COMP6771
Advanced C++ Programming

Week 3
Part Two: Classes - Constructors (cont.), Uniform Initialisation, Friends, Type Conversions, Scope and Copy Control

2017

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

Assignment Submission

- Due: Tuesday Night (15 Aug 2017)
- Late Penalty: 20%/day off your mark

  A day is defined as a 24-hour day and includes weekends and holidays.

  No submissions will be accepted more than three days after the due date.
- Course staff do not have to reply to emails on Weekends!
- Submit by logging to a CSE computer:

```
give cs6771 ass1 calculator.cpp
```

- It is important that you check that you have successfully submitted before the deadline:

```
Submission datestamp:        Tue Aug 10 13:59:59 2017
Assignment deadline:        Tue Aug 15 23:59:59 2017
This submission is 5 days and 10 hours early
Submission accepted
```

Checking your submission

```
6771 classrun -check calculator
```

```
Session: 17s2

Current day and time:        Thu Aug 10 13:59:59 2017
Assignment deadline:        Tue Aug 15 23:59:59 2017
A submission now would be 5 days and 10 hours early
Event           Day and time              Details
=====================================================
Submission      Thu Aug 10 13:59:59 2017 3344556 orphans calculator [5 days +10 hours early]


Most recent submission:
-rw-r--r-- neda/neda 4740 2017-08-10 13:59 calculator.cpp

Your submission has been marked. Do you want to examine it (y/n)?
```

Compile Errors

- You must check that your code compiles on school computers before submission.
- Your code is checked during the submission.
- Your code is tested against the sample input/output during submission.
- If your code does not compile and you don't have time to fix your it, you should continue with the submission.

Constructor Initialiser List!

```
class A {
A() :          { }
}
```

Constructor Phases

The initialisation phase occurs *before* the body of the constructor is executed, regardless of whether the initialiser list is supplied.

A constructor will:

1. initialise all data members: each data member is initialised using the constructor initialiser or by default value (using the same rules as those used to initialise variables); then
2. execute the body of constructor: the code may assign values to the data members to override the initial values

Delegating Constructors (C++11)

A constructor can call another constructor

```
Sales_data(const std::string &s, unsigned n, double p):
             bookNo{s}, units_sold{n}, revenue{p*n} { }

Sales_data() : Sales_data{"", 0, 0} { }

// Sales_data(const std::string &s): bookName{s} { }
// replace with:
Sales_data(const std::string &s) : Sales_data{s, 0, 0} { }

// tell this constructor to call the default constructor first
Sales_data(std::istream &is) : Sales_data{} {
  read(is, *this);
}
```

In the last constructor, the data members are initialised first before they are possibly re-initialised again inside.

Most Vexing Parse

Consider:

```
struct Timer {
  Timer() {}
};

struct TimeKeeper {
  TimeKeeper(const Timer& t) {}
  void printTest() { std::cout << "In TimeKeeper" << std::endl; }
};

int main() {
  TimeKeeper time_keeper(Timer());
  time_keeper.printTest();
}
```

What is time_keeper?

- a TimeKeeper object?
- a function prototype that returns a TimeKeeper object?

Most Vexing Parse

Answer: a function prototype that returns a TimeKeeper object
The code then doesn't compile because of the member function call:
How do we fix this?

Most Vexing Parse 2

Consider:

```
void f(double aDouble) {
  int i(int(aDouble));
  std::cout << i << std::endl;
}
```

What is i?

- an int containing aDouble cast/constructed to an int?
- a function prototype that takes an int and returns a int?

Most Vexing Parse 2

Answer: a function prototype that takes an int and returns a int
The code then doesn't compile because of the print (with -Wall):

```
uniformInit.cpp: In function 'void f(double)':
uniformInit.cpp:5:16: error: the address of 'int i(int)' will
always evaluate as 'true' [-Werror=address]
   std::cout << i << std::endl;
                ^
cc1plus: all warnings being treated as errors
```

How do we fix this?

Initialisation

From week 2:

```
int main() {
  int i1{1};
  int i2 = {1};
  int i3 = 1;
  int i4(1);

  int j1{};         // the default value used
  int j2 = int{};   // a temporary containing the default value
}
```

Lots of different ways to initialise objects! Some due to backwards compatibility with C.

