



COMP 2012H Honors Object-Oriented Programming and Data Structures

Topic 11: Object Initialization, Construction and Destruction

Dr. Desmond Tsoi

Department of Computer Science & Engineering
The Hong Kong University of Science and Technology
Hong Kong SAR, China

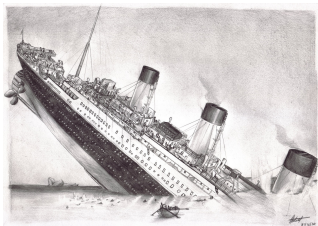


Class Object Initialization

- If **all** data members of a class are **public** (so the class is actually a basic struct), they can be initialized when they are created using the brace initializer "{ }".

```
class Word                                /* File: public-member-init.cpp */
{
    public:
        int frequency;
        const char* str;
};

int main() { Word movie = {1, "Titanic"}; }
```



Class Object Initialization ..

- What happens if some of data members are **private**?

```
class Word                                /* File: private-member-init.cpp */
{
    public:
        int frequency;
    private:
        const char* str;
};
```

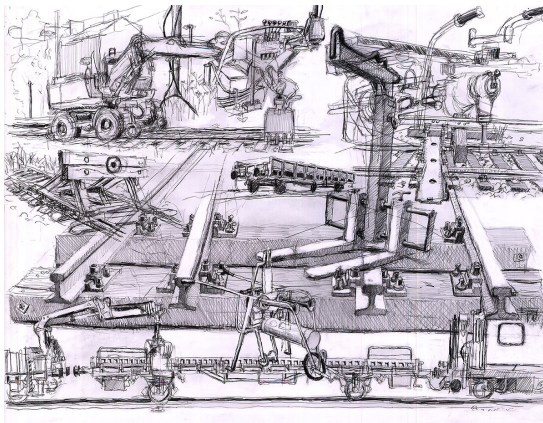
```
int main() { Word movie = {1, "Titanic"}; }
```

```
private-member-init.cpp:9:40: error: could not convert '{1, "Titanic"}'
      from '<brace-enclosed initializer list>' to 'Word'
```








```
int main() { Word movie = {1, "Titanic"}; }
```

Part I

Constructors



Different Types of C++ Constructors

	blank CD	default constructor
 	MP3 to CD	conversion constructor
 	pirated CD	copy constructor
<p>Memories I Dreamed a Dream Phantom of the Opera Don't Cry for me Argentina</p> 		other constructors

C++ Constructor Member Functions

```
Word movie; // Default constructor
Word director = "J. Cameron"; // Implicit conversion constructor
Word sci_fi("Avatar"); // Explicit conversion constructor
Word drama {"Titanic"}; // C++11: Explicit conversion constructor
Word* p = new Word("action", 1); // General constructor
```

- Syntactically, a class **constructor** is a special **member function** having the **same** name as the class.
- A **constructor** must **not** specify a return type or explicitly returns a value — **not** even the **void** type.
- A **constructor** is called **whenever** an object is created:
 - ▶ object creation
 - ▶ object **passed** to a function by **value**
 - ▶ object **returned** from a function by **value**

Default Initializers for Non-static Data Members (C++11)

```
class Word                                /* File: default-initializer.cpp */
{
    int frequency {0};
    const char* str {nullptr};
};

int main() { Word movie; }
```

- C++11 allows **default** values for **non-static data members** of a class.
- Nevertheless, C++ supports a more **general** mechanism for user-defined initialization of class objects through **constructor member functions**.
- During an object's construction, if the **constructor** does not initialize a **non-static member**, it will have the value of its **default initializer** if it exists, otherwise its value is undefined.

Default Constructor

Default Constructor `X::X()` for Class `X`

A constructor that *can be* called with **no** arguments.

```
class Word                                /* File: default-constructor.cpp */
{
    private:
        int frequency;
        char* str;
    public:
        Word() { frequency = 0; str = nullptr; } // Default constructor
};

int main()
{
    Word movie; // No arguments => expect default constructor
}
```

- c.f. Variable definition of basic data types: `int x`; `float y`;
- It is used to create objects with user-defined **default** values.

Compiler-Generated Default Constructor

```
class Word    /* File: compiler-default-constructor.cpp */
{    // Implicitly private members
    int frequency;
    char* str;
};

int main() { Word movie; }
```

- If there are **no** user-defined constructors in the definition of class X, the compiler will generate the following **default constructor** for it,

X::X() { }

- **Word::Word() { }** only creates a Word object with enough space for its **int** component and **char*** component.
- The **initial** values of the data members **cannot** be trusted.

