COMP6771
Advanced C++ Programming

Week 5
Part Two: Dynamic Memory Management

2017

www.cse.unsw.edu.au/~cs6771

Initialisation Revisited

```
#include <iostream>

struct X {
  X() { std::cout << "X() "; }
};

struct Y {
  Y() { std::cout << "Y() "; }
};

class A {
  X x;
public:
  A() { std::cout << "A() "; }
};

class B : public A {
  Y y;
public:
  B() { std::cout << "B() "; }
};

int main() {
  B b;
}
```

What is the output?

a. X() A() Y() B()

b. A() X() B() Y()

c. Y() B() X() A()

d. B() Y() A() X()

e. X() Y() A() B()

Initialisation Revisited

```
#include <iostream>

struct X {
  X() { std::cout << "X() "; }
};

struct Y {
  Y() { std::cout << "Y() "; }
};

class A {
  X x;
public:
  A() { std::cout << "A() "; }
};

class B : public A {
  Y y;
public:
  B() { std::cout << "B() "; }
};

int main() {
  B b;
}
```

What is the output?

a. X() A() Y() B()

b. A() X() B() Y()

c. Y() B() X() A()

d. B() Y() A() X()

e. X() Y() A() B()

Finalisation Revisited

```cpp
#include <iostream>

struct X {
  ~X() { std::cout << "~X() "; }
};

struct Y {
  ~Y() { std::cout << "~Y() "; }
};

class A {
  X x;
public:
  ~A() { std::cout << "~A() "; }
};

class B : public A {
  Y y;
public:
  ~B() { std::cout << "~B() "; }
};

int main() {
  B b;
}
```

What is the output?

Finalisation Revisited

```
#include <iostream>

struct X {
  ~X() { std::cout << "~X() "; }
};

struct Y {
  ~Y() { std::cout << "~Y() "; }
};

class A {
  X x;
public:
  ~A() { std::cout << "~A() "; }
};

class B : public A {
  Y y;
public:
  ~B() { std::cout << "~B() "; }
};

int main() {
  B b;
}
```

What is the output?

Reversed:

~B() ~Y() ~A() ~X()

Lifetimes and Named/Unnamed Objects

- Lifetime: the period of time in which memory is allocated for an object
- Different kinds of objects:
  - Static objects: allocated in a global (static) area
  - Local (or automatic or stack) objects
  - Heap objects
- Named objects: (named by the programmer): their lifetimes determined by their scope
- Heap objects (unnamed objects): their lifetimes determined by the programmer

Local Objects

```
void f() {
  int i = 1;            // automatic
}
```

- Lifetime: created when f is called and cease to exist when this invocation of f completes (with destructors called)
- Primitive objects are not initialised by default
- Class objects are initialised by constructors
- The same name denotes different objects during different invocations of the function f
- Memory management done by the compiler

Local Objects (Cont'd)

```
{
    std::stack<int> s1;
        {
           std::stack<int> s2;
        } ---> call ~stack() for s2
} ---> call ~stack() for s1
```

- Every object inaccessible at its closing '}'s'
- The destructors s1 and s2 are called when s1 and s2 die

```
{
    std::stack<int> s1;
        {
          std::stack<int> &s2 = s1;
        } ---> ???
} ---> call ~stack() for s1
```

- s2 is a reference to s1
- The destructor for s1 is called when s1 dies

Temporary Objects

- Unnamed objects created on the stack by the compiler.
- Read the detailed rules:
  http://www-01.ibm.com/support/knowledgecenter/SSGH3R_8
- Used during reference initialization and during evaluation of expressions including type conversions, argument passing, function returns, and evaluation of the throw expression.
- There are two exceptions in the destruction of full-expressions:
  - The expression appears as an initializer for a declaration defining an object: the temporary object is destroyed when the initialization is complete.
  - A reference is bound to a temporary object: the temporary object is destroyed at the end of the reference's lifetime.

Lifetimes of Temporary Objects

```
#include<vector>

std::vector<int> f() { return std::vector<int>{}; }

void g() {
  const std::vector<int>& v = f();
}
```

The lifetime of the temporary vector created f may persist as long as the reference is live.

Static Objects

- Four categories of objects:
  - Local static objects
  - Global objects
  - Namespace objects
  - Class static objects
- Memory management done by the compiler

These objects are often referred to as global/static objects.