Uniform Initialisation

- Problem: the C++ compiler confuses type construction, function calls, and function declarations.
- "if it can possibly be interpreted as a function prototype, then it will be"
- Solution: Rather than using () to construct objects, use {} as they can't be confused for function calls.
- Change: `TimeKeeper time_keeper(Timer());` to: `TimeKeeper time_keeper{Timer{}};`
- This cannot be confused with a function prototype
- See: http://programmers.stackexchange.com/questions/133688/i
- And: https://mbevin.wordpress.com/2012/11/16/uniform-initial

Narrowing

What about our casting problem?

```
int i(int(aDouble));
```

Is this a solution?

```
int i{int{aDouble}};
```

Narrowing

What about our casting problem?

```
int i(int(aDouble));
```

Is this a solution?

```
int i{int{aDouble}};
```

```
uniformInit.cpp: In function 'void f(double)':
uniformInit.cpp:4:20: error: narrowing conversion of 'aDouble'
from 'double' to 'int' inside { } [-Werror=narrowing]
   int i{int{aDouble}};
                    ^
cc1plus: all warnings being treated as errors
```

Narrowing and Casting

- Using C-style initialisation we can cast and lose precision without any warnings. e.g.,

```
    int pi = 3.14;
    int pi = int(3.14);
```

- Using Uniform Initialisation this will always warn and if using -Wall -Werror won't compile.

```
    int pi = {3.14}
```

- To correctly "narrow" these values we need to use a static cast.

```
    int i(static_cast<int>(aDouble));
    int pi = static_cast<int>(3.14);
```

Friends (of a Class)

- Friend declarations can be anywhere in the class
- Friends are not members $\Longrightarrow$ not affected by access control
- A friend of a class can access its private members
- Friends are part of the class's interface

A class can allow another class or function to access its nonpublic members by making that class or function a friend. A class makes a function its friend by including a declaration for that function preceded by the keyword **friend**:

The Interface: Sales_data.h

```
#include <string>
#include <iostream>

class Sales_data {
friend std::ostream& print(std::ostream&, const Sales_data&);
friend std::istream& read(std::istream&, Sales_data&);
public:
  // constructors
  // operations on Sales_data objects
private:
  std::string bookNo;
  unsigned units_sold{0}; // in-class initialiser
  double revenue{0.0};    // in-class initialiser
};

// nonmember Sales_data interface functions
std::ostream &print(std::ostream&, const Sales_data&);
std::istream &read(std::istream&, Sales_data&);
Sales_data add(const Sales_data&, const Sales_data&);
```

Three Types of Friends

- An ordinary function (e.g., add shown before)

```
// friendship can be stated without seeing
// print's declaration
class Sales_data {
  friend std::ostream& print(std::ostream&,
    const Sales_data&);
}
```

- A class

```
// friendship can be stated without seeing
// Window_Mgr's declaration
class Screen {
  friend class Window_Mgr;
};
```

- A member function of a class (infrequently used)

```
// friendship can only be stated if the
// declaration of Window_Mgr is available
class Screen2 {
  friend Window_Mgr& Window_Mgr::relocate(Screen&);
};
```

Friends: Friendship vs. Type Declarations

- A friend declaration only specifies access or friendship

```
// X.h
class X {
  friend class Y;
  friend void f() { return; }
};
```

```
// user.cpp
#include "x.h"

void g() { return f(); }

Y *ymem;
```

```
error: 'f' was not declared in this scope
user.cpp:6:3: error: 'Y' was not declared in this scope
```

- The general declarations are still required:

```
// X.h
class X {
  friend class Y;
  friend void f() { return; }
};

class Y;
void f();
```

```
// user.cpp
#include "X.h"

void g() { return f(); }

Y *ymem;
```

Overloading Member Functions (Read §7.3)

```
#include <string>
#include <iostream>

class Screen {
public:
  // constructors omitted
  Screen &move(pos r, pos c);
  // Overloaded based on parameter types
  Screen &set(char);
  Screen &set(pos, pos, char);

  // Overloaded based on const
  Screen &display(std::ostream &os)
                  { do_display(os); return *this; }
  const Screen &display(std::ostream &os) const
                  { do_display(os); return *this; }
private:
    void do_display(std::ostream &os) const { os << contents;}
    pos cursor {0};
    pos height {0}, width{0};
    ...
};
```

Static Data Members

Classes sometimes need members that are associated with the class, rather than with individual objects of the class type.

