

# Object-Oriented Programming and Data Structures

## COMP2012: Object Initialization, Construction and Destruction

Brian Mak  
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 “{ }”.

```
1  class Word                                /* File: public-member-init.cpp */
2  {
3      public:
4          int frequency;
5          const char* str;
6  };
7
8  int main() { Word movie = {1, "Titanic"}; }
```

# Class Object Initialization ..

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

```
1  class Word                                /* File: private-member-init.cpp */
2  {
3      public:
4          int frequency;
5      private:
6          const char* str;
7  };
8
9  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 WAV	conversion constructor
 	pirated CD	copy constructor
Memories I Dreamed a Dream Phantom of the Opera Don't Cry for me Argentina		 other constructors

# C++ Constructor Member Functions

```
1 Word movie; // Default constructor
2 Word director = "J. Cameron"; // Implicit conversion constructor
3 Word sci_fi("Avatar"); // Explicit conversion constructor
4 Word drama {"Titanic"}; // C++11: Explicit conversion constructor
5 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)

```
1  class Word                                /* File: default-initializer.cpp */
2  {    // Implicitly private members
3      int frequency {0};
4      const char* str {nullptr};
5  };
6
7  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 the construction of a non-global object, if its **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.

```
1  class Word                                /* File: default-constructor.cpp */
2  {
3      private:
4          int frequency;
5          char* str;
6      public:
7          Word() { frequency = 0; str = nullptr; } // Default constructor
8  };
9
10 int main()
11 {
12     Word movie; // No arguments => expect default constructor
13 }
```

- 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

```
1  class Word    /* File: compiler-default-constructor.cpp */
2  {    // Implicitly private members
3      int frequency;
4      char* str;
5  };
6
7  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)

```
1  #include <cstring>      /* File: implicit-conversion-constructor.cpp */
2  class Word
3  {
4      private: int frequency; char* str;
5      public:
6          Word(char c)
7              { frequency = 1; str = new char[2]; str[0] = c; str[1] = '\0'; }
8          Word(const char* s) // Assumption: s != nullptr
9              { frequency = 1; str = new char [strlen(s)+1]; strcpy(str, s); }
10 };
11
12 int main()
13 {
14     Word movie("Titanic");           // Explicit conversion
15     Word movie2 {'A'};               // Explicit conversion
16     Word movie3 = 'B';               // Implicit conversion
17     Word director = "James Cameron"; // Implicit conversion
18 }
```

- 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) ..

```
1  #include <cstring>          /* File: conversion-constructor-default-arg.cpp */
2  class Word
3  {
4      int frequency; char* str;
5  public:
6      Word(const char* s, int k = 1)    // Still conversion constructor!
7      {
8          frequency = k;
9          str = new char [strlen(s)+1]; strcpy(str, s);
10     }
11 };
12
13 int main()
14 {
15     Word *p = new Word {"action"};    // Explicit conversion
16     Word movie("Titanic");            // Explicit conversion
17     Word director = "James Cameron";  // Implicit conversion
18 }
```

- 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

```
1  #include <iostream>          /* File: implicit-conversion-surprise.cpp */
2  #include <cstring>
3  using namespace std;
4  class Word
5  {
6  private:
7      int frequency; char* str;
8  public:
9      Word(char c)
10         { frequency = 1; str = new char[2]; str[0] = c; str[1] = '\0';
11           cout << "call implicit char conversion\n"; }
12      Word(const char* s)
13         { frequency = 1; str = new char [strlen(s)+1]; strcpy(str, s);
14           cout << "call implicit const char* conversion\n"; }
15      void print() const { cout << str << " : " << frequency << endl; }
16 };
17
18 void print_word(Word x) { x.print(); }
19 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

```
1  #include <iostream>          /* File: copy-constructor.cpp */
2  #include <cstring>
3  using namespace std;
4
5  class Word
6  {
7      private:
8          int frequency; char* str;
9          void set(int f, const char* s)
10             { frequency = f; str = new char [strlen(s)+1]; strcpy(str,s); }
11      public:
12          Word(const char* s, int k = 1)
13             { set(k, s); cout << "conversion\n"; }
14          Word(const Word& w)
15             { set(w.frequency, w.str); cout << "copy\n"; }
16  };
17
18  int main()
19  {
20      Word movie("Titanic");          // which constructor?
21      Word song(movie);               // which constructor?
22      Word ship = movie;              // which constructor?
23      Word actress {"Kate"};         // which constructor?
24  }
```

## 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** 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

