

COMP6771

Advanced C++ Programming

Week 3

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

2016

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

Assignment 1: Fun With Collections (Word Ladder)

- Use some STL containers and algorithms only
- The reference solutions are crude
- Performance hurdles (no more than 10 times slower)

Assignment Submission

- **Due:** Sunday night
- **Late Penalty:** 20%/day off **your** mark
- Course staff do not have to reply to emails at Weekends
- Submit by logging to a CSE computer:

```
1 6771 give cs6771 wl your-files
```

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

```
1 Submission timestamp: Mon Jul 25 09:35:05 2016
2 Assignment deadline: Sat Aug 13 23:59:59 2016
3 This submission is 19 days and 14 hours early
4 Submission accepted
```

Checking your submission

```
6771 classrun -check wl
```

```

1 Session: 16s2
2
3 Current day and time: Mon Jul 25 09:40:29 2016
4 Assignment deadline: Sat Aug 13 23:59:59 2016
5 A submission now would be 19 days and 14 hours early
6 Event          Day and time          Details
7 =====
8 Submission      Sat Jul 23 10:54:14 2016  3344556 orphans wl [21 days +13 hours early]
9 Submission      Mon Jul 25 09:35:05 2016  3344556 orphans wl [19 days +14 hours early]
10
11 Most recent submission:
12 -rw-r--r-- jingling/jingling 4740 2016-07-25 08:58 WordLadder.cpp
13
14 Your submission has been marked. Do you want to examine it (y/n)?

```

Compile Errors

You must check that your code compiles on the school computers before submission.

Your code is checked during the submission as well, if your code does not compile:

```
1 Compiling your submission...
2 g++ -std=c++14 -g -Wall -Werror -O2 -MM WordLadder.cpp genlib.cpp lex
3 g++ -std=c++14 -g -Wall -Werror -O2 -c -o WordLadder.o WordLadder.c
4 WordLadder.cpp: In function 'int main(':
5 WordLadder.cpp:146:4: error: 'deletedic' was not declared in this scope
6 deletedic;
7     ^
8 make: *** [WordLadder.o] Error 1
9 Compilation errors and/or warnings?
10 You should fix these before proceeding with the submission!
11 Submission test failed. Continue with submission (y/n)? )<+>
```

If you do not have time, or cannot fix your code, you should continue with the submission.

Dry Run Errors

Your code is tested against the sample input/output during submission. If you fail the dryrun:

```
1 You have failed this test. You can submit now if you wish,  
2 but you should fix your program in order to improve your  
3 performance mark.  
4  
5 Submission test failed. Continue with submission (y/n)?
```

If you do not have time, or cannot fix your code, you should continue with the submission.

Constructor Initialiser List!

```
class A {  
A() : [REDACTED] { }  
}
```

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

```
1 Sales_data(const std::string &s, unsigned n, double p):  
2     bookNo{s}, units_sold{n}, revenue{p*n} { }  
3  
4 Sales_data() : Sales_data{"", 0, 0} { }  
5  
6 // Sales_data(const std::string &s): bookName{s} { }  
7 // replace with:  
8 Sales_data(const std::string &s) : Sales_data{s, 0, 0} { }  
9  
10 // tell this constructor to call the default constructor first  
11 Sales_data(std::istream &is) : Sales_data{} {  
12     read(is, *this);  
13 }
```

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

Most Vexing Parse

Consider:

```
1 struct Timer {  
2     Timer() {}  
3 };  
4  
5 struct TimeKeeper {  
6     TimeKeeper(const Timer& t) {}  
7     void printTest() { std::cout << "In TimeKeeper" << std::endl; }  
8 };  
9  
10 int main() {  
11     TimeKeeper time_keeper(Timer());  
12     time_keeper.printTest();  
13 }
```

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:

```
1 timeKeeper.cpp: In function int main():  
2 timeKeeper.cpp:14:15: error: request for member printTest in  
3 time_keeper, which is of non-class type TimeKeeper(Timer (*))()
```

How do we fix this?