Default Constructor: Common Bug

- Only when **no** user-defined constructors are found, will the compiler automatically supply the simple default constructor, **X::X(){ }**.

```
1  class Word                                /* File: default-constructor-bug.cpp */
2  {
3      private: int frequency; char* str;
4      public: Word(const char* s, int k = 0);
5  };
6
7  int main() { Word movie; }                // which constructor?
```

```
default-constructor-bug.cpp:7:19: error: no matching function for call to 'Word::Word()'
int main() { Word movie; } // which constructor?
      ~~~~~
```

```
default-constructor-bug.cpp:4:11: note: candidate: Word::Word(const char*, int)
public: Word(const char* s, int k = 0);
      ~~~~~
```

```
default-constructor-bug.cpp:4:11: note:   candidate expects 2 arguments, 0 provided
```

```
default-constructor-bug.cpp:1:7: note: candidate: constexpr Word::Word(const Word&)
default-constructor-bug.cpp:1:7: note:   candidate expects 1 argument, 0 provided
```

```
default-constructor-bug.cpp:1:7: note: candidate: constexpr Word::Word(Word&&)
default-constructor-bug.cpp:1:7: note:   candidate expects 1 argument, 0 provided
```

Implicit Conversion Constructor(s)

```
#include <cstring>    /* File: implicit-conversion-constructor.cpp */
class Word
{
    private: int frequency; char* str;
    public:
        Word(char c)
            { frequency = 1; str = new char[2]; str[0] = c; str[1] = '\0'; }
        Word(const char* s)
            { frequency = 1; str = new char [strlen(s)+1]; strcpy(str, s); }
};

int main()
{
    Word movie("Titanic");           // Explicit conversion
    Word movie2 {'A'};               // Explicit conversion
    Word movie3 = 'B';               // Implicit conversion
    Word director = "James Cameron"; // Implicit conversion
}
```

- A constructor accepting a **single argument** specifies a **conversion** from its argument type to the type of its class:

Word(const char*): const char* \longrightarrow Word

Word(char): char \longrightarrow Word

Implicit Conversion Constructor(s) ..

```
#include <cstring>          /* File: conversion-constructor-default-arg.cpp */
class Word
{
    int frequency; char* str;
public:
    Word(const char* s, int k = 1)    // Still conversion constructor!
    {
        frequency = k;
        str = new char [strlen(s)+1]; strcpy(str, s);
    }
};

int main()
{
    Word *p = new Word {"action"};    // Explicit conversion
    Word movie("Titanic");            // Explicit conversion
    Word director = "James Cameron";  // Implicit conversion
}
```

- A class may have **more** than one **conversion constructor**.
- A constructor may have multiple arguments; if all but **one** argument have **default values**, it is still a **conversion constructor**.

Implicit Conversion By Surprise

```
#include <iostream>      /* File: implicit-conversion-surprise.cpp */
#include <cstring>
using namespace std;
class Word
{
private:
    int frequency; char* str;
public:
    Word(char c)
        { frequency = 1; str = new char[2]; str[0] = c; str[1] = '\0';
          cout << "call implicit char conversion\n"; }
    Word(const char* s)
        { frequency = 1; str = new char [strlen(s)+1]; strcpy(str, s);
          cout << "call implicit const char* conversion\n"; }
    void print() const { cout << str << " : " << frequency << endl; }
};

void print_word(Word x) { x.print(); }
int main() { print_word("Titanic"); print_word('A'); return 0; }
```

- To **disallow** perhaps unexpected **implicit conversion** (c.f. coercion among basic types), add the keyword '**explicit**' before a **conversion constructor**.

Explicit Conversion Constructor(s)

```
1  #include <cstring>      /* File: explicit-conversion-constructor.cpp */
2  class Word
3  {
4      private:
5          int frequency; char* str;
6      public:
7          explicit Word(const char* s)
8              { frequency = 1; str = new char [strlen(s)+1]; strcpy(str,s); }
9  };
10
11 int main()
12 {
13     Word *p = new Word("action");    // Explicit conversion
14     Word movie("Titanic");           // Explicit conversion
15     Word director = "James Cameron"; // Bug: implicit conversion
16 }
```

```
explicit-conversion-constructor.cpp:15:21: error: conversion
  from 'const char [14]' to non-scalar type 'Word' requested
    Word director = "James Cameron"; // Bug: implicit conversion
                   ~~~~~~
```

Copy Constructor

```
#include <iostream>      /* File: copy-constructor.cpp */
#include <cstring>
using namespace std;

class Word
{
private:
    int frequency; char* str;
    void set(int f, const char* s)
        { frequency = f; str = new char [strlen(s)+1]; strcpy(str,s); }
public:
    Word(const char* s, int k = 1)
        { set(k, s); cout << "conversion\n"; }
    Word(const Word& w)
        { set(w.frequency, w.str); cout << "copy\n"; }
};

int main()
{
    Word movie("Titanic");           // which constructor?
    Word song(movie);                // which constructor?
    Word ship = movie;               // which constructor?
    Word actress {"Kate"};          // which constructor?
}
```

Copy Constructor ..

