Object-Oriented Programming and Data Structures

COMP2012: Object Initialization, Construction and Destruction

Prof. Brian Mak
Prof. C. K. Tang

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, 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, "Titantic"}; }
```

- What happens if some of data members are private?

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

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

```
private-member-init.cpp:9:20: error: non-aggregate type
        'Word' cannot be initialized with an initializer list
int main( ) { Word movie = {1, "Titantic"}; }
                   ^       ~~~~~~~~~~~~~~~~~
```

Part I

Constructors

Different Types of C++ Constructors

		blank CD	default constructor
		MP3 to CD	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

- C++ supports a more general mechanism for user-defined initialization of class objects through constructor member functions.

```
Word movie;                                  // Default constructor
Word director = "James Cameron";             // Implicit conversion constructor
Word sci_fi("Avatar");                       // 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 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 = 0; }   // 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

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

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

- If there are not any user-defined constructors in the definition of class X, the compiler will generate the following default constructor for it,

  X::X( ) { }

- 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( ){ }.

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

int main( ) { Word movie; }   // which constructor?
```

```
default-constructor-bug.cpp:7:20: error: no matching constructor for
    initialization of 'Word'
int main( ) { Word movie; }                    // which constructor?
                   ^
default-constructor-bug.cpp:4:11: note: candidate constructor not viable:
      requires at least argument 's', but no arguments were provided
  public: Word(const char* s, int k = 0);
          ^
default-constructor-bug.cpp:1:7: note: candidate constructor (the implicit
      copy constructor) not viable: requires 1 argument, but 0 were 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

Implicit Conversion Constructor(s) ..

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

```cpp
#include <cstring>          /* File: explicit-conversion-constructor.cpp */
class Word {
  private: int frequency; char* str;
  public:
    explicit Word(const char* s)
        { frequency = 1; 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";             // Bug: implicit conversion
}
```

```
explicit-conversion-constructor.cpp:12:10:
    error: no viable conversion from 'const char [14]' to 'Word'
    Word director = "James Cameron"; // Implicit conversion
         ^          ~~~~~~~~~~~~~~~~
explicit-conversion-constructor.cpp:2:7: note: candidate constructor
    (the implicit copy constructor) not viable: no known conversion from
    'const char [14]' to 'const Word &' for 1st argument
```

Copy Constructor

```
#include <iostream>                    /* File: copy-constructor.cpp */
#include <cstring>
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); std::cout << "conversion\n"; }
    Word(const Word& w)
        { set(w.frequency, w.str); std::cout << "copy\n"; }
};

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

Copy Constructor: X::X(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:

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

  Word x("Brian"); Word y = x;

- object returned by a function by value

```
#include <iostream>                                    /* File: return-by-value.cpp */
#include <cstring>
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); std::cout << "conversion\n"; }
    Word(const Word& w)
        { set(w.frequency, w.str); std::cout << "copy\n"; }
    Word to_upper_case( ) const {
            Word x(*this);
            for (char* p = x.str; *p != '\0'; p++) *p += 'A' - 'a';
            return x;
        }
    void print( ) const
        { std::cout << str << " : " << frequency << std::endl; }
};
int main( ) {
    Word movie("titanic"); movie.print( );
    Word song = movie.to_upper_case( ); song.print( );
}
```

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, the compiler will automatically supply a default copy constructor for it,

  X(const X&) { // memberwise assignment }

- ⇒ memberwise assignment (aka copy assignment; memberwise copy). Conceptually, it does the following

  ```
  song.frequency = movie.frequency;
  song.str = movie.str;
  ```

- It works even for array members.

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.
- Similar to the case of copy constructor: if you don't write your own copy constructor, the compiler will provide you the default copy constructor — again, memberwise assignment.
- 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-problem2.cpp */
#include <cstring>
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(1, s); std::cout << "\nImplicit const char* conversion\n"; }
    Word(const Word& w)
        { set(w.frequency, w.str); std::cout << "Copy\n"; }
    void print( ) const
        { std::cout << str << "  :  " << frequency << "  ;  "
                    << reinterpret_cast<const void*>(str) << std::endl; }
};

void print_word(const Word& x) { x.print( ); }
int main( )
{
    Word x("sky"); print_word(x);                    // Conversion constructor
    Word y = x; print_word(y);                       // Default copy constructor
    Word z("bug"); print_word(z);                    // Conversion constructor
    z = x; print_word(z);                            // Default assignment operator
}
```

