return head_->item_; }
const int& UB_stack::top() const { return head_->item_; }

bool UB_stack::empty() const { return head_ == 0; }
bool UB_stack::full() const { return false; };

// move constructor
UB_stack::UB_stack(UB_stack &&s) :  head_{s.head_} {
  s.head_ = nullptr;
}

// move assignment
UB_stack& UB_stack::operator=(UB_stack &&s) {
  if (this != &s) {
    delete head_;
    head_ = s.head_;
    s.head_ = nullptr;
  }
  return *this;
}

```

Implementation: UB_stack.cpp V

```
void swap(UB_stack &s1, UB_stack &s2) {
  using std::swap;
  // swap the pointers to the heads of the list only
  // much faster than swapping all the data
  swap(s1.head_, s2.head_); // call std::swap on the pointers
}
```

Client Code

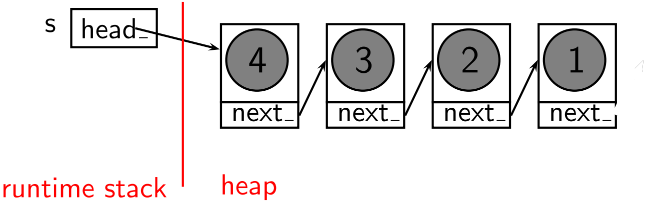
```
#include <iostream>
#include "UB_stack.h"

int main() {
  UB_stack s;
  s.push(1);
  s.push(2);
  s.push(3);
  s.push(4);
  s.pop();
  std::cout << s.top() << std::endl;
  // ...
}
```

The Compiler-Generated Destructor vs. Ours

```
UB_stack::~UB_stack() { } // compiler generated destructor
```

- Client code:

```
{ // new scope
  UB_stack s;
  s.push(1);
  s.push(2);
  s.push(3);
  s.push(4);
} // ~UB_stack()
```



- At the closing scope }:
  1. The destructor is called: do nothing
  2. The members' destructors are called: do nothing

  The stack object has leaked!

- Our destructor works as expected (see code).

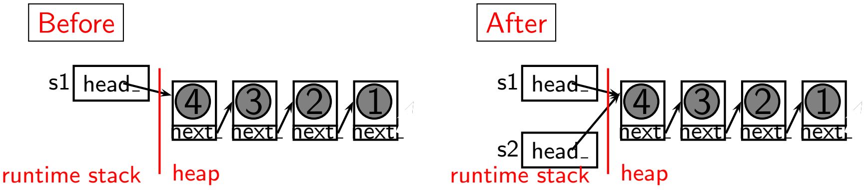
The Compiler-Generated Copy Constructor

- Shallow copy:

```
UB_stack::UB_stack(const UB_stack &s) : head_{s.head_} { }
```

- Example:

```
UB_stack s2 {s1}; // copy construction
```



- Failed to provide value-like semantics
- Potentially lead to memory corruption errors!

Our Copy Constructor

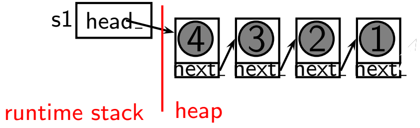
- Deep copy:

```
UB_stack::UB_stack(const UB_stack &s) : head_{nullptr} {
  reverse(s.head_);
}
```

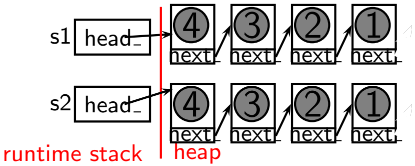
- Example:

`UB_stack s2 {s1}; // copy construction`

Before



After



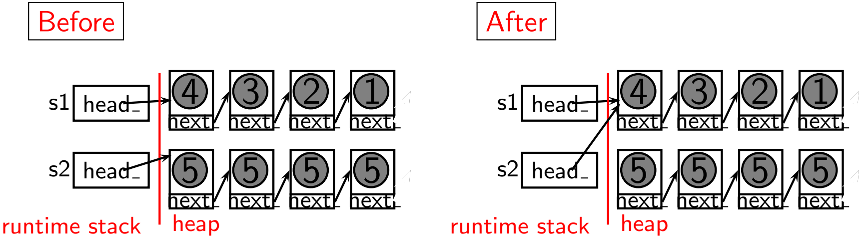
The Compiler-Generated Copy operator=

- Shallow copy:

```
UB_stack& UB_stack::operator=(const UB_stack &s) {
  head_ = s.head_;
  return *this;
}
```

- Example:

  `s2 = s1;`



- Failed to provide value-like semantics
- Potentially lead to memory corruption errors!

The memory pointed by s2.head_ has leaked!

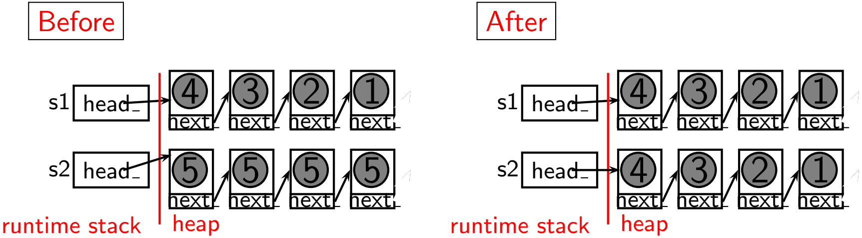
Our Copy operator=

- Deep copy:

```
UB_stack& UB_stack::operator=(const UB_stack &s) {
  if (this != &s) {
    delete head_;
    head_ = nullptr;
    reverse(s.head_);
  }
  return *this;
}
```

- Example:

```
s2 = s1;
```



The memory pointed by s2.head_ before has been freed!

Limitations of Copy Semantics

- Copy constructor called on the returned object of a non-reference type:

```
UB_stack makeStack () {
  UB_stack s;
  int i;
  while (cin >> i)
    s.push(i);
  return s;
}

UB_stack s1 = makeStack();
```

The compiler may optimise some calls to copy ctors away.

Move Semantics

- Using swap:

```
void stack_swap(UB_stack &s1, UB_stack &s2) {
  UB_stack tmp = s1;  // copy constructor
  s1 = s2;            // copy assignment
  s2 = tmp;           // copy assignment
}                     // destructor for tmp
```

- Can we simply swap the internal resources in s1 and s2?
- Yes, we can in C++11:
  - Understand lvalue reference (&) and rvalue references (&&)
  - Understand the move semantics

Move Semantics (for Improved Performance)

The move semantics allows you to avoid unnecessary copies when working with temporary objects that are about to evaporate, and whose resources can safely be taken from that such a temporary object and used by another.

Why Move Semantics?

- Can we do copy construction/initialisation efficiently?

```
Sales_data src;
Sales_data dst = src;
```

  - Copy `src` into `dst` if `src` persists, i.e., will be used again or
  - Move the internal resources of `src` into `dst` if `src` is a temporary object, i.e., one that will be destroyed or assigned to

- Can we also perform assignment efficiently?

```
Sales_data dst;
dst = src;
```

1. Destroy `dst`
2. Assign to `dst`:
   - Copy `src` into `dst` if `src` persists, i.e., will be used again or
   - Move the internal resources of `src` into `dst` if `src` is a temporary object, i.e., one that will be destroyed or assigned to

Interface (+ Move Semantics): UB_stack.h

```
class UB_stack {
public:
  // copy constructor
  UB_stack(const UB_stack &s);
  // move constructor
  UB_stack(UB_stack &&s);

  UB_stack& operator=(const UB_stack &s);
  // move assignment
  UB_stack& operator=(UB_stack &&s);

  ...
};
```

By distinguishing lvalues references from rvalue references:

- Copy and move constructors are overloaded
- Copy and move assignment operators are also overloaded

Implementation (+ Move Semantics): UB_stack.cpp

```
#include "UB_stack.h"
// move constructor
UB_stack::UB_stack(UB_stack &&s) : head_{std::move(s.head_)} {
  s.head_ = nullptr;
}

// move assignment
UB_stack& UB_stack::operator=(UB_stack &&s) {
  if (this != &s)   {
    delete head_;
    head_ = std::move(s.head_);
    s.head_ = nullptr;
  }
  return *this;
}
```

After the “resources” have been stolen, i.e., from the moved-from object, its data members must be modified in order to put it in a valid state (to be destroyed by its destructor).

The Synthesised Move Constructor/Assignment

- Synthesised only if none of the Big Three is provided
- Move constructor/assignment: member-wise move
  - Call a member's move constructor/assignment to move
  - The members of built-int types are copied directly
  - Array members are copied by copying each element
- The synthesised solutions for `UB_stack` are wrong:

```
#include "UB_stack.h"
// move constructor
UB_stack::UB_stack(UB_stack &&s) : head_{std::move(s.head_)}
noexcept { }
// move assignment
UB_stack& UB_stack::operator=(UB_stack &&s) noexcept {
  head_ = std::move(s.head_);
  return *this;
}
```

The Compiler-Generated Move Constructor

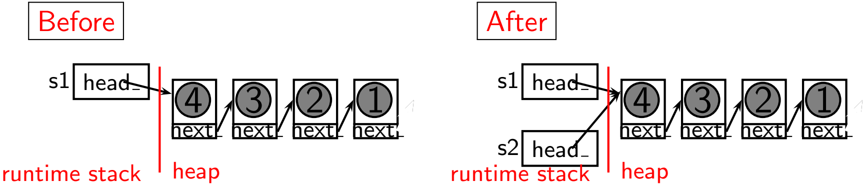
The same as when the synthesised copy constructor is used

- Stealing from the Moved-From Object:

```
UB_stack::UB_stack(UB_stack &&s) noexcept :
head_{std::move(s.head_)} { }
```

- Example:

  `UB_stack s2 = std::move(s1);`



When the moved-from object dies, the destructor for it is called. The commonly shared stack will be freed
$\Longrightarrow$ s2.head_ points to something that has been freed!

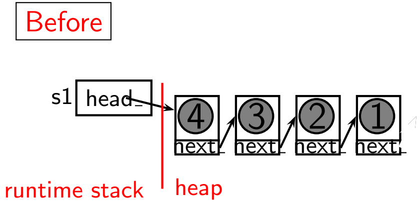
Our Move Constructor

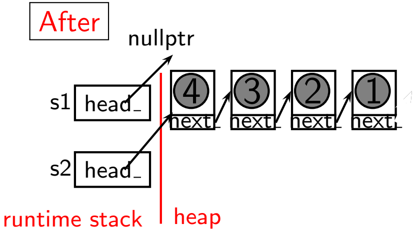
- Move Semantics:

```
UB_stack::UB_stack(UB_stack &&s) : head_{std::move(s.head_)} {
  s.head = nullptr; // put the moved-from object
}                   // in a valid state to be destroyed
```

- Example:

```
UB_stack s2 = std::move(s1);
```





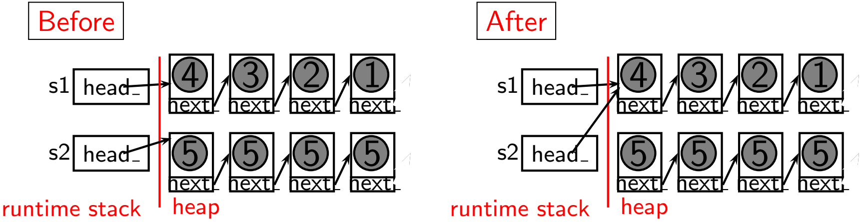
The Compiler-Generated Move operator=

The same as when the synthesised Copy operator= is used

- Stealing from the moved-from object:

```
UB_stack& UB_stack::operator=(UB_stack &&s) noexcept {
  head_ = std::move(s.head_);
  return *this;
}
```

- Example: s2 = std::move(s1);



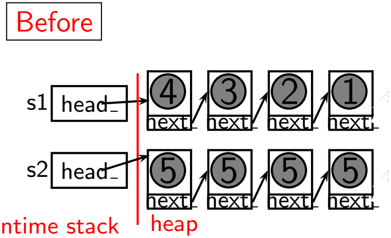
- Failed to provide value-like semantics
- Potentially lead to memory corruption errors!

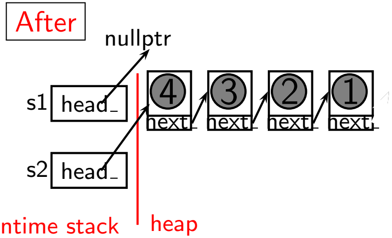
The memory pointed by s2.head_ has leaked!

Our Move operator=

```
UB_stack& UB_stack::operator=(UB_stack &&s) {
  if (this != &s) {
    delete head_;
    head_ = std::move(s.head_);
    s.head_ = nullptr;
  }
  return *this;
}
```

- Example: s2 = std::move(s1);





The memory pointed by s2.head_ before has been freed!

noexcept

- Signals to the compiler and optimiser that no exception will be thrown from the method.
- Automatically provided for compiler synthesised big five.
- Also: http://en.cppreference.com/w/cpp/language/noexcept_spec

Lvalues and Rvalues Revisited

- Lvalues: an object's identity or address:

```
int i = 1;          // i is an lvalue
int& getRef() {
  return i;
}
getRef() += 1;      // getRef() is an lvalue
                    // i = 2
```

- Rvalues: an object's value

```
int getVal() {
  int i = 0;
  i = i + 1;      // i + 1 is an rvalue
  return i;
}
getVal() += 1;  // error since getVal() is an rvalue
```

Lvalue and Rvalue References

- An lvalue reference is formed by placing an & after some type:

```
A   a;
A& a_ref1 = a;   // an lvalue reference
```

- An rvalue reference is formed by placing an && after some type:

```
A a;
A&& a_ref2 = a + a;   // an rvalue reference
```

- An rvalue reference behaves just like an lvalue reference except that it can bind to a temporary (an rvalue), whereas you can not bind a (non const) lvalue reference to an rvalue.