Copy Constructor: $X::X(\text{const } X\&)$ for Class X

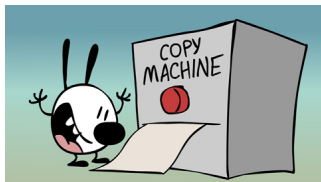
A constructor that has exactly **one argument** of the **same class** passed by its **const reference**.

It is called upon when:

- parameter **passed** to a function by **value**.
- **initialization** using the **assignment syntax** though it actually is **not** an assignment:

```
Word x {"Star Wars"}; Word y = x;
```

- object **returned** by a function by **value**.



Return-by-Value \Rightarrow Copy Constructor

```
1  #include <iostream>      /* File: return-by-value.cpp */
2  #include <cstring>
3  using namespace std;
4  class Word
5  {
6  private:
7      int frequency; char* str;
8      void set(int f, const char* s)
9          { frequency = f; str = new char [strlen(s)+1]; strcpy(str, s); }
10 public:
11     Word(const char* s, int k = 1) { set(k, s); cout << "conversion\n"; }
12     Word(const Word& w) { set(w.frequency, w.str); cout << "copy\n"; }
13     void print() const { cout << str << " : " << frequency << endl; }
14     Word to_upper_case() const
15     {
16         Word x(*this);
17         for (char* p = x.str; *p != '\0'; p++) *p += 'A' - 'a';
18         return x;
19     }
20 };
21 int main()
22 {
23     Word movie {"titanic"}; movie.print();
24     Word song = movie.to_upper_case(); song.print();
25 }
```

Copy Elision and Return Value Optimization

- How many calls of the copy constructor do you expect?
- Below is the actual output from the previous example:

```
conversion  
titanic : 1  
copy  
TITANIC : 1
```

- **Return value optimization (RVO)** is a compiler optimization technique which applies **copy elision** in a **return** statement.
- It omits **copy/move** operation by constructing a local (temporary) object directly into the function's return value!
- For the example, codes that are supposed to be run by 'x' are run directly on 'song'.

Question: Which line calls the copy constructor?

Default Copy Constructor

```
class Word /* File: default-copy-constructor.cpp */
{
    private: ...
    public: Word(const char* s, int k = 0) { ... };
};

int main()
{
    Word movie {"Titanic"}; // which constructor?
    Word song {movie};      // which constructor?
    Word song = movie;      // which constructor?
}
```

- If **no** copy constructor is defined for a class, the compiler will automatically supply it a **default copy constructor**.

`X(const X&) { /* memberwise copy */ }`

- \Rightarrow **memberwise copy** (aka **copy assignment**) by calling the **copy constructor** of each data member:
 - ▶ copy movie.frequency to song.frequency
 - ▶ copy movie.str to song.str
- It works even for array members by copying each array element.

Default Memberwise Assignment

- Objects of basic data types support many **operator** functions such as $+$, $-$, \times , $/$.
- C++ allows user-defined types to overload **most** (not all) operators to re-define the behavior for their objects — **operator overloading**.
- Unless you re-define the assignment operator ' $=$ ' for a class, the compiler generates the **default assignment operator function** — **memberwise assignment** — for it.
- Different from the **default copy constructor**, the **default assignment operator** $=$ will perform **memberwise assignment** by calling the assignment operator $=$ of each data member:
 - ▶ `song.frequency = movie.frequency`
 - ▶ `song.str = movie.str`
- Again for array members, each array element is assigned.
- **Memberwise assignment/copy** is usually **not** what you want when memory allocation is required for the class members.

Default Memberwise Assignment With Array Data

```
#include <iostream>      /* File: default-assign-problem1.cpp */
#include <cstring>
using namespace std;
class Word
{
private:
    int frequency; char str[100];
    void set(int f, const char* s) { frequency = f; strcpy(str, s); }
public:
    Word(const char* s, int k = 1)
        { set(k, s); cout << "\nImplicit const char* conversion\n"; }
    Word(const Word& w) { set(w.frequency, w.str); cout << "\nCopy\n"; }

    void print() const    // Also prints the address of object's str array
        { cout << str << " : " << frequency << " ; "
          << reinterpret_cast<const void*>(str) << endl; }
};

int main()
{
    Word x("rat"); x.print();      // Conversion constructor
    Word y = x;    y.print();      // Copy constructor
    Word z("cat"); z.print();      // Conversion constructor
    z = x;         z.print();      // Default assignment operator
}
```

Default Memberwise Assignment With Array Data ..

