Comp151

Overloading, Construction & Initialization
Introduction

• Our next major topic will be how to initialize new objects using **constructors**. Before doing so we take a short break to introduce another concept that we will need in that discussion, that of function **overloading**. This is a technique that allows the same function name to have many “meanings”.

• In ordinary life, you actually use overloading all the time. E.g., 1 + 2 is not the same thing as 1.0 + 2.0 in C++; the + operator is overloaded.

• As another example suppose you want to write one function to compute the average of two numbers and another to compute the average of three numbers:

  ```
  double avg(double n1, double n2) {
    return ((n1 + n2) / 2.0);
  }
  double avg3(double n1, double n2, double n3) {
    return ((n1 + n2 + n3) / 3.0);
  }
  ```

• In C++, you can use the same name for both functions!
Introduction

• This is legal in C++ (but not in C):

```cpp
double avg(double n1, double n2) {
    return ((n1 + n2) / 2.0);
}

double avg(double n1, double n2, double n3) {
    return ((n1 + n2 + n3) / 3.0);
}
```
Function Overloading

- **Overloading** allows programmers to use the same name for functions that do *similar* things but with different input arguments.
- In general, both ordinary function names and member function names can be overloaded in C++.

```cpp
class Word {
public:
    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'; }
private:
    int frequency;
    char* str;
};
```
Function Overloading..

• But to speak good C++, don’t abuse overloading. Make sure that your overloaded functions really do similar things.

```cpp
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! obscures understanding
};
```

• Actually, operators (which are also functions!) are often overloaded. E.g., what is the type of the operands for “+”? 
Function Overloading

- As we’ll see, constructors are often overloaded.

```cpp
class Word {
public:
    Word() { }
    Word(const char* s, int k = 1);
    Word(const Word& w);
private:
    int frequency;
    char* str;
};
```
Default Arguments

If a function shows some *default* behaviors most of the time, and some exceptional behaviors only *once in awhile*, specifying default arguments is a *better* option than using overloading.

```cpp
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("Brokeback Mountain");
    Word director("Ang Lee", 20);
}
```

In fact, this is also a kind of overloading. (Why?)
Default Arguments..

• There may be more than one default argument.

```c
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:

```c
void download( char os = LINUX, char prog, char format = ZIP ); // error
```

```c
int main() { download(LINUX, ‘x’); } // can’t tell how to interpret this!
```

• 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:

```c
// word.hpp

```c
class Word {
    public:
        Word(const char* s, int k = 1);

    Word(const char* s, int k = 1);  
}
```

```c
// word.cpp
#include "word.hpp"

Word::word(const char* s, int k = 1)   
{ 
    ...  
    } 
```

```c
```
```c````
Default Arguments..

- There may be more than one default argument.

```c
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:

```c
void download( char os = LINUX, char prog, char format = ZIP );  // error
int main() { download(LINUX, ‘x’); }  // can’t tell how to interpret this!
```

- 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 okay:

```c
// word.hpp

class Word {

public:

    Word(const char* s, int k = 1);  // ok
    Word(const char* s, int k = 1);  // ok

};

// word.cpp

#include “word.hpp”

Word::word(const char* s, int k) // ok
{
    ...
    ...
}
```
Summary: Overloading

• If you have two or more function definitions for the same function name that is called **overloading**.

• When you overload a function name the different definitions must have different numbers of formal parameters, or some formal parameters of different types.

• The compiler checks each function call and matches it with the particular function definition whose number and type of formal parameters matches.

• The use of the same name to mean different things is called **polymorphism** (Greek for "many forms").
  – Technically, the kind of polymorphism we’ve just seen is called **ad hoc polymorphism**.
  – We’ll see another kind of polymorphism when we discuss templates.
Class Object Initialization

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

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

int main() { Word movie = {1, "Brokeback Mountain"}; }
```
Class Object Initialization ...

- What happens if some of data members are private?

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

int main() { Word movie = {1, "Brokeback Mountain"}; }

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 = “Ang Lee”;
  - Word movie = Word(“Brokeback Mountain”);
  - 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 is created, even when the object is only created temporarily, e.g., as a local variable.

- A constructor must NOT specify a return type or explicitly returns a value—NOT even the void type.
Default Constructor

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

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