(Local) Static Objects

```
int f() {
  static int i = 0;
  return ++i;
}
cout << f(); // prints 1
cout << f(); // prints 2
```

- lifetime: created when f is first called and persist until the program completes (with destructors called)
- Primitive objects are initialised to default values
- Class objects are initialised by constructors
- (Local) static objects are still visible locally

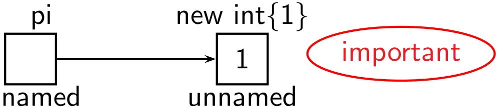
Global, Namespace and Class Static Objects

```
file.cpp:
    std::stack<int> s;             // global
    namespace X {
       std::queue<short> q;        // namespace
    }
    class Y {
       static std::list<char> l; // class static
    }
```

- All created at program start-up and persist until the program completes (with their destructors called)
- Primitive objects initialised to default values
- Class objects are initialised by constructors

Heap (or Free-Store) Objects

- Allocated via new and persist until deleted via delete
- In "int $*$pi $=$ new int$\{1\}$", two objects are involved:



  - pi is destroyed at the end of its lifetime automatically
  - the pointed-to object must be destroyed explicitly

- Thus, important to remember:

  `delete pi` $\equiv$ `delete *pi` $\equiv$ `unnamed`

  `delete pi` $\neq$ `delete named`

  1. First, the destructor for the pointed-to object, *p, is called
  2. Then, the corresponding memory is freed

The destructor for a built-in type does nothing.

Heap Objects (Cont'd)

```
int* f() {
  int *pi = new int{1};
  return pi;
}
int main() {
  int *pj = f();
  delete pj;
}
```

- pi and pj are named local objects
- new int(1) is an unnamed heap object

Match delete with the Corresponding new

```
int *pi = new int{1};
int *pj = new int[10];
            // an array of 10 ints uninitialised
...

delete pi;    // delete [] pi is undefined
delete [] pj; // delete pj is undefined
```

An array object typically contains the information regarding the number of its elements

new: Dynamically Allocate and Initialise Objects

```
int *pi = new int;       // an uninitialised int
int *pj = new int{};     // an int initialised to 0

std::string *ps1 = new std::string;  // initialised to empty
                        // string by calling string's default ctor
std::string *ps2 = new std::string{"abc"}; // initialised to
                                           // string "abc"

auto *pf = new auto{3.14};
        // inferred as float *pf = new float{3.14};

const int *pci = new const int{100};
             // an const int object initialised to 100

std::vector<int> *pv1 = new std::vector<int>{};
             // a vector<int> initialised to be empty
             // if allocation fails, throws std::bad_alloc

std::vector<int> *pv2 = new (nothrow) std::vector<int>{};
             // a vector<int> initialised to be empty
             // if allocation fails, returns nullptr
```

Many Memory-Corruption Bugs in C/C++ Code!

```
int i, *pi1 = &i, *pi2 = nullptr;
double *pd = new double{3.14}, *pd2 = pd;

delete i;    // error: i is not a pointer
delete pi1; // undefined: pi1 points to a stack location
delete pd;  // ok
delete pd2; // double free!
*pd = 1.0;  // accessing a a dangling pointer
delete pi2; // ok: deleting nullptr does nothing

int* foo() {
  int *pi = new std::vector<int>{50};
  return pi;
}

int *pj = foo();
// If the programmer never calls delete pj, then you have
// a memory leak! The memory storing the vector has leaked.
```

Resetting the Value of a Pointer After a `delete`

- A deleted pointer is not automatically nullified
- Set a deleted pointer to nullptr yourself:

```
int *p = new int{1};
delete p;
p = nullptr;
```

- We nullify the pointers in a moved-from object:

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

new: Dynamic Arrays

- Allocate and initialise dynamic arrays

```
int *p = new int[5];    // block of 5 uninitialised ints
int *p = new int[5]{}; // block of 5 ints initialised to 0

// block of 5 empty strings
std::string *p = new std::string[5];
// block of 5 empty strings
std::string *p = new std::string[5]{};

int *p = new int[5](1);             // error
std::string *p = new std::string[5]("abc");    // error

int *p = new int[5]{0, 2, 3, 4, 5};    // ok
std::string *p = new std::string[5]{"an"};
// ok: the first initialised
// to "an" and the last four to empty strings
```