Implicit const char* conversion

```
rat : 1 ; 0x7fff5cd2e5d4
```

Copy

```
rat : 1 ; 0x7fff5cd2e56c
```

Implicit const char* conversion

```
cat : 1 ; 0x7fff5cd2e504
```

```
rat : 1 ; 0x7fff5cd2e504
```



Default Memberwise Assignment With Pointer Data

```
#include <iostream>      /* File: default-assign-problem2.cpp */
#include <cstring>
using namespace std;
class Word
{
private: int frequency; char* str;
    void set(int f, const char* s)
        { frequency = f; str = new char [strlen(s)+1]; strcpy(str, s); }
public:
    Word(const char* s, int k = 1)
        { set(k, s); cout << "\nImplicit const char* conversion\n"; }
    Word(const Word& w) { set(w.frequency, w.str); cout << "\nCopy\n"; }

    void print() const // Also prints the address of object's str array
        { cout << str << " : " << frequency << " ; "
            << reinterpret_cast<void*>(str) << endl; }
};

int main()
{
    Word x("rat");      x.print(); // Conversion constructor
    Word y = x;          y.print(); // Copy constructor
    Word z("cat", 2);    z.print(); // Conversion constructor
    z = x;               z.print(); // Default assignment operator
}
```

Default Memberwise Assignment With Pointer Data ..

Implicit const char* conversion

```
rat : 1 ; 0x7fc7dbd039c0
```

Copy

```
rat : 1 ; 0x7fc7dbd039d0
```

Implicit const char* conversion

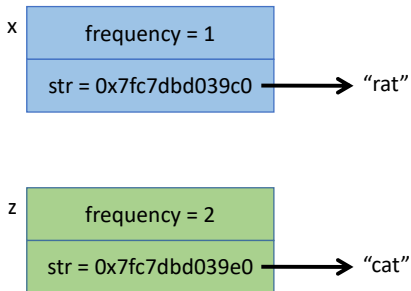
```
cat : 2 ; 0x7fc7dbd039e0
```

```
rat : 1 ; 0x7fc7dbd039c0
```

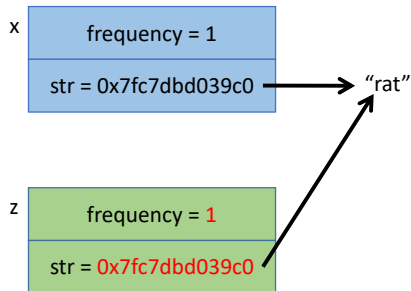


Problem With Default Memberwise Assignment

Before $z = x$



After $z = x$



Quiz: Constructors

Which constructor is called in the following statements?

1. Word nothing;
2. Word dream_grade('A');
3. Word major { "COMP" };
4. Word hkust = "hkust";
5. Word exchange_to(hkust);
6. Word grade = dream_grade;
7. Word grade {dream_grade};



Uniform Initialization Using the `{}` Initializers Again

- In general, initializations may be done using `()`, `=`, or `{}`
`int x(1); int y = 2; int z {3};`
- The **braced initialization** syntax helps avoid some misleading syntax from the other two kinds:

1. when `=` doesn't really mean assignment!

`Word word1 = word2; // What is this?`

2. when `()` doesn't really mean calling the default constructor!

`Word w(); // What is this?`

In both cases, **braced initialization** works fine:

`Word word1 {word2}; Word w {};`

- When a class member of **user-defined types** is initialized, its corresponding **constructor** will be called.
- `()` initializer **cannot** be used to do **default** initialization of **non-static** class data members.