```
A&  a_ref3 = A{};   // error!
A&& a_ref4 = A{};   // ok
```

Lvalue vs. Rvalue References (Cont'd)

In general

- Lvalue references are persistent – every variable is an lvalue reference
- Rvalue references are bound to objects that
  - are about to be destroyed, and
  - don't have any other user any more.

Rvalue references identify temporary objects.

Example Lvalue and Rvalue References

- Lvalue references:
  - Functions that return lvalue references
  - ++i
  - *p
  - a[2]
- Rvalue references:
  - Functions that return non-reference types
  - i++
  - i + j
  - i < k

  where the result in each case will be stored in a compiler-generated temporary object.

Our stack_swap for UB_stack

- The one written for copy semantics:

```
void stack_swap(UB_stack &s1, UB_stack &s2) {
  UB_stack tmp = s1;  // copy constructor
  s1 = s2;            // copy assignment
  s2 = tmp;           // copy assignment
}
```

- The one written for move semantics:

```
void stack_swap(UB_stack &s1, UB_stack &s2) {
  UB_stack tmp = std::move(s1);  // move constructor
  s1 = std::move(s2);            // move assignment
  s2 = std::move(tmp);           // move assignment
}
```

- Every variable/lvalue reference/rvalue reference is an lvalue
- `std::move` converts its argument into an rvalue reference so that the move-related copy-control members can be called.
- `std::move` is a potentially destructive read

swap in the C++ Library

No need to write `stack_swap`. There is one in the C++ library.

```
template<class T>
void swap(T& a, T& b) {
  T tmp = std::move(a);
  a = std::move(b);
  b = std::move(tmp);
}
```

Argument-Dependent Lookup (ADL)

```
namespace A {
  struct X { };
  void f(const X&) {
  }
}

int main() {
  A::X x;
  f(x);  SAME as A::f(x)
}
```

1. First, the normal name lookup for `f` is performed
2. Then, look for `f` in *the namespace scope where* `x` *is defined*.

Why ADL

http://www.gotw.ca/publications/mill08.htm

Criticisms

http://en.wikipedia.org/wiki/Argument-dependent_name_lookup#Criticism

Interface (+ Specialised swap): UB_stack.h

```
class UB_stack {
friend void swap(UB_stack &s1, UB_stack &s2);
public:
  // copy constructor
  UB_stack(const UB_stack &s);
  // move constructor
  UB_stack(UB_stack &&s);

  UB_stack& operator=(const UB_stack &s);
  // move assignment
  UB_stack& operator=(UB_stack &&s);

  ...
};
// the declaration is needed still
void swap(UB_stack &s1, UB_stack &s2);
```

Provides a specialised, faster version than `std::swap`

Implementation (+ Specialised swap): UB_stack.cpp

```
#include "UB_stack.h"

...

void swap(UB_stack &s1, UB_stack &s2) {
  using std::swap;
  // swap the pointers to the heads of the list only
  // much faster than swapping all the data
  swap(s1.head_, s2.head_); // call std::swap on the pointers
}

...

```

The using std::swap is important:

- Use a type-specific version of swap via ADL if it exists
- Otherwise, use the one from using std::swap

Carefully read §7.3.

Some Advice

- Use STL containers whenever possible as you don't have to worry about copy control – done for you (Assignment 1)
- Sometimes, you need to write your own containers
  - If you want your class to behave like a value, you need to manage your own copy control (Assignment 2)
  - If you want your class to behave like a pointer, you can use and/or develop smarter pointers with reference counting (covered later)

Readings

- Chapter 13
- Rvalue references:
  http://thbecker.net/articles/rvalue_references/section_01.html
- Move semantics:
  http://www.drdobbs.com/move-constructors/184403855
- Will look at perfect forwarding when we learn how to write function and class templates
- Will have a chance to practice the Big Five in Assignment 2