Comp151

Construction & Initialization

Class Object Initialization

If ALL data members of the class are public, they can be initialized when they are created as follows:

```
class Word
{
  public:
    int frequency;
    char* str;
};

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

Class Object Initialization ..

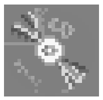
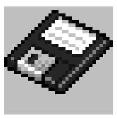
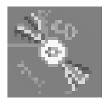
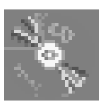
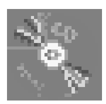
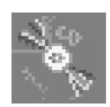
What happens if some of data members are private?

```
class Word
{
    char* str;
  public:
    int frequency;
};

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

```
Error: a.cc:8: 'movie' must be initialized by
               constructor, not by '{...}'
```

Constructor: Introduction

 *blank CD*	default constructor
 →  *MD to CD*	conversion constructor
 →  盗 版 *pirated CD*	copy constructor
All4One's "I Swear" Eric Clapton's "Tears in Heaven" Beethoven's Symphony No. 5 Phantom of the Opera } 	other constructors

C++ Constructors

- C++ supports a more general mechanism for user-defined initialization of class objects through *constructor member functions*.
  - Word movie;
  - Word director = "James Cameron";
  - Word movie = Word("Titanic");
  - Word *p = **new** Word("action", 1);

- Syntactically, a constructor of a class is a special member function having the *same* name as the class.
- A constructor is called whenever an object of a class is created.
- A constructor must **NOT** specify a return type or explicitly returns a value — NOT even the **void** type.

Default Constructor


```
class Word
{
    int frequency;
    char* str;
  public:
    Word() { frequncey = 0; str = 0; }
};

int main(int argc, char** argv)
{
    Word movie;
}
```

- A *default constructor* is a constructor that is called with NO argument: X::X() for class X.
- It is used to initialize an object with user-defined default values.

Compiler Generates a Default Constructor

```
class Word
{
    int frequency;
    char* str;
};

int main(int argc, char** argv)
{
    Word movie;
}
```

- If there are **NO** user-defined constructors, the compiler will generate the default constructor: X::X() for class X for you.
- Word() { } only creates a record with space for an **int** quantity and a **char*** quantity. Their initial values CAN**NOT** be trusted.

Default Constructor: Bug

BUT: Only when there are **NO** user-defined constructors, will the compiler automatically supply the default constructor.

```
class Word
{
    ...
    public: Word(const char* s, int k = 0);
};

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

```
a.cc:16: no matching function for call to ‘Word::Word ()
a.cc:12: candidates are: Word::Word(const Word &)
a.cc:7:                  Word::Word(const char *, int)
```

Type Conversion Constructor


```
class Word {
    ...
  public:
    Word(const char* s)
    {
        frequency = 1;
        str = new char [strlen(s)+1]; strcpy(str, s);
    }
};

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

- A constructor accepting a single argument specifies a conversion from its argument type to the type of its class: Word(**const char***) converts from type **const char*** to type Word.

Type Conversion Constructor ..

```
class Word {
    ...
  public:
    Word(const char* s, int k = 1) {
        frequency = k;
        str = new char [strlen(s)+1]; strcpy(str, s);
    }
};

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

- Notice that if all but ONE argument of a constructor have default values, it is still considered a conversion constructor.

Copy Constructor: Example

```
class Word
{
  public:
    Word(const char* s, int k = 1);
    Word(const Word& w)
    {
        frequency = w.frequency;
        str = new char [strlen(w.str)+1];
        strcpy(str, w.str);
    }
};

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

Copy Constructor

- Copy constructor has only ONE argument of the same class.
- Syntax: X(const X&) for class X.
- It is called upon:
  - parameter passing to a function (call-by-value)
  - initialization assignment: Word x("Brian"); Word y = x;
  - value returned by a function:

```
Word Word::to_upper_case()
{
    Word x(*this);
    for (char* p = x.str; *p != '\0'; p++)
        *p += 'A' - 'a';

    return x;
}
```

Default Copy Constructor

For a class X, if no copy constructor is defined by the user, the compiler will automatically supply: X(const X&).

```
class Word {
  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?
}
```

⇒ memberwise copy

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

Constructor: Quiz

Quiz: How are class initializations done in the following statements:

- Word vowel(“a”);
- Word article = vowel;
- Word movie = “Titanic”;

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
{
    int frequency;
    char* str;

  public:
    Word() {};
    Word(const char* s, int k = 1);
    Word(const Word& w);
};
```

Function Overloading ..

- In general, function names can be overloaded in C++.