Constructors and Function Overloading

- **Overloading** allows programmers to use the **same** name for **functions** that do *similar* things but with **different** input **arguments**.
- **Constructors** are often **overloaded**.

```
class Word                                /* File: overload-constructor.cpp */
{
    private:
        int frequency;
        char* str;

    public:
        Word();                            // Default constructor
        Word(const char* s, int k = 1);    // Conversion constructor
        Word(const Word& w);               // Copy constructor
};
```

Review: Function Overloading

- In general, function names can be overloaded in C++.
- Actually, operators are often overloaded.
e.g., What is the type of the operands for “+”?

```
#include <iostream>      /* File: overload-function.cpp */
#include <cstring>
using namespace std;
class Word
{
private:
    int frequency; char* str;
public:
    void set() const { cout << "Input the string: "; cin >> str; } // Error!
    void set(int k) { frequency = k; }
    void set(char c) { str = new char [2]; str[0] = c; str[1] = '\0'; }
    void set(const char* s) { str = new char [strlen(s)+1]; strcpy(str, s); }
};

int main()
{
    Word movie;          // Which constructor?
    movie.set();          // Which set function?
}
```

Review: Functions with Default Arguments

- If a function shows some **default** behaviors most of the time, and some **exceptional** behaviors only **once awhile**, specifying **default arguments** is a **better** option than using **overloading**.
- There may be more than one **default argument**.

```
void upload(char* prog, char os = LINUX, char format = TEXT);
```

- Parameters **without** default values **must** be declared to the **left** of those with **default arguments**. The following is an error:

```
void upload(char os = LINUX, char* prog, char format = TEXT);
```

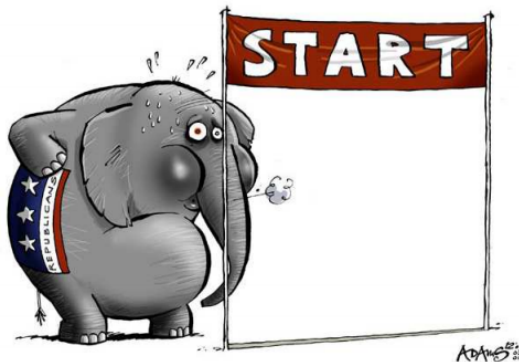
- A parameter can have its **default argument** specified only **once** in a file, usually in the public **header file**, and not in the function definition. Thus, the following is an error.

```
class Word // File: word.h
{
    ...
public:
    Word(const char* s, int k = 1);
}
```

```
#include "word.h" // File: word.cpp
Word::Word(const char* s, int k = 1)
{
    ...
}
```

Part II

Member_INITIALIZER List



Member Initializer List (MIL)

- So far, data members of a class are initialized **inside** the body of its **constructors**.
- It is actually preferred to initialize them **before** the constructors' function body through the **member initializer list** by calling their **own constructors**.
 - ▶ It starts **after** the constructor header but **before** the opening **{** .
 - ▶ **: *member₁(expression₁), member₂(expression₂), ...***
 - ▶ The order of the members in the list doesn't matter; the actual execution order is their order in the class declaration.

Member_INITIALIZER List ..

```
class Word                                /* File: mil-word.h */
{
    private:
        char lang;
        int freq;
        char* str;

    public:
        Word() : lang('E'), freq(0), str(nullptr) { };

        /* Or, using the braced initialization syntax as follows
           Word() : lang{'E'}, freq{0}, str{nullptr} { };
        */

        Word(const char* s, int f = 1, char g = 'E') : lang(g), freq(f)
            { str = new char [strlen(s)+1]; strcpy(str, s); }

        void print() const { cout << str << " : " << freq << endl; }
};
```

Member Initializer List

- Since the **MIL** calls the constructors of the data member, it works well for data members of **user-defined types**.
- Thus, it is better to perform initialization by **MIL** than by **assignments** inside constructors.
- Make sure that the corresponding member constructors **exist!**

```
class Word_Pair          /* File: mil-word-pair.h */
{
    private:
        Word w1; Word w2;
    public:
        Word_Pair(const char* s1, const char* s2) : w1(s1,5), w2(s2) { }
        void print() const
        {
            cout << "word1 = "; w1.print();
            cout << "word2 = "; w2.print();
        }
};
```

Problem If Member Initialization List Is Not Used

```
class Word_Pair /* File: member-class-init-by-mil.h */
{
    private:
        Word w1; Word w2;
    public:
        Word_Pair(const char* s1, const char* s2) : w1(s1,5), w2(s2) { }
};
```

⇒ w1 and w2 are initialized using the **conversion constructor**,
Word(const char*, int = 1, char = 'E')

```
Word_Pair(const char* x, const char* y) { w1 = x; w2 = y; }
```

⇒ **error-prone** because w1 and w2 are initialized by assignment. If the **assignment operator** function is **not** appropriately defined, the **default memberwise assignment** may **not** be good enough.

Initialization of const or Reference Members

- **const** or **reference** members **must** be initialized using **member initializer list** if they don't have **default initializers**.
- c.f. `float y; float& z = y; const int x = 123;`

```
#include <iostream>          /* File: mil-const-ref.cpp */
using namespace std;
int a = 5;

class Example
{
    const int const_m = 3;
    int& ref_m = a;
public:
    Example() { }
    Example(int c, int& r) : const_m(c), ref_m(r) { }
    void print() const { cout << const_m << "\t" << ref_m << endl; }
};

int main()
{
    Example x; x.print();
    int b = 55; Example y(10, b); y.print();
}
```

Initialization of const or Reference Members ..