Default Memberwise Assignment With Array Data ..

```
Implicit const char* conversion
sky : 1 ; 0x7fff5cd2e5d4
Copy
sky : 1 ; 0x7fff5cd2e56c

Implicit const char* conversion
bug : 1 ; 0x7fff5cd2e504
sky : 1 ; 0x7fff5cd2e504
```

Default Memberwise Assignment With Pointer Data

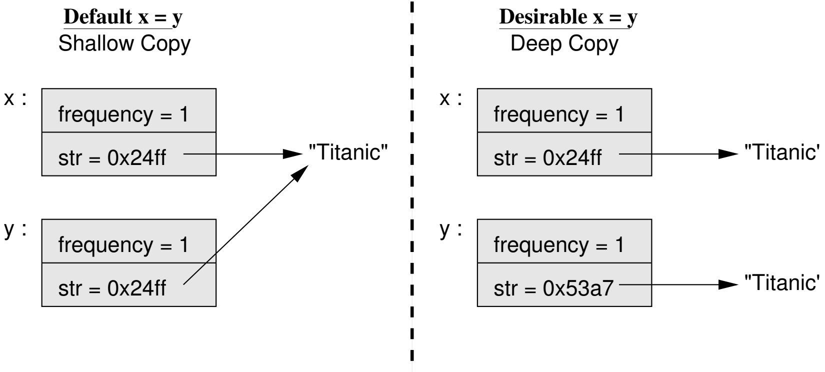
```
#include <iostream>                     /* File: default-assign-problem.cpp */
#include <cstring>
class Word {
  private: int frequency; char* str;
    void set(int f, const char* s)
        { frequency = f; str = new char [strlen(s)+1]; strcpy(str, s); }
  public:
    Word(const char* s, int k =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

void print_word(const Word& x) { x.print( ); }
int main( )
{
    Word x("sky"); print_word(x);              // Conversion constructor
    Word y = x; print_word(y);                 // Default copy constructor
    Word z("bug"); print_word(z);              // Conversion constructor
    z = x; print_word(z);                      // Default assignment operator
}
```

```
Implicit const char* conversion
sky : 1 ; 0x7fc7dbd039c0
Copy
sky : 1 ; 0x7fc7dbd039d0

Implicit const char* conversion
bug : 1 ; 0x7fc7dbd039e0
sky : 1 ; 0x7fc7dbd039c0
```

Problem With Default Memberwise Assignment



Constructor: Quiz

How are class initializations done 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;

Review: 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
};
```

- 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 */
class Word {
  private:
    int frequency; char* str;
  public:
    void set( ) const { std::cout << str; }   // Bad overloading! Obscure!
    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?
}
```

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 arguments.

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

- All arguments without default values must be declared to the left of default arguments. Thus, the following is an error:

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

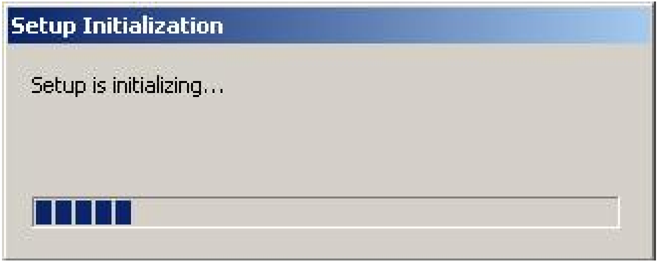
- An argument can have its default initializer specified only once in a file, usually in the public header file, and not in the function definition. Thus, the following is an error.

```
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 Initialization List



Member Initialization List

- 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 initialization list by calling their own constructors.
  - It starts after the constructor header but before the opening { .
  - : $member_1(expression_1)$, $member_2(expression_2)$, ...
  - The order of the members in the list doesn't matter; the actual execution order is their order in the class declaration.