Motivations for Memory Management

- Standard solution (value semantics)

```
SMatrix operator+(const SMatrix &a, const SMatrix &b) {
    SMatrix c = a;
    return c+=b;
} // can be efficient now in C++11 due to move ctor/assignment
```

- Efficient

```
SMatrix& operator+(const SMatrix &a, const  SMatrix &b) {
    SMatrix c = a;
    return c+=b;
} // but incorrect
```

- Efficient

```
SMatrix& operator+(const SMatrix &a, const  SMatrix &b) {
    SMatrix *c = new SMatrix;
    return *c+=b;
} // but who is to free the heap-allocated object?
```

- When objects are "large", heap objects may be used. Using pointers is a burden on users $\Longrightarrow$ memory management techniques (smart pointers, reference counting, handles, etc.) so that users don't have to manage pointers explicitly themselves

Smart Pointers: Motivations

- Memory leak:

```
void f() {
    X* x = new X{};
    ...
    delete x;
}
```

  - if x is not deleted, you suffer the memory leak problem
  - If an exception is thrown in ..., delete x will not be executed. You will then suffer the memory leak problem

- Double free and dangling pointers

One Solution: Bad Programming Style

```
void f() {
   X* x = new X{};
   try {
      ...
   }
   catch (...) {
      delete x;
      throw;       // rethrow the exception
   }

   delete x;
}
```

Smart Pointers: Overloading of ->

- Usual notations:
  - if p is a built-in pointer, `p->x` accesses the member `x`
  - if p is an object or a reference to an object, `p.x` accesses the member `x`
- Smart pointers:
  - a class type exhibits a pointer-like behaviour when the -> is overloaded
  - If p is an object or reference, p->x can be used
  - p is known as a smart pointer
- Item 28, Meyers' More Effective C++

Basic Idea: Defining a Smart Pointer to Strings

```
#include <iostream>

class StringPtr {
public:
  StringPtr(std::string *p) : ptr{p} { }
  ~StringPtr() { delete ptr; }
  std::string* operator->() { return ptr; }
  std::string& operator*() { return *ptr; }
private:
  std::string *ptr;
};

int main() {
  std::string *ps = new std::string{"smart pointer"};
  StringPtr p{ps};
  std::cout << *p << std::endl;
  std::cout << p->size() << std::endl;
}
```

Using (Named) Objects to Prevent Memory Leaks

- Place a pointer inside a named object and have the object manage the pointer for you

```
class StringPtr {
  ~StringPtr() { delete ptr; }
  ...
  std::string *ptr;
};
```

- User code:

```
{
StringPtr p(new std::string("smart pointer"));
} <-- *ptr freed here when ~StringPtr() called
```

- The object p's destructor will delete the heap string object

What About Double Free?

- Yes, it is still with us:

```
class StringPtr {
  ~StringPtr() { delete ptr; }
  ...
  std::string *ptr;
};
```

- User code:

```
{
      StringPtr p(new std::string("smart pointer"));
      {
         StringPtr q(p);
      } <-- *ptr freed here
      std::cout << p->size() // Oops
} <-- *ptr freed again
```

Reference Counting (RC)

- A classic automatic garbage collection technique
- Can be used manually by the C++ programmer to prevent
  - memory leaks, and
  - the dangling pointer problem
- FAQs 31.10 – 31.11: RC with copy-on-write semantics
- Item 29, Meyers' More Effective C++

A Simple RC Scheme (without COW)

```
class PointPtr;

class Point {
public:
  friend class PointPtr;
  static PointPtr create();
  static PointPtr create(int i, int j);
private:
  Point() : count_(0) { i = j = 0; }
  Point(int i, int j) : count_(0) {
    this->i = i; this->j = j;
  }
  unsigned count_;
  int i, j;
};
```

- Users cannot construct point objects directly
- Factory methods used to construct point objects

A Simple RC Scheme (Cont'd)

```
class PointPtr { // smart pointer
public:
  Point* operator-> () { return p_; }
  Point& operator* () { return *p_; }
  PointPtr(Point* p) : p_{p} { ++p_->count_; } // p cannot be NULL
  ~PointPtr() { if (--p_->count_ == 0) delete p_; }
  PointPtr(const PointPtr& p) : p_{p.p_} { ++p_->count_; }
  PointPtr& operator= (const PointPtr& p) {
    ++p.p_->count_;
    if (--p_->count_ == 0) delete p_;
    p_ = p.p_;
    return *this;
  }
private:
  Point* p_; // p_ is never NULL
};
inline PointPtr Point::create() { return new Point(); }
inline PointPtr Point::create(int i, int j) {
  return new Point(i,j);
}
```