Example:

A bank account class might need a data member to represent the current prime interest rate. In this case, we'd want to associate the rate with the class, not with each individual object. From an efficiency standpoint:

- No reason for each object to store the rate.
- If the rate changes, we'd want each object to use the new value.

Static Data Members

- Change the interface in `Sales_data.h`:

```
class Sales_data {
public:
  Sales_data(const string &s): bookNo{s} { ++counter; }
private:
  std::string bookNo;
  unsigned units_sold{0}; // in-class initialiser
  double revenue{0.0};    // in-class initialiser
  // Count how many objs are created by this constructor
  static int counter;
};
```

- Add the definition in `Sales_data.cpp`:

```
int Sales_data::counter = 0;
```

- User code:

```
std::cout << Sales_data::counter << std::endl; // 0
Sales_data d1{"123"};
Sales_data d2{"456"};
std::cout << d2.counter << std::endl; // 2
```

- Objects contain only non-static data members

Implicit Type Conversions

Constructors can create implicit conversions from other types to the class type. This makes sense since constructors build new objects of the class type based on arguments of different types.

NB

A constructor that can be called with a *single* argument defines an implicit conversion.

```
class Foo {
public:
  Foo(const std::vector<int> v) : data{v} {}
private:
  std::vector<int> data;
};

std::vector<int> ivec;
Foo foo = ivec;  // create a Foo object by assigning a
                 // std::vector to it
```

Suppressing Implicit Conversions

Sometimes it may be dangerous to allow certain implicit type conversions.
We can qualify the constructor with explicit, which will stop it from being used in implicit conversions.

```
#include <string>
#include <iostream>

class Sales_data {
public:
  Sales_data(const std::string &s, unsigned n, double p):
             bookNo{s}, units_sold{n}, revenue{p*n} { }
  explicit Sales_data(const std::string &s): bookNo{s} { }
  explicit Sales_data(std::istream &);
  ...
```

- The explicit keyword is meaningful only on constructors that can be called with a single argument.
- The explicit keyword is used only on the constructor declaration inside the class.

(Explicit) Type Conversions

- A constructor must be called explicitly:

```
std::string null_book = "123";

Sales_data data1{"123", 1, 20.0};

data1.combine(null_book);                // error

data1.combine(Sales_data{null_book}); // ok
```

- Same here:

```
Sales_data data1{"123", 1, 20.0};
data1.combine(std::cin);                                    // error

data1.combine(Sales_data{std::cin});                        // ok

data1.combine(static_cast<Sales_data>(std::cin));  // ok
  // The compiler examines the Sales_data class to find
  // an appropriate constructor for the conversion

```

vector: the Constructor That Takes a Size is Explicit

```
#include<iostream>
#include<vector>

int main() {
  std::vector<int> v(10);  // ok - note: needs to be () not {}
  std::vector<int> v = 10; // error: explicit

  void f(std::vector<int>);  // declare function f
  f(10);                      // error: explicit
  f(std::vector<int>(10));   // ok
}
```

Name Resolution in Class Scope (§7.4.1)

Two steps used in finding the declaration of a name used for the type of a parameter/return of a member function in a class:

1. Considering the declarations seen earlier in the class
2. Considering the declarations in the class's enclosing scope(s)

Three steps for a name inside the definition of a member function, *foo*, given both inside or outside a class:

1. Considering the declarations preceding it inside *foo*
2. Consider all the declarations in the class
3. Considering the declarations in the *foo*'s enclosing scope(s)

```
typedef double Money;
std::string bal;
class Account {
public:
  Money balance() { return bal; }
private:
  Money bal;
}
```

`Money` is an alias for `double` and `bal` is the data member.