- It cannot be done using default arguments.

```
1  #include <iostream>      /* File: mil-const-member-error.cpp */
2  using namespace std;
3  class Word
4  {
5      private:
6          const char lang; int freq; char* str;
7      public:
8          Word() : lang('E'), freq(0), str(nullptr) { };
9          Word(const char* s, int f = 1, char g = 'E')
10             { str = new char [strlen(s)+1]; strcpy(str, s); }
11          void print() const
12             { cout << str << " : " << freq << endl; }
13  };
14
15  int main() { Word x("hkust"); }
```

mil-const-member-error.cpp:9:5: error: constructor for 'Word'
must explicitly initialize the const member 'lang'

Word(const char* s, int f = 1, char g = 'E')

^

Part III

Garbage Collection & Destructor



Destructor

Destructor $X::\sim X()$ for Class X

The destructor of a class is invoked **automatically** whenever its object goes out of (e.g., function/block) scope.

- A **destructor** is a special class member function.
- A **destructor** takes **no arguments**, and has **no return type**.
- Thus, there can only be **one destructor** for a class.
- If **no destructor** is defined, the compiler will automatically generate a **default destructor** which does *nothing*:

$$X::\sim X() \{ \}$$

- The **destructor** itself does not actually release the object's memory.
- The **destructor** performs **termination housekeeping** before the object's memory is reclaimed by the system.

Sometimes Default Destructor Is Not Good Enough

```
void Example() /* File: default-destructor-problem.cpp */
{
    Word x("bug", 4);
    ...
}

int main() { Example(); .... }
```

- On return from `Example()`, the `local` `Word` object “`x`” of `Example()` is destructed from the `run-time stack`.
- i.e., the storage of `(int)` `x.frequency` and `(char*)` `x.str` are released.

Question: How about the memory dynamically allocated for the string, “`bug`” that `x.str` points to?

User-Defined Destructor

- C++ supports a general mechanism for user-defined destruction of objects through **destructor member function**.
- Usually needed when there are **pointer members** pointing to memory dynamically allocated by **constructor(s)** of the class.

```
#include <cstring>          /* File: destructor.cpp */
class Word
{
private:
    int frequency; char* str;
public:
    Word() : frequency(0), str(nullptr) { };
    Word(const char* s, int k = 0): frequency(k)
        { str = new char [strlen(s)+1]; strcpy(str, s); }
    ~Word() { delete [] str; }
};

int main()
{
    Word* p = new Word {"Titanic"};
    Word* x = new Word [5];
    delete p;           // Destruct a single object
    delete [] x;        // Destruct an array of objects
}
```

Bug: Default Memberwise Assignment

```
1  #include <cstring>          /* File: default-assign-bug.cpp */
2
3  class Word
4  {
5      private:
6          int frequency; char* str;
7
8      public:
9          Word() : frequency(0), str(nullptr) { }
10         Word(const char* s, int k = 0): frequency(k)
11             { str = new char [strlen(s)+1]; strcpy(str, s); }
12         ~Word() { delete [] str; }
13 };
14
15 void Bug(Word& x) { Word bug("bug", 4); x = bug; }
16
17 int main() { Word movie {"Titanic"}; Bug(movie); return 0; }
```

Question: How many bugs are there?

Summary: Compiler-generated Member Functions

Unless you define the following, they will be **implicitly** generated by the compiler for you:

1. **default constructor**
(but only if you don't define other constructors)
2. default **copy constructor**
3. default **(copy) assignment operator** function
4. default **move constructor** (C++11)
5. default **move assignment operator** function (C++11)
6. **default destructor**

C++11 allows you to **explicitly** generate or not generate them:

- to generate: **= default;**
- not to generate: **= delete;**

Example: = default; = delete;

```
#include <iostream>          /* File: default-delete.cpp */
#include <cstring>
using namespace std;
class Word
{
private:
    int frequency {0}; char* str {nullptr};
public:
    Word() = default; // Still want the simple default constructor
    Word(const Word& w) = delete; // Words can't be copied
    Word(const char* s, int k) : frequency(k)
        { str = new char [strlen(s)+1]; strcpy(str, s); }
    void print() const
        { cout << ((str == nullptr) ? "not-a-word" : str)
              << " : " << frequency << endl; }
};

int main()
{
    Word x; x.print();
    Word y("good", 3); y.print();
    Word z(y);          // Error: call to deleted constructor of 'Word'
}
```