A Simple RC Scheme (Cont'd)

- Client code:

```
int main() {
  PointPtr p1 = Point::create(1, 1);
  PointPtr p2 = p1;
  p1 = Point::create(2, 2);
  p2 = p1;
}
```

- In the two overloaded Point::create, the constructor PointPtr (Point *p) is called on return

Does new Do More Than Allocate Memory (FAQ12.01)?

- Consider:

```
X *p = new X();
```

- The compiler-generated code looks something like:

```
X *p = (X*) operator new( sizeof(X) ); //Step 1
try {
 new (p) X(); // Step 2
} catch (...) {
 operator delete(p);
 throw;
}
```

- No memory leak occurs if the constructor throws an exception

Does delete Do More Than Deallocate Memory (FAQ12.01)?

- Consider:

```
delete p // p is an object of class X
```

- The compiler-generated code looks something like:

```
p->~X(); //Step 1: calling the destructor
operator delete(p); //Step 2: freeing the memory
```

- Deleting a null pointer is safe

```
if (p != 0) delete p; // test redundant
// the test done in operator delete(void *p)
```

A Variety of Smarter Pointers!

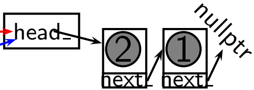
- Smart pointers: Strategies for handling p = q, where p and q are smarter pointers
- "New" smart pointers introduced in C++11:
  - unique_ptr
  - shared_ptr
  - weak_ptr
- Why smarter pointers?
  - Less bugs (e.g., fewer double-free & dangling pointer problems)
  - Exception safety (e.g., automatic memory deallocation)
  - Gabagge collection (RC)
  - Efficiency (copy-on-write)

unique_ptr (§12.1.5)

- A unique_ptr owns the object to which it points to
- The underlying object always owned by one owner only
- When the unique_ptr goes out of scope, its internal object pointed to by its internal pointer is destroyed

```
{
  // C++14: auto p = std::make_unique<UB_stack>();
  std::unique_ptr<UB_stack> p(new UB_stack{});
  p.get()->push(1);
  p.get()->push(2);
} ~unique_ptr<UB_stack> { delete ptr; }
```

ptr



- Explicit ownership transfer:

```
std::unique_ptr<UB_stack> p(new UB_stack{});
std::unique_ptr<UB_stack> q1 = p;   // error: no copy
std::unique_ptr<UB_stack> q2;
q2 = p;                             // error: no assignment
std::unique_ptr<UB_stack> q3;
q3.reset(p.release());              // ok: ownership transfer
```

Passing and Returning `unique_ptr`

```
std::unique_ptr<UB_stack> foo() {
  std::unique_ptr<UB_stack> p(new UB_stack{});
  p.get().push(1);
  p.get().push(2);
  return p;
}

int main() {
  std::unique_ptr<UB_stack> q = foo();
}
```

- Can assign/copy a temporary `unique_ptr` that is about to die
- The compiler makes sure that q will be the unique owner after the call to `foo` returned; the temporary has lost the ownership

Smarter Pointers and Exception

- Smarter pointers:

```
void f() {
        std::unique_ptr<UB_stack> up(new UB_stack{});;

        code that throws an exception

} // the destructor for up called here
  // the underlying stack object is freed
```

- Dumb pointers:

```
void f() {
        UB_stack *p = new UB_stack();

        code that throws an exception

        delete p;
} // the destructor for *p not called here
  // the underlying stack object has leaked
```

The destructors for all named objects are called only!

unique_ptr

```
std::unique_ptr<int> x = new int{i};
std::unique_ptr<int> y = x; // Compile error
std::unique_ptr<int> z = std::move(x); // Transfers ownership
                                        // z now owns the memory
                                        // x is rendered invalid
z.reset(); // Deletes the memory
```