```
class Word
{
    ...
    set(int k) { frequency = k; }
    set(const char* s) { str = new char [strlen(s)+1]; strcpy(str, s); }
    set(char c) { str = new char [2]; str[0] = c; str[1] = '\0'; }
    set() { cout << str; }   // Bad overloading! Obscure understanding
};
```

- Actually, operators are often overloaded.
  e.g. What is the type of the operands for "+"?

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.

```
class Word {
    ...
  public:
    Word(const char* s, int k = 1)
    {
        frequency = k;
        str = new char [strlen(s)+1]; strcpy(str, s);
    }
};

int main() {
    Word movie("Titanic");
    Word director("Steven Spielberg", 20);
}
```

Default Arguments ..

- There may be more than one default arguments.

  void download(**char*** prog, **char** os = LINUX, **char** format = ZIP);

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

  void download(**char** os = LINUX, **char*** prog, **char** format = ZIP);

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

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

Member Initialization List

Most of the class members may be initialized inside the body of constructors or through member initialization list as follows:

```
class Word
{
    int frequency;
    char* str;

  public:
    Word(const char* s, int k = 1) : frequency(k)
    {
        str = new char [strlen(s)+1]; strcpy(str, s);
    }
};
```

Member Initialization List ..

Member initialization list also works for data members which are user-defined class objects.

```
class Word_Pair
{
    Word w1;
    Word w2;

  public:
    Word_Pair(const char* s1, const char* s2) : w1(s1), w2(s2) { }
};
```

But make sure that the corresponding member constructor exist!

Initialization of const or & Members

const or reference members can ONLY be initialized through member initialization list.

```
class Word
{
    const char language;
    int frequency;
    char* str;

  public:
    Word(const char* s1, int k = 1) : language('E'), frequency(k)
    {
        str = new char [strlen(s)+1]; strcpy(str, s);

    }
};
```

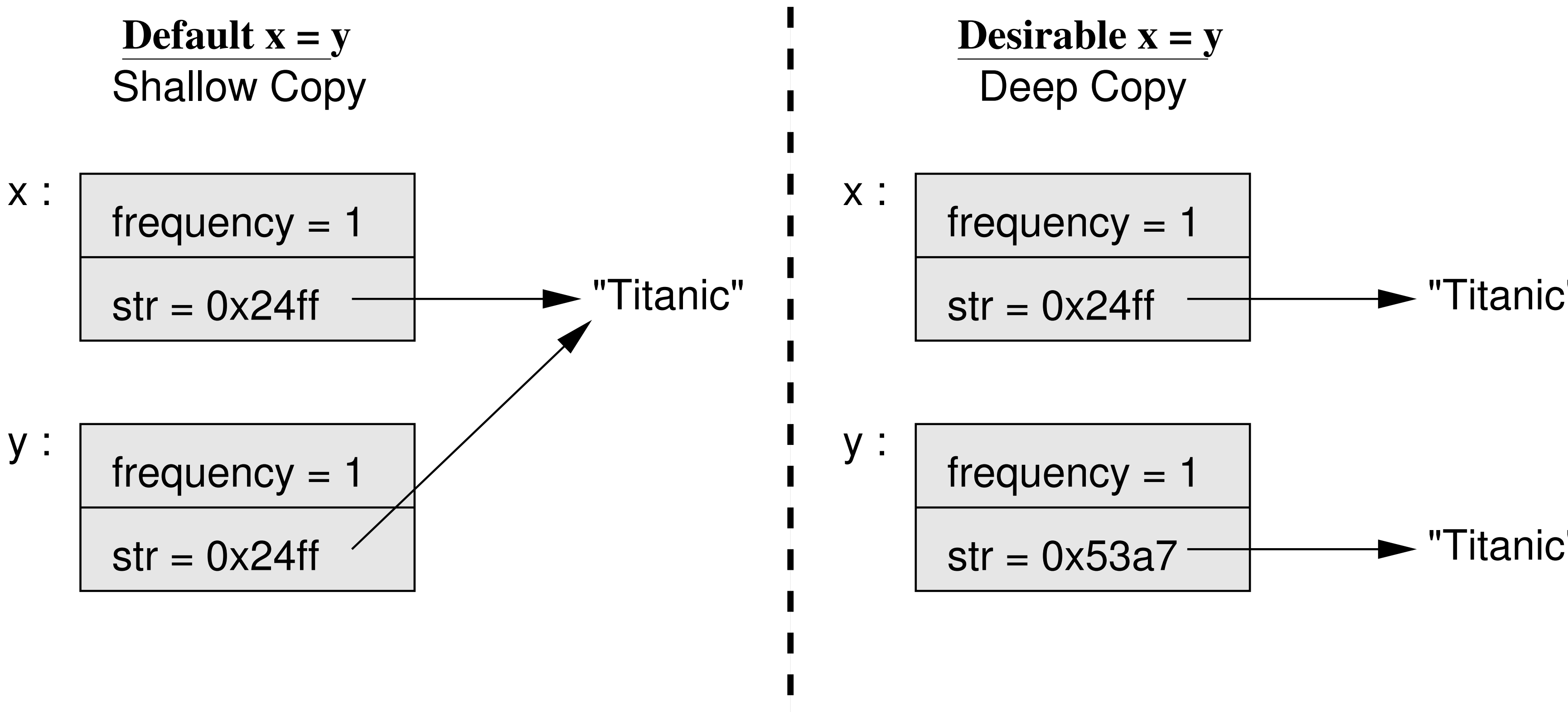
Default Memberwise Assignment

Word x("Titanic", 1);	// *Word(const char*, int) constructor*
Word y;	// *Word() constructor*
y = x;	// *default memberwise assignment*