Name Resolution

```
void foo1() {}
typedef int value_type;
class Foo {
  value_type foo1(); // OK: foo1 redefined in different scope
  typedef double value_type; // error: cannot redefine type
};
```

Once a name has been used as a type it may not be redefined.

```
void foo1() {}
typedef int value_type;
class Foo {
  // swap the previous lines around, typedef not yet used
  typedef double value_type; // OK: value_type redefined
  value_type foo1(); // OK: foo1 redefined
};
```

Name Resolution

Names used in class member function definitions are resolved by:

1. considering declarations in the local scope of the member
2. considering declarations in class scope
3. considering declarations that appear before the member function in the outer scope

```
int x = 1;
struct Foo {
  int x;
  Foo() : x{2} { }
  void print(const int& x) { std::cout << x; }
};

Foo f;
f.print(3); // what is printed?
```

How is the Name g Resolved in print?

```
#include<iostream>

class Foo {
public:
  Foo() : x(2) { }
  void print(int x);
private:
  int x;
};

int g() { return 3; }
void Foo::print(int x) { std::cout << g() << std::endl; }

int main() {
  Foo f;
  f.print(g());
}
```

How is the Name `g` Resolved in `print`?

```
#include<iostream>

class Foo {
public:
  static int g() { return 4; }
  Foo() : x(2) { }
  void print(int x);
private:
  int x;
};

int g() { return 3; }
void Foo::print(int x) { std::cout << g() << std::endl; }

int main() {
  Foo f;
  f.print(g());
}
```

How is the Name `g` Resolved in `print`?

```
#include<iostream>

class Foo {
public:
  static int g() { return 4; }
  Foo() : x(2) { }
  void print(int x);
private:
  int x;
};

int g() { return 3; }
void Foo::print(int x) { std::cout << ::g() << std::endl; }

int main() {
  Foo f;
  f.print(g());
}
```

Object-Based Programming: Copy Control

Classes can control what happens when objects of the class type are copied, assigned, moved, or destroyed. Classes control these actions through special member functions:

1. Copy Constructor
2. Copy Assignment
3. Destructors
4. Move Constructor
5. Move Assignment

The Big Five:

- C++: 1 – 3 (the Big Three)
- C++11: Added 4 and 5 (the Big Five now)

The Big Three/Five

Through the use of constructors, C++ allows the programmer to specify exactly how class objects are created. Creation is clearly a key event in the lifetime of an object, in addition, C++ allows the programmer to define how objects are to be copied, moved, assigned and destroyed. Together these are known as copy control.

Copy and Move Constructors: define what happens when an object is initialised from another of the same type

Copy- and Move-Assignment Operators: define what happens when an existing object is assigned to from, i.e., overwritten with another object of the same type

Destructors: define what happens when an object ceases to exist

Destructor

A member function of the form:

```
class A {
  ~A() { ... }
  ...
}
```

- Unique (as it cannot be overload due to lack of parameters)
- When the destructor is called, two things happen:
  1. The function body of the destructor is executed
  2. The members are destroyed by calling their destructors in their reverse declaration order

> The destructors for built-in types do nothing.

- This is the process of object construction in reverse
- Members of STL container types or smarter pointers are automatically destroyed (by their destructors)
- Used to free memory allocated via new or some other resources, e.g., files and sockets opened in a constructor.

The Compiler-Generated Destructor

- Synthesised for a class if a class doesn't have one:

```
Class A {
private:
  X1 d1;
  X2 d2;
   ...
  Xn dn;
}
```

```
// The synthesised destructor
~A() noexcept { }
```

After the body of ~A() has been executed, the members d1, d2, ... dn are destroyed by calling their destructors in reverse declaration order.

- The synthesised destructor for `Sales_data`:

```
~Sales_data() noexcept { }
```

No need to provide a destructor for a class if its data members are of non-pointer-related built-in types or STL container types

When Is a Destructor Called? (Page 502)

- Variables are destroyed when they go out of scope
- Members of an object are destroyed when the object of which they are a part is destroyed
- Elements in a STL container are destroyed – whether a library container or an array – are destroyed when the container is destroyed
- Dynamically allocated objects are destroyed when the delete operator is applied to a pointer to the object
- Temporary objects are destroyed at the end of full expression in which the temporary was created

In summary, the destructor for a variable is called when it goes out of scope. But a pointer has two associated objects, remember?