```
class Word                                          /* File: mil-word.h */
{
  private:
    char lang; int freq; char* str;
  public:
    Word( ) : lang('E'), freq(0), str(NULL) { };
    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 { std::cout << str << " : " << freq << "\n"; }
};
```

- Since the member initialization list calls the constructors of the data member, it works well for data members of user-defined types.
- It is better to perform initialization by member initialization list than by assignments.
- 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 {
        std::cout << "word1 = "; w1.print( );
        std::cout << "word2 = "; w2.print( );
    }
};
```

```
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) { }
};
```

⇒ word1/word2 are initialized using the conversion constructor, Word(const char*, int = 1, char = 'E').

```
Word_Pair(const char* x, const char* y) { word1 = x; word2 = y; }
```

⇒ error-prone because word1/word2 are initialized by assignment. If there is no user-defined assignment operator function, the default memberwise assignment may not be good enough when there are pointer data members.

- const or reference members must be initialized using member initialization list.
- c.f. `float y; float& z = y; const int x = 123;`

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

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

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

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

- It cannot be done using default arguments.

```
#include <iostream>          /* File: mil-const-member-error.cpp */
class Word
{
  private:
    const char lang; int freq; char* str;
  public:
    Word( ) : lang('E'), freq(0), str(NULL) { };
    Word(const char* s, int f = 1, char g = 'E')
        { str = new char [strlen(s)+1]; strcpy(str, s); }
    void print( ) const
        { std::cout << str << " : " << freq << " ; " << "\n"; }
};

int main( ) { Word("hkust"); }
```

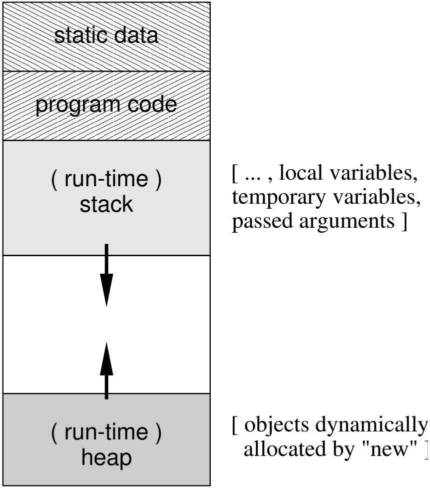
```
mil-const-member-error.cpp:8:5: error: constructor for 'Word' must
explicitly initialize the const member 'lang'
    Word(const char* s, int f = 1, char g = 'E')
    ^
```

Part III

Garbage Collection & Destructor

```
void f( )          /* File: var.cpp */
{
    // 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
    // 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 destructed (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; otherwise, it will stay until the program finishes.

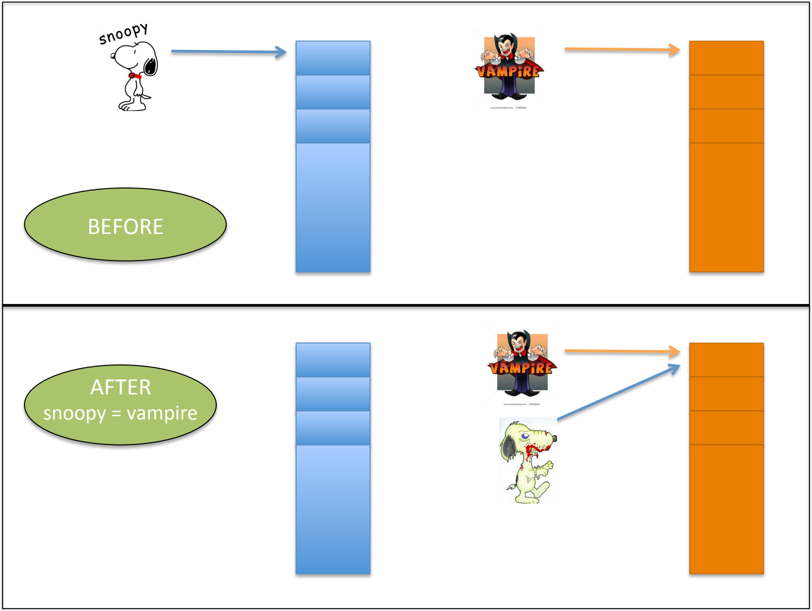
Garbage and Memory Leak

- 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.