```
=>      y.frequency = x.frequency
        y.str = x.str
```

- If an assignment operator function is NOT supplied (through operator overloading) , the compiler will provide the *default* assignment function — *memberwise assignment*.
- c.f. The case of copy constructor: if you DON'T write your own copy constructor, the compiler will provide the *default* copy constructor — memberwise copy.
- Memberwise assignment/copy does NOT work whenever memory allocation is required for the class members.

Default Memberwise Assignment ..



Member Class Initialization

Class members should be initialized through member initialization list which calls the appropriate constructors than by assignments.

```
class Word_Pair
{
    Word word1;
    Word word2;
    Word_Pair(const char* x, const char* y): word1(x), word2(y) { }
};
```

⇒ word1/word2 are initialized using the type conversion constructor, Word(const char*).

```
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 do what is required.

Summary

- Constructors: default, conversion, copy
- Constructor overloading
- Default arguments
- Member initialization list
- The important relationship between
  - Assignment of class objects, and
  - default/copy constructor

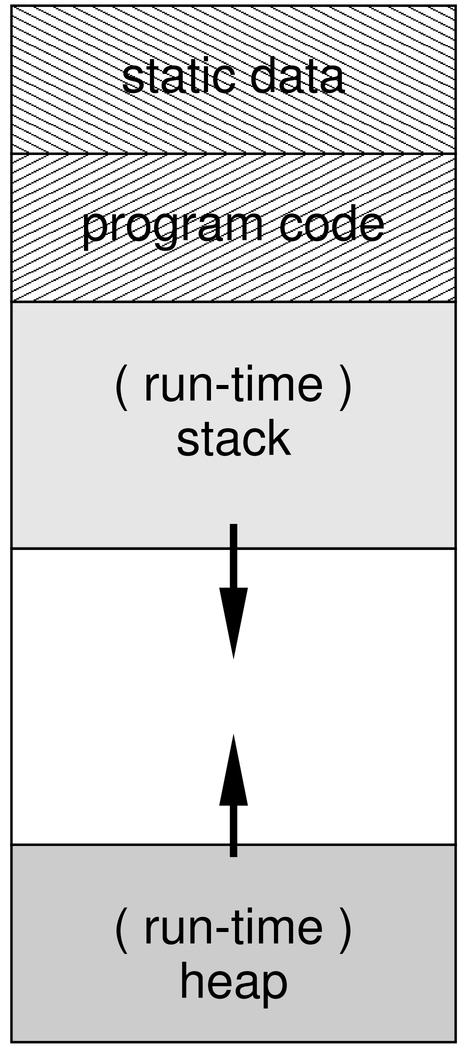
Comp151

Garbage Collection & Destructors

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
    // that p points to
    // is on the heap
    int* p = new int [100];
}
```



[... , local variables, temporary variables, passed arguments]

[objects dynamically allocated by "new"]

Memory Usage on 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 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.
- BUT, 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.

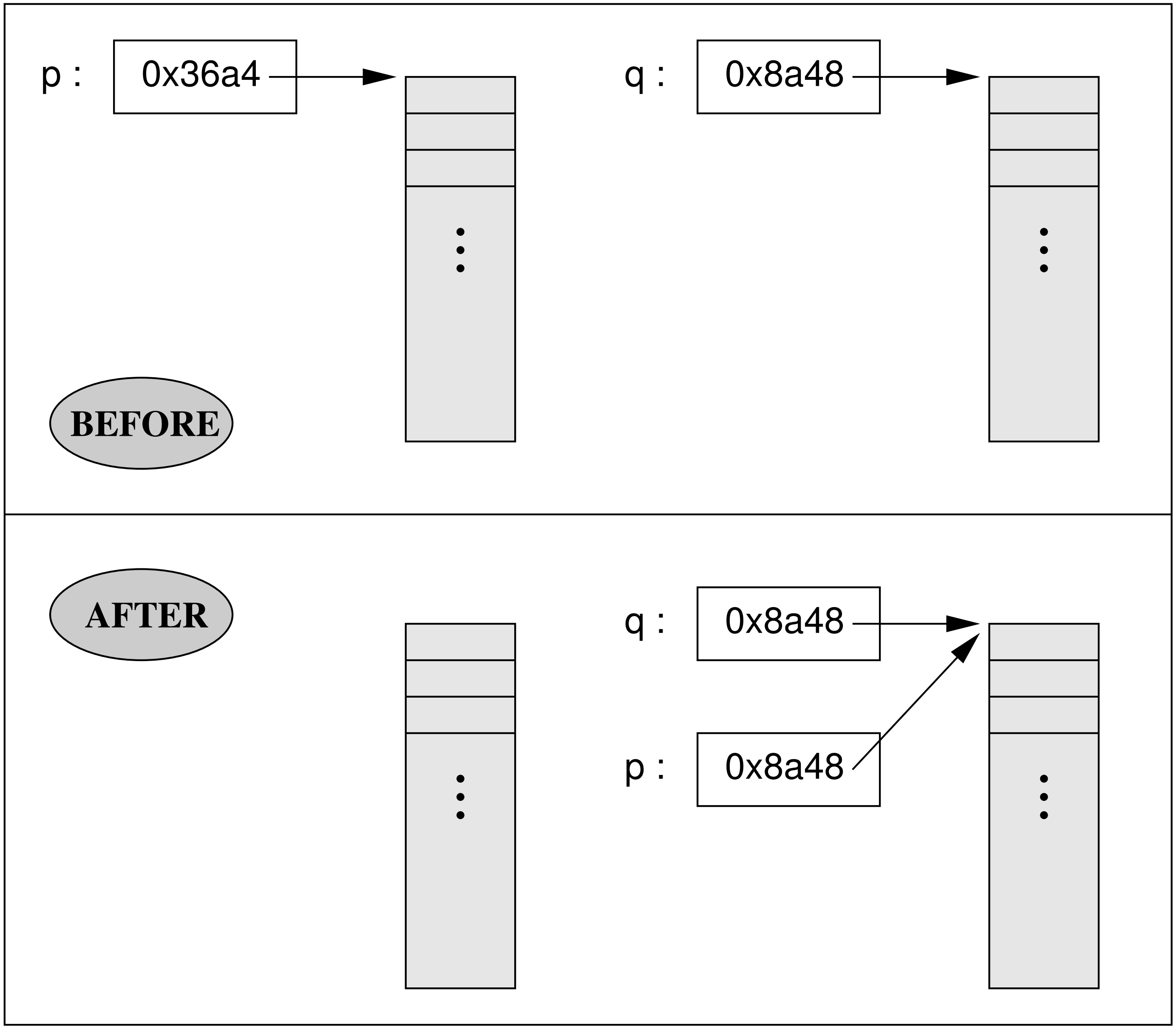
Garbage and Memory Leak