```
1  class Word /* File: default-copy-constructor.cpp */
2  {
3      private: ...
4      public: Word(const char* s, int k = 0) { ... };
5  };
6  int main()
7  {
8      Word movie {"Titanic"}; // which constructor?
9      Word song {movie};      // which constructor?
10     Word song = movie;       // which constructor?
11 }
```

- 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

```
1  #include <iostream>      /* File: default-assign-problem1.cpp */
2  #include <cstring>
3  using namespace std;
4  class Word
5  {
6  private:
7      int frequency; char str[100];
8      void set(int f, const char* s) { frequency = f; strcpy(str, s); }
9  public:
10     Word(const char* s, int k = 1)
11         { set(k, s); cout << "\nImplicit const char* conversion\n"; }
12     Word(const Word& w) { set(w.frequency, w.str); cout << "\nCopy\n"; }
13
14     void print() const    // Also prints the address of object's str array
15         { cout << str << " : " << frequency << " ; "
16           << reinterpret_cast<const void*>(str) << endl; }
17 };
18
19 int main()
20 {
21     Word x("rat"); x.print();    // Conversion constructor
22     Word y = x;    y.print();    // Copy constructor
23     Word z("cat"); z.print();    // Conversion constructor
24     z = x;         z.print();    // Default assignment operator
25 }
```

# 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

```
1  #include <iostream>      /* File: default-assign-problem2.cpp */
2  #include <cstring>
3  using namespace std;
4  class Word
5  {
6      private: int frequency; char* str;
7      void set(int f, const char* s)
8          { frequency = f; str = new char [strlen(s)+1]; strcpy(str, s); }
9      public:
10         Word(const char* s, int k = 1)
11             { set(k, s); cout << "\nImplicit const char* conversion\n"; }
12         Word(const Word& w) { set(w.frequency, w.str); cout << "\nCopy\n"; }
13
14         void print() const // Also prints the address of object's str array
15             { cout << str << " : " << frequency << " ; "
16               << reinterpret_cast<void*>(str) << endl; }
17 };
18
19 int main()
20 {
21     Word x("rat");      x.print(); // Conversion constructor
22     Word y = x;          y.print(); // Copy constructor
23     Word z("cat", 2);    z.print(); // Conversion constructor
24     z = x;               z.print(); // Default assignment operator
25 }
```

Implicit const char\* conversion

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

Copy

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

Implicit const char\* conversion

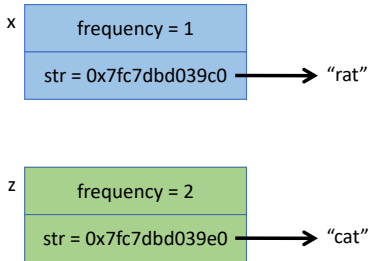
```
cat : 2 ; 0x7fc7dbd039e0
```

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

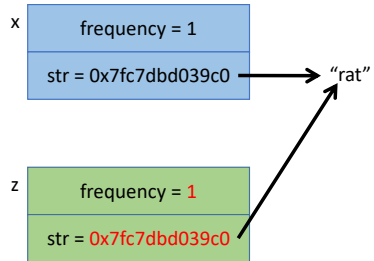


# Problem With Default Memberwise Assignment

Before  $z = x$



After  $z = x$



# Quiz: Constructors

Which constructor is called in the following statements?

- ➊ `Word nothing;`
- ➋ `Word dream_grade('A');`
- ➌ `Word major { "COMP" };`
- ➍ `Word hkust = "hkust";`
- ➎ `Word exchange_to(hkust);`
- ➏ `Word grade = dream_grade;`
- ➐ `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**.

```
1  class Word                                /* File: overload-constructor.cpp */
2  {
3      private:
4          int frequency;
5          char* str;
6
7      public:
8          Word();                            // Default constructor
9          Word(const char* s, int k = 1);    // Conversion constructor
10         Word(const Word& w);               // Copy constructor
11     };
```

# 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 “+”?

```
1  #include <iostream>          /* File: overload-function.cpp */
2  #include <cstring>
3  using namespace std;
4  class Word
5  {
6      private:
7          int frequency; char* str;
8      public:
9          void set() const { cout << "Input the string: "; cin >> str; } // Error!
10         void set(int k) { frequency = k; }
11         void set(char c) { str = new char [2]; str[0] = c; str[1] = '\0'; }
12         void set(const char* s) { str = new char [strlen(s)+1]; strcpy(str, s); }
13 };
14
15 int main()
16 {
17     Word movie;                // Which constructor?
18     movie.set();               // Which set function?
19 }
```

# 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 **{** .
  - **: *member<sub>1</sub>(expression<sub>1</sub>), member<sub>2</sub>(expression<sub>2</sub>), ...***
  - 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 ..