Copy Constructor

- A constructor is the copy constructor if its first parameter is a reference to the class type and any additional parameters have default values:

```
class A {
Public:
  A ();                              // default constructor
  A(const A &a) { ... }              // copy constructor
  ...
}
```

- The first parameter must be a reference type. That parameter is almost always a reference to **const**
- The copy constructor is used implicitly in several circumstances. Hence, the copy constructor usually should not be explicit (§ 7.5.4, p. 296).

The Compiler-Generated Copy Constructor

- Synthesised for a class if a class doesn't have one:

```
Class A {
private:
  X1 d1;
  X2 d2;
   ...
  Xn dn;
}
```

```
// The synthesised constructor
A(const A& a) :
  d1(a.d1),
  d2(a.d2),
  ...
  dn(a.dn)
  { }
```

- Memberwise copy in declaration order.
  - Call a member's copy constructor to copy
  - The members of built-in types are copied directly
  - Array members are copied by copying each element
- The synthesised copy constructor for `Sales_data`:

```
Sales_data::Sales_data(const Sales_data &d) :
   bookNo{d.bookNo},
   units_sold{d.units_sold},
   revenue{d.revenue}
   {  }
```

Direct vs. Copy Initialisation (Pages 497 – 498)

```
#include <iostream>

class MyString {
public:
  MyString(const char* s) : str{s} {}
  MyString(const MyString &rhs) : str{rhs.str} {}
private:
  std::string str;
};

int main() {
  MyString s1{"abc"};   // direct initialisation
  MyString s2 = "abc";  // copy initialisation
}
```

- Direct: ordinary function matching to select the best constructor
- Copy: convert the RHS if necessary to an object, copy the RHS to the LHS

Constraints on Copy Initialisation

Remember that if a constructor is declared as explicit we cannot copy initialise

```
#include<iostream>
#include<vector>

int main() {
  std::vector<int> v(10);       // ok
  std::vector<int> v = 10;      // error: explicit

  void f(std::vector<int>);
  f(10);                      // error: explicit
  f(std::vector<int>(10));      // ok
}
```

Initialisation vs. Assignment

- The variable `two` is initialised to one because it is created as a copy of `one`:

```
MyClass one;
MyClass two = one;
```

- However, if we rewrite the code, then two is assigned the value of one.

```
MyClass one, two;
two = one;
```

Remember

When a variable is created to hold a specified value, it is being initialised, whereas when an existing variable is set to hold a new value, it is being assigned.

Copy Assignment Operator

```
class A {
  A& operator=(const A &a) { ... }
  ...
}
```

- Takes only one argument of the same type
- Returns a reference to its left-hand operand, so that a class type behaves similar as a built-in type:

```
A a, b, c;                int i, j, k;
a = b = c;                i = j = k;
```

- The syntax:

```
Sales_data x1, x2;
x1 = x2; // SAME AS x1.operator=(x2);
```

Will look at operator overloading in Week 4.

The Compiler-Generated Copy Assignment

- Synthesised for a class if a class doesn't have one:

```
Class A {
private:
  X1 d1;
  X2 d2;
   ...
  Xn dn;
}
```

```
// The synthesised operator=
A &operator=(const A &rhs) {
  d1 = rhs.d1;
  d2 = rhs.d2;
  ...
  dn = rhs.dn;
}
```

- Memberwise assignment in declaration order.
  - Call a member's assignment operator to assign
  - The members of built-in types are assigned directly
  - Array members are assigned by assigning each element

Preventing Copies (§13.1.5 – 13.1.6)

```
class NoCopy {
public:
  NoCopy() = default;
  NoCopy(const NoCopy &data) = delete;
  NoCopy &operator=(const NoCopy &data) = delete;
  ~Sale_data = default;
```

- The = delete tells the compiler not to generate these members automatically (as we don't want them).
- I/O types don't allow copying or assignment:

```
// error: cannot copy
std::ostream print(std::ostream os, const Sales_data&);
print(std::cout, data);

// ok: pass by a reference
std::ostream& print(std::ostream &os, const Sales_data&);
print(std::cout, data);
```

- If a class has a deleted destructor, objects can only be created via *new*.

To be continued...

Next week:

- Copy control continued
- Move semantics
- Operator Overloading