Part IV

Order of Construction & Destruction



“Has” Relationship

- When an object A has an object B as a data member, we say
“A has a B.”
- It is easy to see which objects **have** other objects. All you need to do is to look at the **class definition**.

```
/* File: example-has.h */  
class B { ... };  
  
class A  
{  
    private:  
        B my_b;  
  
    public:  
        // Declaration of public members or functions  
};
```

Cons/Destruction Order: Postoffice Has a Clock

```
class Clock                      /* File: postoffice1.h */
{
    public:
        Clock() { cout << "Clock Constructor\n"; }
        ~Clock() { cout << "Clock Destructor\n"; }
};

class Postoffice
{
    Clock clock;
    public:
        Postoffice() { cout << "Postoffice Constructor\n"; }
        ~Postoffice() { cout << "Postoffice Destructor\n"; }
};
```

```
#include <iostream> /* File postoffice1.cpp */
using namespace std;
#include "postoffice1.h"
int main()
{
    cout << "Beginning of main\n";
    Postoffice x;
    cout << "End of main\n";
}
```

Beginning of main
Clock Constructor
Postoffice Constructor
End of main
Postoffice Destructor
Clock Destructor

Cons/Destruction Order: Postoffice Has a Clock ..

- When an object is constructed, all its **data members** are constructed **first**.
- The order of **destruction** is the exact **opposite** of the order of **construction**: The Clock **constructor** is called **before** the Postoffice **constructor** code; but, the Clock **destructor** is called **after** the Postoffice **destructor** code.
- As always, construction of data member objects is done by calling their appropriate **constructors**.
 - ▶ If you do not do this **explicitly** then their **default constructors** are assumed. Make sure they exist! That is,

```
Postoffice::Postoffice() { }
```

is equivalent to,

```
Postoffice::Postoffice() : clock() { }
```
 - ▶ Or, you may do this **explicitly** by calling their appropriate constructors using the **member initialization list** syntax.

Cons/Destruction Order: Postoffice “Owns” a Clock

```
class Clock                      /* File: postoffice2.h */
{
public:
    Clock() { cout << "Clock Constructor\n"; }
    ~Clock() { cout << "Clock Destructor\n"; }
};

class Postoffice
{
    Clock* clock;
public:
    Postoffice()
    { clock = new Clock; cout << "Postoffice Constructor\n"; }
    ~Postoffice() { cout << "Postoffice Destructor\n"; }
};
```

```
/* File: postoffice2.cpp */
#include <iostream>
using namespace std;
#include "postoffice2.h"
int main()
{
    cout << "Beginning of main\n";
    Postoffice x;
    cout << "End of main\n";
}
```

Beginning of main
Clock Constructor
Postoffice Constructor
End of main
Postoffice Destructor

Cons/Destruction Order: Postoffice “Owns” a Clock ..

- Now the Postoffice “owns” a Clock.
- This is the terminology used in OOP. If A “owns” B, A only has a **pointer** pointing to B.
- The Clock object is constructed in the Postoffice **constructor**, but it is never destructed, since we have not implemented that.
- Remember that objects on the **heap** are never destructed automatically, so we have just created a **memory leak**.
- When object A **owns** object B, A is responsible for B's **destruction**.



Cons/Destruction Order: Postoffice “Owns” a Clock ...

```
class Clock                      /* File: postoffice3.h */
{
public:
    Clock() { cout << "Clock Constructor\n"; }
    ~Clock() { cout << "Clock Destructor\n"; }
};

class Postoffice
{
    Clock* clock;
public:
    Postoffice()
    { clock = new Clock; cout << "Postoffice Constructor\n"; }
    ~Postoffice()
    { cout << "Postoffice Destructor\n"; delete clock; }
};
```

```
/* File: postoffice3.cpp */
```

```
#include <iostream>
```

```
using namespace std;
```

```
#include "postoffice3.h"
```

```
int main()
```

```
{
```

```
    cout << "Beginning of main\n";
```

```
    Postoffice x;
```

```
    cout << "End of main\n";
```

```
}
```

Beginning of main
Clock Constructor
Postoffice Constructor
End of main
Postoffice Destructor
Clock Destructor

