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 ‘{...}’
```

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 once 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);

    }
};
```

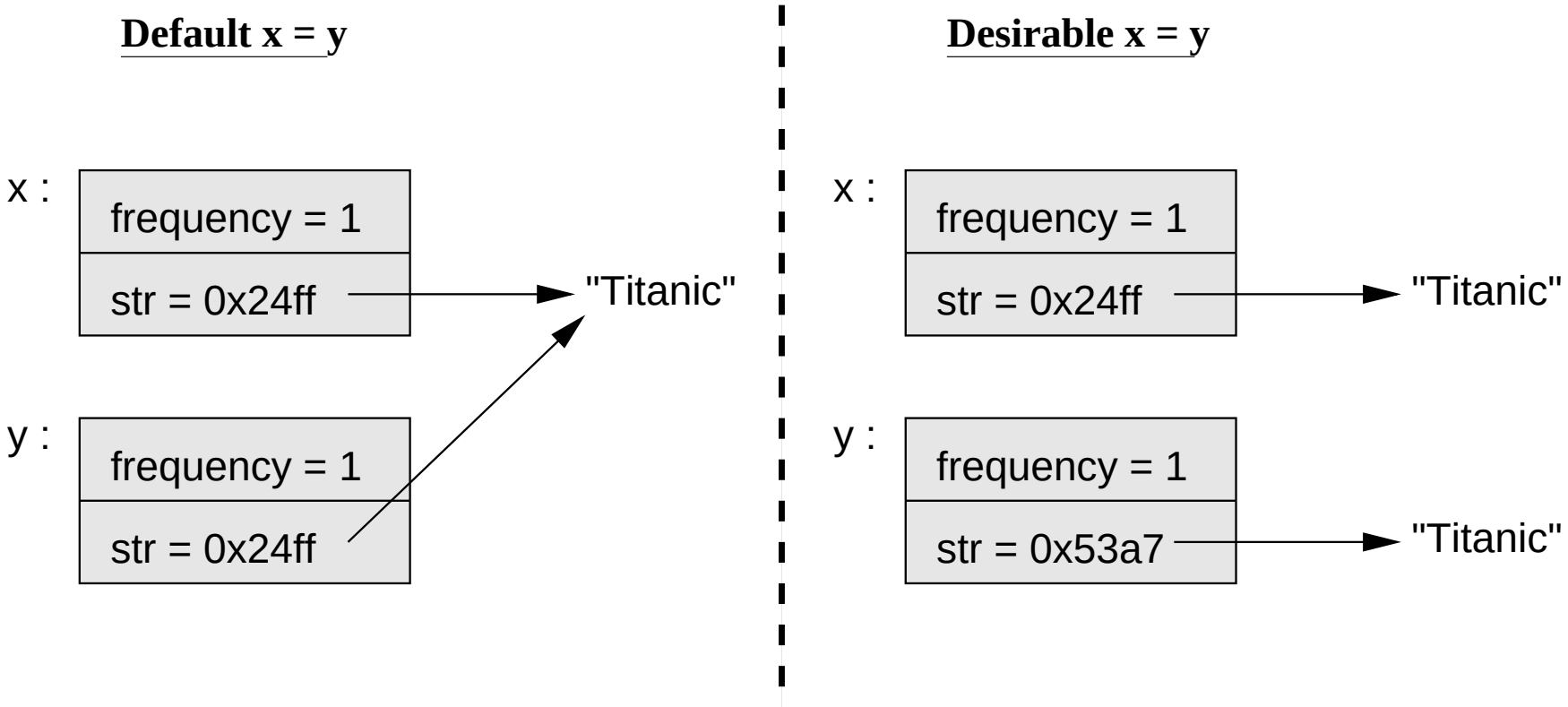
Default Memberwise Assignment


```
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 compile 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.