```
1  class Word                                /* File: mil-word.h */
2  {
3      private:
4          char lang;
5          int freq;
6          char* str;
7
8      public:
9          Word() : lang('E'), freq(0), str(nullptr) { };
10
11         /* Or, using the braced initialization syntax as follows
12            Word() : lang{'E'}, freq{0}, str{nullptr} { };
13         */
14
15         Word(const char* s, int f = 1, char g = 'E') : lang(g), freq(f)
16             { str = new char [strlen(s)+1]; strcpy(str, s); }
17
18         void print() const { cout << str << " : " << freq << endl; }
19     };
```

# 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!**

```
1  class Word_Pair          /* File: mil-word-pair.h */
2  {
3      private:
4          Word w1; Word w2;
5      public:
6          Word_Pair(const char* s1, const char* s2) : w1(s1,5), w2(s2) { }
7          void print() const
8          {
9              cout << "word1 = "; w1.print();
10             cout << "word2 = "; w2.print();
11         }
12     };
```

# Problem If Member\_INITIALIZER List Is Not Used

```
1  class Word_Pair  /* File: member-class-init-by-mil.h */
2  {
3      private:
4          Word w1; Word w2;
5      public:
6          Word_Pair(const char* s1, const char* s2) : w1(s1,5), w2(s2) { }
7  };
```

⇒ 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;`

```
1  #include <iostream>      /* File: mil-const-ref.cpp */
2  using namespace std;
3  int a = 5;
4
5  class Example
6  {
7      const int const_m = 3;
8      int& ref_m = a;
9  public:
10     Example() { }
11     Example(int c, int& r) : const_m(c), ref_m(r) { }
12     void print() const { cout << const_m << "\t" << ref_m << endl; }
13 };
14
15 int main()
16 {
17     Example x; x.print();
18     int b = 55; Example y(10, b); y.print();
19 }
```

# 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')

^

# Delegating Constructor vs. Private Utility Function

```
1  #include <iostream>          /* File: copy-constructor2.cpp */
2  #include <cstring>
3  using namespace std;
4
5  class Word
6  {
7      private:
8          int frequency; char* str;
9          void set(int f, const char* s) // Private utility function
10             { frequency = f; str = new char [strlen(s)+1]; strcpy(str,s); }
11      public:
12          Word(const char* s, int k = 1)
13             { set(k, s); cout << "conversion\n"; }
14          Word(const Word& w)
15             { set(w.frequency, w.str); cout << "copy\n"; }
16  };
```

- In this previous example, since most of the code of the conversion and copy constructors are similar, they are defined with a **private utility function** `set()`.
- May we achieve similar result without defining the latter?

# Example: Delegating Constructor (C++11)

```
1  #include <iostream>          /* File: delegating-constructor.cpp */
2  #include <cstring>
3  using namespace std;
4
5  class Word                    // Modified from copy-constructor.cpp
6  {
7  private:
8      int frequency; char* str;
9  public:
10     Word(const char* s, int f = 1)
11     {
12         frequency = f; str = new char [strlen(s)+1]; strcpy(str, s);
13         cout << "conversion" << endl;
14     }
15     Word(const Word& w) : Word(w.str, w.frequency) { cout << "copy" << endl; }
16     void print() const { cout << str << " : " << frequency << endl; }
17 };
18
19 int main()
20 {
21     Word movie("Titanic"); movie.print(); // which constructor?
22     Word song(movie); song.print();       // which constructor?
23     Word ship = movie; ship.print();      // which constructor?
24 }
```

# Delegating Constructor (C++11)

- In this example, the copy constructor, using the **member initializer list** syntax, **delegates** the conversion constructor to create an object.
- The copy constructor is now a **delegating constructor**.
- **Restriction**: the **delegated constructor** must be the **only** item in the **MIL**.



## Part III

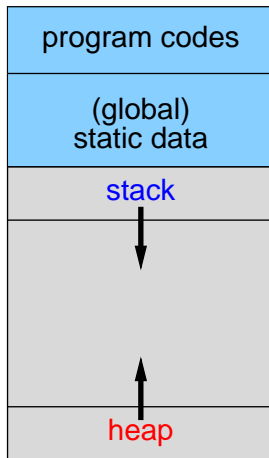
# Garbage Collection & Destructor



# Memory Layout of a Running Program