Cons/Destruction Order: Postoffice Has Clock + Room

```
class Clock          /* File: postoffice4.h */
{
private: int HHMM;    // hour, minute
public:
    Clock() : HHMM(0)
        { cout << "Clock Constructor\n"; }
    ~Clock() { cout << "Clock Destructor\n"; }
};

class Room
{
public:
    Room() { cout << "Room Constructor\n"; }
    ~Room() { cout << "Room Destructor\n"; }
};

class Postoffice
{
private:
    Room room; Clock clock;
public:
    Postoffice()
        { cout << "Postoffice Constructor\n"; }
    ~Postoffice()
        { cout << "Postoffice Destructor\n"; }
};
```

```
/* File: postoffice4.cpp */
#include <iostream>
using namespace std;
#include "postoffice4.h"
int main()
{
    cout << "Beginning of main\n";
    Postoffice x;
    cout << "End of main\n";
}
```

Beginning of main
Room Constructor
Clock Constructor
Postoffice Constructor
End of main
Postoffice Destructor
Clock Destructor
Room Destructor

†† Note that the 2 data members, Clock and Room are **constructed first**, in the order that they appear in the Postoffice class.

Cons/Destruction Order: Postoffice Moves Clock to Room

```
class Clock                /* File: postoffice5.h */
{
public:
    Clock() { cout << "Clock Constructor\n"; }
    ~Clock() { cout << "Clock Destructor\n"; }
};

class Room
{
private:
    Clock clock;
public:
    Room() { cout << "Room Constructor\n"; }
    ~Room() { cout << "Room Destructor\n"; }
};

class Postoffice
{
private:
    Room room;
public:
    Postoffice()
        { cout << "Postoffice Constructor\n"; }
    ~Postoffice()
        { cout << "Postoffice Destructor\n"; }
};
```

```
/* File: postoffice5.cpp */
#include <iostream>
using namespace std;
#include "postoffice5.h"
int main()
{
    cout << "Beginning of main\n";
    Postoffice x;
    cout << "End of main\n";
}
```

Beginning of main
Clock Constructor
Room Constructor
Postoffice Constructor
End of main
Postoffice Destructor
Room Destructor
Clock Destructor

Cons/Destruction Order: Postoffice w/ a Temporary Clock

```
class Clock {                               /* File: postoffice6.h */
private: int HHMM;
public:
    Clock() : HHMM(0) { cout << "Clock Constructor\n"; }
    Clock(int hhmm) : HHMM(hhmm)
        { cout << "Clock Constructor at " << HHMM << endl; }
    ~Clock() { cout << "Clock Destructor at " << HHMM << endl; }
};

class Postoffice {
private: Clock clock;
public:
    Postoffice()
        { cout << "Postoffice Constructor\n"; clock = Clock(800); }
    ~Postoffice() { cout << "Postoffice Destructor\n"; }
};

#include <iostream>                          /* File: postoffice6.cpp */
using namespace std;
#include "postoffice6.h"
int main() {
    cout << "Beginning of main\n";
    Postoffice x;
    cout << "End of main\n";
}
```

Cons/Destruction Order: Postoffice w/ a Temp Clock ..

Beginning of main
Clock Constructor
Postoffice Constructor
Clock Constructor at 800
Clock Destructor at 800
End of main
Postoffice Destructor
Clock Destructor at 800

- Here a **temporary** clock object is created by **Clock(800)**.
- Like a ghost, it is created and destroyed **behind** the scene.

Default Member Initialization and Order of Construction

```
#include <iostream>      /* file: default-member-init.cpp */
using namespace std;

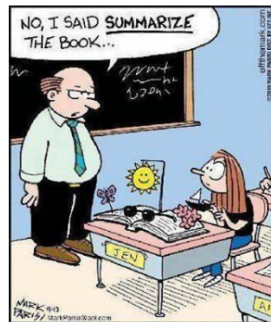
class A
{
    int a;
public:
    A(int z) : a(z) { cout << "call A's constructor: " << a << endl; }
    ~A() { cout << "call A's destructor: " << a << endl; }
    int get() const { return a; }
};

class B
{
    int b1 = 999;          // Remember: can't initialize by ( )
    A b2 = 10;             // Call A's conversion constructor
    A b3 {100};            // Call A's conversion constructor
public:
    B() { cout << "call B's default constructor" << endl; }
    ~B() { cout << "call B's destructor: " << b1 << "\t"
              << b2.get() << "\t" << b3.get() << endl; }
};

int main() { B x; return 0; }
```


Summary

- When an object is **constructed**, its data members are **constructed first**.
- When the object is **destroyed**, the data members are **destroyed after** the **destructor** code of the object has been executed.
- When object A **owns** other objects, remember to destroy them as well in A's **destructor**.
- By default, the **default constructor** is used for the data members.
- We can use a different constructor for the data members by using **member initializer list** — the “**colon syntax**”.



That's all!

Any questions?