```
int main( )                                    /* File: memory-leak.cpp */
{
    for (int j = 1; j <= 10000; j++)
    {
        int* snoopy = new int [100];
        int* vampire = new int [100];
        snoopy = vampire;                      // Now snoopy becomes vampire
        ...                                    // Where is the old snoopy?
    }
}
```

Question: What happens if garbages are huge or continuously created inside a big loop?!

Example: Before and After snoopy = vampire



```
main( )                                          /* File: delete.cpp */
{
    Stack* p = new Stack(9);        // A dynamically allocated stack object
    int* q = new int [100];     // A dynamically allocated array of integers
    delete p;                                        // delete an object
    delete [ ] q;                               // delete an array of objects
    p = 0;                   // It is a good practice to set a pointer to NULL
    q = 0;                               // when it is not pointing to anything
}
```

- Explicitly remove a single object you don't need anymore by calling delete on a pointer to the object.
- Explicitly remove an array of objects by calling delete [ ] on a pointer to the first object of the array.
- Notice that delete only puts the dynamically allocated memory back to the heap, and the local variables (p and q above) stay behind on the run-time stack until the objects go out of scope (e.g., the enclosing block/function terminates).

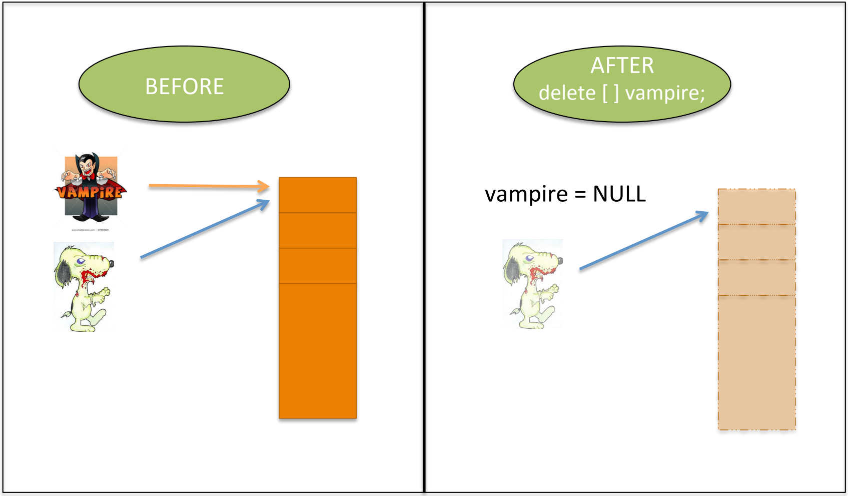
Dangling References and Pointers

```
main( )                                    /* File: dangling-reference.cpp */
{
    char* p;
    char* q = new char [128];          // Dynamically allocate a char buffer
    p = q;                        // p and q now points to the same char buffer
    delete [ ] q; q = 0;                             // delete the char buffer

    /* Now p is a dangling pointer! */
    p[0] = 'a';                         // Error: possibly segmentation fault
    delete [ ] p;                       // Error: possibly segmentation fault
}
```

- Careless use of delete may cause dangling references.
- A dangling reference is created when memory pointed by a pointer is deleted but the user thinks that the address is still valid.
- Dangling references are due to carelessness and pointer aliasing — an object is pointed to by more than one pointer.

Example: Dangling References



Other Solutions: Garbage, Dangling References

- Garbage and dangling references are due to careless pointer manipulation and pointer aliasing.
- Some languages provide automatic garbage collection facility which
  - stops a program from running from time to time,
  - checks for garbages,
  - and puts them back to the heap for recycling.
- Some languages do not have pointers at all!
  (It was said that most program bugs are due to pointers.)

Destructor

Destructor X::~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 for a class,

  X::~X( ) { }

  which simply releases the storage of the object's data members.

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.

```
class Word {                                    /* File: destructor.cpp */
  private:
    int frequency; char* str;
  public:
    Word( ) : frequency(0), str(0) { };
    Word(const char* s, int k = 0) { ... }
    ~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

```
void Bug(Word& x)                        /* File: default-assign-bug.cpp */
{
    Word bug("bug", 4);
    x = bug;
}

int main( )
{
    Word movie("Titanic");               // Which constructor?
    Bug(movie);                          // Which constructor?
}
```

Question: How many bugs are there?

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
};
```

```
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
```

- 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; but, the `Clock` destructor is called after the `Postoffice` destructor.
- 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"; }
};
```

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

- 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.

```
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 in which they appear in the `Postoffice` class.

Cons/Destruction Order: Postoffice Moves Clock to Room

```cpp
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

```cpp
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"; }
};
```

```
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.

Summary

- When an object is constructed, its data members are constructed first.
- When the object is destructed, the data members are destructed after the destructor 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 initialization list — the "colon syntax".