```
void f()
{
    // x, y are local variables
    // on the runtime stack
    int x = 4;
    Word y {"Titanic"};

    // p is another local variable
    // on the runtime stack.
    // But the array of 100 int's
    // that p points to
    // is on the heap
    int* p = new int [100];
}
```



# Memory Usage on the Runtime Stack and Heap

- **Local** variables are **constructed** (created) when they are defined in a function/block on the **run-time stack**.
- When the function/block terminates, the local variables inside and the call-by-value (CBV) arguments will be **deconstructed** (and removed) from the **run-time stack**.
- Both construction and destruction of variables are done **automatically** by the compiler by calling the appropriate **constructors** and **destructors**.
- **Dynamically** allocated memory **remains** after function/block terminates, and it is the user's responsibility to return it back to the **heap** for recycling using **delete**; otherwise, it will stay until the program finishes.
- **Garbage** is a piece of storage that is part of a running program but there are **no** more references to it.
- **Memory leak** occurs when there is **garbage**.

## 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

```
1 void Example() /* File: default-destructor-problem.cpp */
2 {
3     Word x("bug", 4);
4     ...
5 }
6
7 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.

```
1  #include <cstring>          /* File: destructor.cpp */
2  class Word
3  {
4      private:
5          int frequency; char* str;
6      public:
7          Word() : frequency(0), str(nullptr) { };
8          Word(const char* s, int k = 0): frequency(k)
9              { str = new char [strlen(s)+1]; strcpy(str, s); }
10         ~Word() { delete [] str; }
11 };
12 int main()
13 {
14     Word* p = new Word {"Titanic"};
15     Word* x = new Word [5];
16     delete p;                // Destruct a single object
17     delete [] x;             // Destruct an array of objects
18 }
```

# 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:

- ➊ **default constructor**  
(but only if you don't define other constructors)
- ➋ default **copy constructor**
- ➌ default **(copy) assignment operator** function
- ➍ default **move constructor** (C++11 )
- ➎ default **move assignment operator** function (C++11 )
- ➏ **default destructor**

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

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



## Example: = default; = delete;

```
1  #include <iostream>      /* File: default-delete.cpp */
2  #include <cstring>
3  using namespace std;
4  class Word
5  {
6  private:
7      int frequency {0}; char* str {nullptr};
8  public:
9      Word() = default; // Still want the simple default constructor
10     Word(const Word& w) = delete; // Words can't be copied
11     Word(const char* s, int k) : frequency(k)
12         { str = new char [strlen(s)+1]; strcpy(str, s); }
13     void print() const
14         { cout << ((str == nullptr) ? "not-a-word" : str)
15             << " : " << frequency << endl; }
16 };
17 int main()
18 {
19     Word x; x.print();
20     Word y("good", 3); y.print();
21     Word z(y); // Error: call to deleted constructor of 'Word'
22 }
```

## 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**.

```
1  /* File: example-has.h */
2  class B { ... };
3
4  class A
5  {
6      private:
7          B my_b;
8
9      public:
10         // Declaration of public members or functions
11     };
```

# 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 postoffice.cpp */
using namespace std;
#include "postoffice.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 initializer 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"; }
};
```

```
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; }
};
```

```
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"; }
};
```

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"; }
};
```

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"; }
};
```

# 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

```
1  #include <iostream>      /* file: default-member-init.cpp */
2  using namespace std;
3
4  class A
5  {
6      int a;
7  public:
8      A(int z) : a(z) { cout << "call A's constructor: " << a << endl; }
9      ~A() { cout << "call A's destructor: " << a << endl; }
10     int get() const { return a; }
11 };
12
13 class B
14 {
15     int b1 = 999;          // Remember: can't initialize by ( )
16     A b2 = 10;             // Call A's conversion constructor
17     A b3 {100};            // Call A's conversion constructor
18 public:
19     B() { cout << "call B's default constructor" << endl << endl; }
20     ~B() { cout << "call B's destructor: " << b1 << "\t"
21           << b2.get() << "\t" << b3.get() << endl; }
22 };
23
24 int main() { B x; return 0; }
```

- When an object is **constructed**, its data members are **constructed first**.
- When the object is **deconstructed**, the data members are **deconstructed after** the **destructor** code of the object has been executed.
- When object A **owns** other objects, remember to destruct 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**”.