- copy constructor and assignment disabled
- move used to transfer ownership
- Create more than one owner (Don't do this!):

```
{
  int *pi = new int{1};
  std::unique_ptr<int> p ( pi );
  std::unique_ptr<int> q ( pi );
}      --> new int(1) will be deleted twice
```

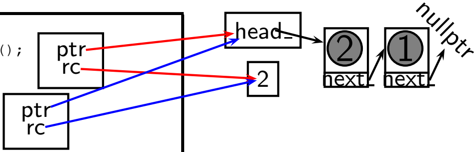
Two unique_ptr objects should not hold ownership to the same heap object. The double-free problem!

shared_ptr (§12.1.1)

- Allows multiple pointers to point to the same object (with the number of pointers reference-counted)
- The pointed-to object is destroyed when the last pointer goes out of scope

```
{
  auto p = std::make_shared<UB_stack>();
  p.get()->push(1);
  p.get()->push(2);
  {
    std::shared_ptr<UB_stack> q = p;
  }
  // ~shared_ptr() called here: ref_count (rc) = 1

} // ~shared_ptr() called here: ref_count (rc) = 0
  // the stack and counter destroyed
```



- The destructor:

```
~shared_ptr::shared_ptr() {
  if (--*ref_count == 0) { delete ptr; delete ref_count; }
}
```

shared_ptr

```
std::shared_ptr<int> x = std::make_shared<int>(1);
std::shared_ptr<int> y = x; // Both now own the memory

x.reset(); // Memory still exists, due to y.
y.reset(); // Deletes the memory, since
           // no one else owns the memory
```

- Referece counted ownership: every copy of the same shared_ptr owns the same pointed-to object
- Freed only when all instances are destroyed
- What about cycles?
  (e.g., A has shared_ptr to B, B has shared_ptr to A)

weak_ptr

```
std::shared_ptr<int> x = std::make_shared<int>(1);
std::weak_ptr<int> wp = x; // x owns the memory
{
  std::shared_ptr<int> y = wp.lock(); // x and y own the memory
  if (y)
    { // Do something with y }
} // y is destroyed. Memory is owned by x

 x.reset(); // Memory is deleted

 std::shared_ptr<int> z = wp.lock(); // Memory gone; get null_ptr
 if (z)
   { // will not execute this }
```

- Useful when you need to access to a resource through a pointer that can outlive the resource
- Read: http://www.drdobbs.com/weak-pointers/184402026

weak_ptr + shared_ptr (§12.1.6)

- Points to an object managed by a shared_ptr
- Doesn't get reference-counted

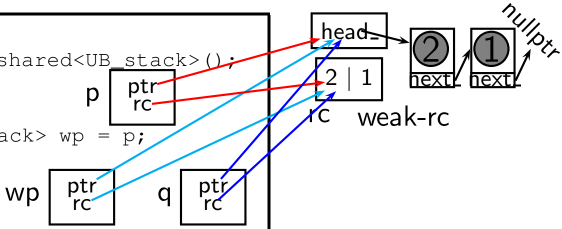
```
{
  auto p = std::make_shared<UB_stack>();
  p.get()->push(1);
  p.get()->push(2);
  std::weak_ptr<UB_stack> wp = p;

  {

    std::shared_ptr<UB_stack> q = p;

    // The snapshot at this point
  } // ~shared_ptr(): rc = 1

  p.reset();    // ~shared_ptr(): rc = 0
                // the stack destroyed
} // ~weak_ptr(): weak-rc = 0
```



- Conceptually illustrated but may vary across implementations

Smart Pointers for Arrays

- Different operations provided for arrays (§12.2.1)

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

int main() {
  auto p =std::unique_ptr<UB_stack[]>(5);
  p[0].push(1);
  p[1].push(2);
} // <-- delete[] called on the underlying array object
```

- shared_ptr does not provide support for arrays. You can supply your own deleter (Page 480) - Do not do this!:

```
struct deleter {
  void operator ()( UB_stack const * p) { delete[] p; }
};

int main() {
  std::shared_ptr<UB_stack> p (new UB_stack[5](), deleter());
}
```

Reading

- Section 18.1
- C++ Memory and Resource Management
  http://www.informit.com/articles/article.asp?p=30642
- C++ FAQ Lite: http://yosefk.com/c++fqa/exceptions.html
- How is exception handling implemented?
  http://www.codeproject.com/cpp/exceptionhandler.asp