```cpp
struct Word {
    int frequency;
    char* str;
};

int main(int argc, char* argv[])
{
    Word movie; // which constructor called?
}
```

- 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 **CANNOT** be trusted.
```cpp
class Word { // identical meaning to the previous struct
public:
    int frequency;
    char* str;
};

int main(int argc, char* argv[])
{
    Word movie; // which constructor called?
}

• 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 CANNOT be trusted.
```
Default Constructor: Bug

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

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

int main()
{
  Word movie; // which constructor?
  Word song("Brokeback Mountain"); // 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)
```
Caution: Weird C++ Syntax

• The default constructor is a function with no parameters so you might think that it should actually be called using
  
  Word movie();

  the same way as any other function without parameters. This in not correct. A default constructor should be called as

  Word movie;

  without using the ().
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("Brokeback Mountain");
    Word director = "Ang Lee";
}

• 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("Brokeback Mountain");
    Word director = "Ang Lee";
}

• 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("Brokeback Mountain"); // which constructor?
    Word song(movie); // which constructor?
}
Copy Constructor

• A copy constructor has only ONE argument of the same class
• Syntax: \texttt{X(const \ X&) for the class X.}
• It is called upon:
  – parameter passing to a function (call-by-value)
  – initialization assignment: \texttt{Word \ x("Oscars"); Word y = x;}
  – value returned by a function:

\begin{verbatim}
Word Word::to_upper_case()
{
    Word x(*this);
    for (char* p = x.str; *p != '\0'; ++p)
        *p += 'A' - 'a';
    return x;
}
\end{verbatim}
Default Copy Constructor

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

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

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

=> CAUTION: the compiler-generated default copy constructor does memberwise copy! i.e.,

song.frequency = movie.frequency;
song.str = movie.str;
Default Copy Constructor
Beware: performs a memberwise copy!

**Default song(movie)**

**movie:**
- **frequency** = 1
- **str** = 0x24ff

**song:**
- **frequency** = 1
- **str** = 0x24ff

**Desirable song(movie)**

**movie:**
- **frequency** = 1
- **str** = 0x24ff

**song:**
- **frequency** = 1
- **str** = 0x53a7

“Brokeback Mountain”
Constructor: Quiz

Quiz: How is class initialization done in the following statements?

• Word vowel("a");

• Word article = vowel;

• Word movie = "Brokeback Mountain";
Member Initialization List

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

```cpp
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 WordPair {
    const Word w1;
    Word w2;

public:
    WordPair(const char* s1, const char* s2) :
        w1(s1),
        w2(s2)
    
    
};

But make sure that the corresponding member constructors exist!
Member Initialization List..

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

class WordPair {
    const Word w1;
    Word w2;

public:
    WordPair(const char* s1, const char* s2) :
        w2(s2)  
    {
        w1 = s1; // quiz: what’s the difference here?
    }
};

But make sure that the corresponding member constructors exist!
Initialization of `const` or & Members

`const` or reference members can **ONLY** be initialized via the member initialization list. (Why?)

```cpp
class Word2 {
    `const char` language;
    `const` Word2& w2;
    `int` frequency;
    `char*` str;
public:
    Word2(`const char*` s1, `const` Word2& w, `int` k = 1) :
        language(‘E’), w2(w), frequency(k) {
            str = `new char` [strlen(s) + 1]; `strcpy` (str, s);
        }
};
```
Initialization of `const` or & Members

`const` or reference members can **ONLY** be initialized via the member initialization list. (Why?)

class Word2 {
    `const char` language;
    `const` Word2& w2;
    `int` frequency;
    `char*` str;

public:
    Word2(`const char*` s1, `const` Word2& w, `int` k = 1) :
        language(‘E’), w2(w), frequency(k) {
            str = new `char` [strlen(s) + 1]; strcpy( str, s);
            language = ‘E’; // compile-time error
            w2 = ?????
        }
};
Default Memberwise Assignment

Word x(“Brokeback Mountain”, 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—which does memberwise copy;
• Memberwise assignment/copy does NOT work whenever memory allocation is required for the class members.
Default Memberwise Assignment

Default x = y

x:
- frequency = 1
- str = 0x24ff

y:
- frequency = 1
- str = 0x24ff

Desirable x = y

x:
- frequency = 1
- str = 0x24ff

y:
- frequency = 1
- str = 0x53a7

"Brokeback Mountain"
Member Class Initialization

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

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

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

    WordPair(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.