Most Vexing Parse 2

Consider:

```
1 void f(double aDouble) {  
2     int i(int(aDouble));  
3     std::cout << i << std::endl;  
4 }
```

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):

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

How do we fix this?

Initialisation

From week 2:

```
1 int main() {  
2     int i1{1};  
3     int i2 = {1};  
4     int i3 = 1;  
5     int i4(1);  
6  
7     int j1{};           // the default value used  
8     int j2 = int{};     // a temporary containing the default value  
9 }
```

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/is-c11-uniform-initialization-a-replacement-for-the-old>
- And: <https://mbevin.wordpress.com/2012/11/16/uniform-initialization/>

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}};
```

```
1 uniformInit.cpp: In function void f(double):
2 uniformInit.cpp:4:20: error: narrowing conversion of aDouble
3 from double to int inside { } [-Werror=narrowing]
4     int i{int{aDouble}};
5             ^
6 cclplus: all warnings being treated as errors
```


Narrowing and Casting

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

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

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

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

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

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

Friends (of a Class)

- Friend declarations can be anywhere in the class
- Friends are not members \implies **not affected by access control**
- A friend of a class can access its private members
- **Friends are part of the class's interface**
- Friendship is not transitive, reciprocal or inherited
- The scenarios where friends are often used:
 - Operator overloading for I/O

```
1 Sales_data dl
2
3 std::cout << dl;           // we want this
4 dl << std::cout;          // but not this
```

- Define nonmember functions that don't modify the states of their arguments, as illustrated by:
`std::ostream &print(std::ostream&, const Sales_data&);`
- Grant access to some related classes

Operator overloading will be dealt with in Assignment 2.

The Interface: Sales_data.h

```
1 #include <string>
2 #include <iostream>
3
4 class Sales_data {
5     friend std::ostream& print(std::ostream&, const Sales_data&);
6     friend std::istream& read(std::istream&, Sales_data&);
7 public:
8     // constructors
9     // operations on Sales_data objects
10 private:
11     std::string bookNo;
12     unsigned units_sold{0}; // in-class initialiser
13     double revenue{0.0};    // in-class initialiser
14 };
15
16 // nonmember Sales_data interface functions
17 std::ostream& print(std::ostream&, const Sales_data&);
18 std::istream& read(std::istream&, Sales_data&);
19 Sales_data add(const Sales_data&, const Sales_data&);
```

Three Types of Friends

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

```
1 // friendship can be stated without seeing
2 // print's declaration
3 class Sales_data {
4     friend std::ostream& print(std::ostream&,
5                               const Sales_data&);
6 }
```

- A class

```
1 // friendship can be stated without seeing
2 // Window_Mgr's declaration
3 class Screen {
4     friend class Window_Mgr;
5 };
```

- A member function of a class (infrequently used)

```
1 // friendship can only be stated if the
2 // declaration of Window_Mgr is available
3 class Screen2 {
4     friend Window_Mgr& Window_Mgr::relocate(Screen&);
5 };
```

Friends: Friendship vs. Type Declarations

- A friend declaration only specifies access or friendship

```

1 // X.h
2 class X {
3     friend class Y;
4     friend void f() { return; }
5 };

```

```

1 // user.cpp
2 #include "x.h"
3
4 void g() { return f(); }
5
6 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:

```

1 // X.h
2 class X {
3     friend class Y;
4     friend void f() { return; }
5 };
6
7 class Y;
8 void f();

```

```

1 // user.cpp
2 #include "X.h"
3
4 void g() { return f(); }
5
6 Y *ymem;

```

Nonmember Non-Friend Functions

- Cannot access the private members of a class
- Packaged as part of the same interface as its related class
- Our example:

```
add(const Sales_data &d1, const Sales_data &d2)
```

- User code:

```
Sales_data d1{"123", 2, 40.0}, d2{"123", 1, 22.0};  
Sales_data d3 = add(d1, d2);
```

Don't want d1 and d2 to be modified.