```
main()
{
    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?
    }
}
```

- Garbage is a piece of storage that is part of a program but there are no more references to it in the program.
- Memory Leak occurs when there is garbage.

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

Example: Before and After p = q



delete: To Remove Garbage

```
main() {
    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 garbage object by calling **delete** on a pointer to the object.
- Explicitly remove an array of garbage 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 function terminates.

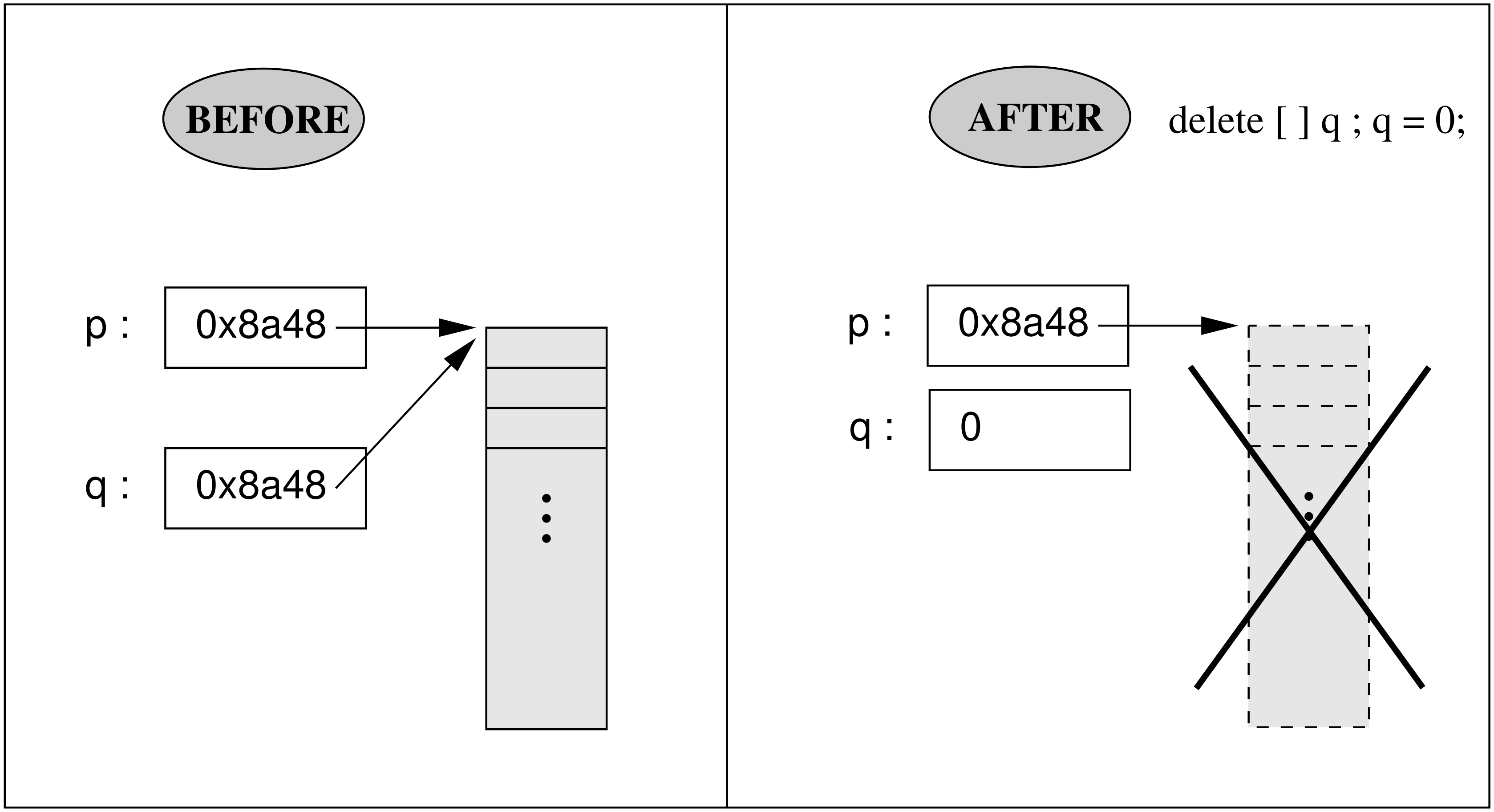
Dangling References and Pointers

However, careless use of **delete** may cause dangling references.

```
main()
{
    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
}
```

- 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.)

Destructors: Introduction

```
void Example()
{
    Word x("bug", 4);
    ...
}

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

- On return from Example(), the local Word object "x" of Example() is destroyed from the run-time stack of Example(). i.e. the memory space of (int) x.frequency and (char*) x.str are released.

Quiz: How about the dynamically allocated memory for the string, "bug" that x.str points to?

Destructors

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

~Word() { **delete** [ ] str; }

- A *destructor* of a class X is a special member function with the name X::~X( ).
- A destructor takes no arguments, and has no return type — thus, there can only be ONE destructor for a class.
- The destructor of a class is invoked automatically whenever its object goes out of scope — out of a function/block.
- If not defined, the compiler will generate a default destructor of the form X::~X( ) { } which does nothing.

Example: Destructors

```
class Word {
    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;                          // destroy a single object
    delete [ ] x;                      // destroy an array of objects
}
```

Bug: Default Assignment

```
void Bug(Word& x)
{
    Word bug("bug", 4);
    x = bug;
}

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

Quiz: What is movie.str after returning from the call Bug(movie)?

What you should take from this lecture

- The causes of garbages and memory leak
- How to define destructor function

Comp151

Order of
Construction & Destruction

"Has" relationship

When an object A has an object B as a data member, we say that

"A has a B."

```
class B { ... };

class A
{
      B my_b;

   public:
      // some public members or functions
};
```

- It is easy to see which objects *have* other objects. All you need to do is to look at the class definition.