Overloading Member Functions (Read §7.3)

```
1 #include <string>
2 #include <iostream>
3
4 class Screen {
5 public:
6     // constructors omitted
7     Screen &move(pos r, pos c);
8     // Overloaded based on parameter types
9     Screen &set(char);
10    Screen &set(pos, pos, char);
11
12    // Overloaded based on const
13    Screen &display(std::ostream &os)
14        { do_display(os); return *this; }
15    const Screen &display(std::ostream &os) const
16        { do_display(os); return *this; }
17 private:
18     void do_display(std::ostream &os) const { os << contents;}
19     pos cursor {0};
20     pos height {0}, width{0};
21     ...
22 };
```

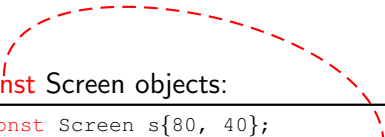
User Code

- non-const Screen objects:

```
1 Screen s{80, 40};  
2 s.display(std::cout); // call the non-const version  
3 s.move(4,5).set(#); // Move the cursor to (4,5)  
4                       // and set it to #
```

- **const** Screen objects:

```
1 const Screen s{80, 40};  
2 s.display(cout); // call the const version  
3 s.move(4,5).set(#); // error
```



Static Data Members

- Change the interface in Sales_data.h:

```
1 class Sales_data {
2 public:
3     Sales_data(const string &s): bookNo{s} { ++counter; }
4 private:
5     std::string bookNo;
6     unsigned units_sold{0}; // in-class initialiser
7     double revenue{0.0};    // in-class initialiser
8     // Count how many objs are created by this constructor
9     static int counter;
10};
```

- Add the definition in Sales_data.cpp:

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

- User code:

```
1 std::cout << Sales_data::counter << std::endl; // 0
2 Sales_data d1{"123"};
3 Sales_data d2{"456"};
4 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.

```
1  class Foo {  
2  public:  
3      Foo(const std::vector<int> v) : data{v} {}  
4  private:  
5      std::vector<int> data;  
6  };  
7  
8  std::vector<int> ivec;  
9  Foo foo = ivec;  // create a Foo object by assigning a  
10                 // std::vector to it
```

Implicit Type Conversions

- One type conversion is allowed:

```
1 std::string null_book = "123";  
2  
3 Sales_data data1{"123", 1, 20.0};  
4 data1.combine(null_book); // creates a new sales_data object  
5                           // for null_book  
6 print(std::cout, data1) << endl;
```

- But two type conversions are prohibited:

```
1 // two conversions disallowed:  
2 // (1) convert "123" of type const char[4] to string  
3 // (2) convert the string to Sales_data  
4 Sales_data data1{"123", 1, 20.0};  
5 data1.combine("123");  
6 print(std::cout, data1) << endl;
```

- This works but looks a bit problematic!

```
1 Sales_data data1{"123", 1, 20.0};  
2 data1.combine(std::cin);  
3 // same as: data1.combine(Sales_data{std::cin});  
4 print(std::cout, data1) << endl;
```

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.

```
1 #include <string>
2 #include <iostream>
3
4 class Sales_data {
5 public:
6     Sales_data(const std::string &s, unsigned n, double p):
7         bookNo{s}, units_sold{n}, revenue{p*n} { }
8     explicit Sales_data(const std::string &s): bookNo{s} { }
9     explicit Sales_data(std::istream &);
10    ...
```

Read §7.5.4.

(Explicit) Type Conversions

- A constructor must be called explicitly:

```

1 std::string null_book = "123";
2
3 Sales_data data1{"123", 1, 20.0};
4
5 data1.combine(null_book);           // error
6
7 data1.combine(Sales_data{null_book}); // ok

```

- Same here:

```

1 Sales_data data1{"123", 1, 20.0};
2 data1.combine(std::cin);           // error
3
4 data1.combine(Sales_data{std::cin}); // ok
5
6 data1.combine(static_cast<Sales_data>(std::cin)); // ok
7 // The compiler examines the Sales_data class to find
8 // an appropriate constructor for the conversion
9

```

vector: the Constructor That Takes a Size is Explicit

```
1 #include<iostream>
2 #include<vector>
3
4 int main() {
5     std::vector<int> v(10); // ok - note: needs to be () not {}
6     std::vector<int> v = 10; // error: explicit
7
8     void f(std::vector<int>); // declare function f
9     f(10); // error: explicit
10    f(std::vector<int>(10)); // ok
11 }
```

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)

```
1 typedef double Money;  
2 std::string bal;  
3 class Account {  
4 public:  
5     Money balance() { return bal; }  
6 private:  
7     Money bal;  
8 }
```

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

Name Resolution

```
1 void foo1() {}
2 typedef int value_type;
3 class Foo {
4     value_type foo1(); // OK: foo1 redefined in different scope
5     typedef double value_type; // error: cannot redefine type
6 };
```

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

```
1 void foo1() {}
2 typedef int value_type;
3 class Foo {
4     // swap the previous lines around, typedef not yet used
5     typedef double value_type; // OK: value_type redefined
6     value_type foo1(); // OK: foo1 redefined
7 };
```


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

```
1  int x = 1;
2  struct Foo {
3      int x;
4      Foo() : x{2} { }
5      void print(const int& x) { std::cout << x; }
6  };
7
8  Foo f;
9  f.print(3); // what is printed?
```

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

```
1  #include<iostream>
2
3  class Foo {
4  public:
5      Foo() : x(2) { }
6      void print(int x);
7  private:
8      int x;
9  };
10
11  int g() { return 3; }
12  void Foo::print(int x) { std::cout << g() << std::endl; }
13
14  int main() {
15      Foo f;
16      f.print(g());
17  }
```

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

```
1 #include<iostream>
2
3 class Foo {
4 public:
5     static int g() { return 4; }
6     Foo() : x(2) { }
7     void print(int x);
8 private:
9     int x;
10 };
11
12 int g() { return 3; }
13 void Foo::print(int x) { std::cout << g() << std::endl; }
14
15 int main() {
16     Foo f;
17     f.print(g());
18 }
```

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

```
1 #include<iostream>
2
3 class Foo {
4 public:
5     static int g() { return 4; }
6     Foo() : x(2) { }
7     void print(int x);
8 private:
9     int x;
10 };
11
12 int g() { return 3; }
13 void Foo::print(int x) { std::cout << ::g() << std::endl; }
14
15 int main() {
16     Foo f;
17     f.print(g());
18 }
```

Object-Based Programming: Copy Control

- ➊ Copy Constructor
- ➋ Copy Assignment
- ➌ Destructors
- ➍ Move Constructor
- ➎ Move Assignment
- ➏ The Big Five:
 - C++: 1 – 3 (the Big Three)
 - C++11: Added 4 and 5 (the Big Five now)
- ➐ Lvalue and Rvalue References!

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:

```
1 class A {  
2     ~A() { ... }  
3     ...  
4 }
```

- 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:

```
1 Class A {  
2 private:  
3     X1 d1;  
4     X2 d2;  
5     ...  
6     Xn dn;  
7 }
```

```
1 // The synthesised destructor  
2 ~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`:

```
1 ~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?**

When Is a Destructor Called?

- The destructor for a built-in type does nothing

```
1 { // new scope
2   Sales_data d1;
3   std::vector<Sales_data> v(10);
4   int i;
5 } -----> All destructors are called HERE
6 // d1, v destroyed but nothing to do for i
```

- A pointer has two associated objects:

```
1 { // new scope
2   Sales_data *p = new Sales_data{};
3   Sales_data *q = new Sales_data{};
4   ...
5   delete p; // means really delete *p!
6               // (1) the destructor for *p to be called
7               // (2) the memory for *p is freed
8 }
9 // Nothing to be done to destroy p and q themselves
10 // The pointed to object by q leaks!
```

The lifetime of a heap object is the entire program execution.

Copy Constructor

- A constructor of the form:

```
1 class A {  
2     A(const A &a) { ... }  
3     ...  
4 }
```

- The first must be a reference to the **same type** (usually a const reference since it cannot be bound to a temporary object otherwise)
- Any additional parameters must have default values

- Used to initialise an object from another:

```
1 A a = other; // copy construction/initialisation  
2 f(other); // pass other to a parameter of a non-ref type  
3 return other; // return from a func with a non-ref type  
4 catch (A other); // catch by value (covered later)  
5 std::array<A, 2> a { A(), A() };
```

The Compiler-Generated Copy Constructor

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

```
1 Class A {  
2 private:  
3     X1 d1;  
4     X2 d2;  
5     ...  
6     Xn dn;  
7 }
```

```
1 // The synthesised constructor  
2 A(const A& a) :  
3     d1(a.d1),  
4     d2(a.d2),  
5     ...  
6     dn(a.dn)  
7 { }
```

- 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:

```
1 Sales_data::Sales_data(const Sales_data &d) :  
2     bookNo{d.bookNo},  
3     units_sold{d.units_sold},  
4     revenue{d.revenue}  
5 { }
```

Direct vs. Copy Initialisation (Pages 497 – 498)

```
1 #include <iostream>
2
3 class MyString {
4 public:
5     MyString(const char* s) : str{s} {}
6     MyString(const MyString &rhs) : str{rhs.str} {}
7 private:
8     std::string str;
9 };
10
11 int main() {
12     MyString s1{"abc"};    // direct initialisation
13     MyString s2 = "abc";  // copy initialisation
14 }
```

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

The compiler may bypass the copy/move ctor to create an object directly.

Constraints on Copy Initialisation

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

```
1 #include<iostream>
2 #include<vector>
3
4 int main() {
5     std::vector<int> v(10);           // ok
6     std::vector<int> v = 10;         // error: explicit
7
8     void f(std::vector<int>);
9     f(10);                           // error: explicit
10    f(std::vector<int>(10));          // ok
11 }
```

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

```
1 class A {  
2     A& operator=(const A &a) { ... }  
3     ...  
4 }
```

- 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;  
a = b = c;
```

```
int i, j, k;  
i = j = k;
```

- The syntax:

```
1 Sales_data x1, x2;  
2 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:

```
1 Class A {  
2 private:  
3     X1 d1;  
4     X2 d2;  
5     ...  
6     Xn dn;  
7 }
```

```
1 // The synthesised operator=  
2 A &operator=(const A &rhs) {  
3     d1 = rhs.d1;  
4     d2 = rhs.d2;  
5     ...  
6     dn = rhs.dn;  
7 }
```

- 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
- The synthesised assignment for Sales_data:

```
1 Sales_data&  
2 Sales_data::operator=(const Sales_data &rhs) {  
3     bookNo = rhs.bookNo;  
4     units_sold = rhs.units_sold;  
5     revenue = rhs.revenue;  
6 }
```

Preventing Copies (§13.1.5 – 13.1.6)

```

1 class NoCopy {
2 public:
3     NoCopy() = default;
4     NoCopy(const NoCopy &data) = delete;
5     NoCopy &operator=(const NoCopy &data) = delete;
6     ~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:

```

1 // error: cannot copy
2 std::ostream print(std::ostream os, const Sales_data&);
3 print(std::cout, data);
4
5 // ok: pass by a reference
6 std::ostream& print(std::ostream &os, const Sales_data&);
7 print(std::cout, data);

```

- If a class has a deleted destructor, objects can only be created via `new` and can never be freed.

To be continued...

Next week:

- Copy control continued
- Move semantics
- Operator Overloading