Example: Order of Constructions


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

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

```
#include <iostream.h>
#include "postoffice.h"
int main() {
    cout << "Beginning of main\n";
    Postoffice x;
    cout << "End of main\n";
    return 0;
}
```

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

Order of Constructions: Remarks

- 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.

Order of Constructions with Owned Objects

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

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

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

Order of Constr. with Owned Objects: Remarks

- Now the `Postoffice` *owns* the `Clock`.
- 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.

Order of Constr. with Owned Objects: Fix


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

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

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

Order of Constructions w/ Multiple Objects


```
class Clock
{
    int HHMM;
  public:
    Clock() : HHMM(0) { cout << "Constructor Clock\n"; }
    Clock(int hhmm) : HHMM(hhmm)
        { cout<<"Constructor Clock at "<< HHMM <<endl; }
    ~Clock() { cout << "Destructor Clock at " << HHMM << endl; }
};

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

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

```
Beginning of main
Constructor Clock
Constructor Room
Constructor Postoffice
End of main
Destructor Postoffice
Destructor Room
Destructor Clock at 0
```

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

Order of Construction w/ Nested Objects

Let's move the clock to the room.

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

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

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

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

Order of Constructions w/ Temporary Objects

<pre>class Clock
{
    int HHMM;
  public:
    Clock() : HHMM(0) { cout << "Constructor Clock\n"; }
    Clock(int hhmm) : HHMM(hhmm)
        { cout<<"Constructor Clock at "<< HHMM <<endl; }
    ~Clock() { cout << "Destructor Clock\n"; }
};
 	
class Postoffice
{
    Clock clock;
  public:
    Postoffice()
        { clock = Clock(800); cout << "Constructor Postoffice\n"; }
    ~Postoffice(){ cout << "Destructor Postoffice\n"; }
};</pre>
	<pre>Beginning of main
	Constructor Clock
	Constructor Clock at 800
	Destructor Clock
	Constructor Postoffice
	End of main
	Destructor Postoffice
	Destructor Clock</pre>

- 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 for 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."