Continuing on with classes. And a bit on compiling.

02/17/2023

Ricky Takkar
Part 1/2
Continuing on with classes

http://www.trytoprogram.com/cplusplus-programming/hierarchical-inheritance/
Recap: Constructors

- A constructor has the same name as the class and no return type. It can have as many arguments as needed (just like a regular function).

- You can write as many constructors as you need.

- E.g.,
  - `myClass();`
  - `myClass(int x, std::string str);`
  - `myClass(someOtherClass otherClassObject) and so on`
Recap: Constructors

- Special constructors:
  - Default constructor – takes no arguments
  - Copy constructor (careful with this!) – `myClass(const myClass& other);`
  - Move constructor – `myClass(myClass&& other);` *see Ed post #71 for more info between “&” and “&&”*

- The compiler provides a default constructor (public) when no constructors are defined

- It also provides a default copy and a default move constructor unless the user defines them
Recap: Constructors

• Using the keywords **default** and **delete**, you can enable or disable a constructor.

• What if you want to disable the copy constructor? For e.g., you want unique ownership of a resource and don’t want it duplicated.
  
  • `myClass(const myClass& other) = delete;`

• What if you write a custom constructor that takes some arguments, but still want to keep a default constructor?

  • `myClass() = default;`
Constructors you may be familiar with

• Think of different ways to construct a **vector** object

```cpp
std::vector<int> numbers; // default constructor
std::vector<int> numbers(5); // notice the parentheses, creates a vector of size 5, all 0s
std::vector<int> numbers(5, 100); // all 5 elements initialized to 100
std::vector<int> one_to_ten = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10}; // uses initializer list
std::vector<int> numbers(one_to_ten); // one_to_ten is of type std::vector<int>, invokes the copy constructor
```

• Vectors use dynamically allocated arrays to store elements. What happens when this array needs to grow to accommodate new elements?

• A new array is allocated to which all elements are moved. This is **expensive**

• Tip: **reserve()** allocates sufficient memory to store specified number of elements in vector
Solution:
Vector cannot default construct constituent objects
How to rectify

Solution:
push_back() constructed elements. push_back() will invoke the copy constructor to copy objects into the vector.
Bonus

- To reduce copy operations (i.e., improve performance), one can use `emplace_back()` instead of `push_back()`.
- Note the argument passed to `emplace_back()`: it matches that of `myClass` constructor.
Constructor initializer list

• Problem: How to construct constituent objects of a class in the constructor?

• e.g.,

• Suppose we have `Person(std::string name);`, constructor for Person

• Next, we have Group constructor that contains three Person objects A, B, and C

• How can we construct the Person objects, part of a group, in the constructor of Group?
Constructor initializer list

• Unlike Java, you cannot construct data members in the body of the constructor. In Java, you would do something like,

```cpp
Group::Group() {
    this->A("Ken");
    this->B("Ricky");
    this->C("Alicia");
}
```

• But in C++, objects cannot be null. Member objects must be constructed when the enclosing class object is constructed.
Constructor initializer list

• So, the signature of the constructor and before the body, include a constructor initializer list

  Group::Group(std::string name1, std::string name2, std::string name3) : A(name1), B(name2), C(name3) {
    // body of constructor
  }

• comma-separated list of the type class_member(args...)
Hierarchical Inheritance

• Sometimes, it’s important to create a new (sub) class derived from some base class so objects of the derived class have both: access to inherited traits of the base class, but liberty to extend beyond…

• e.g., child inherits traits of parents but also develops unique features
Hierarchical Inheritance

class BaseClass
{
    // data members
    // member functions
}

class DerivedClass1 : visibility_mode BaseClass
{
    // data members
    // member functions
}

class DerivedClass2 : visibility_mode BaseClass
{
    // data members
    // member functions
}
Recap: Access Specifiers

• 3 access specifiers for class variables and methods in C++:

  • **public** - accessible outside the class
  • **private** (default) - inaccessible outside the class
  • **protected** - only accessible to inherited classes outside the class itself. More on Inheritance later…
Hierarchical Inheritance: Visibility Mode

- Determines how base class features will be inherited by child class

```cpp
class DerivedClass1 : public BaseClass
{ // body }

class DerivedClass2 : private BaseClass
{ // body }

class DerivedClass3 : protected BaseClass
{ // body }
```

Access specifiers of base class maintained as is (private remains private, public remains pub…)

Public and protected access specifiers from base become private (i.e., inaccessible by derived class objects)

Public members from base class become protected (while protected and private members remain as is).
Exercise: Fill in the blanks

<table>
<thead>
<tr>
<th>Base Class</th>
<th>Derived Class</th>
<th>Derived Class</th>
<th>Derived Class</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Public</td>
<td>Protected</td>
<td>Private</td>
</tr>
<tr>
<td>Public</td>
<td>Public</td>
<td>Protected</td>
<td>Private</td>
</tr>
<tr>
<td>Protected</td>
<td>Protected</td>
<td>Protected</td>
<td>Private</td>
</tr>
<tr>
<td>Private</td>
<td>Not inherited</td>
<td>Not inherited</td>
<td>Not inherited</td>
</tr>
</tbody>
</table>
Function Overloading

- What happens if functions share the same name in the same scope?
  - No problem! As long as...
    - At compile time, the compiler can choose which overload to use based on types and number of arguments passed in by caller
Function Overloading

- Both, free and member functions, can be overloaded

<table>
<thead>
<tr>
<th>Overloading Considerations</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Function declaration element</strong></td>
</tr>
<tr>
<td>Function return type</td>
</tr>
<tr>
<td>Number of arguments</td>
</tr>
<tr>
<td>Type of arguments</td>
</tr>
<tr>
<td>Presence or absence of ellipsis</td>
</tr>
<tr>
<td>Use of <code>typedef</code> names</td>
</tr>
<tr>
<td>Unspecified array bounds</td>
</tr>
<tr>
<td><code>const</code> or <code>volatile</code></td>
</tr>
<tr>
<td>Reference qualifiers (<code>&amp;</code> and <code>&amp;&amp;</code>)</td>
</tr>
</tbody>
</table>

What about function overloading with hierarchical inheritance?

```cpp
class BaseClass {
public:
  int foo(int i) {
    std::cout << "foo(int): ";
    return i+1;
  }
};

class DerivedClass : public BaseClass {
public:
  double foo(double d) {
    std::cout << "foo(double): ";
    return d+1.1;
  }
};

int main() {
  DerivedClass dObject = DerivedClass();
  std::cout << dObject.foo(4) << std::endl;
  std::cout << dObject.foo(4.3) << std::endl;
  return 0;
}
```

**Question:** What will the program output?

A. foo(double): 5.1
   foo(double): 5.4
B. foo(int): 5
   foo(double): 5.4
C. Error

No overload resolution between class hierarchy in C++
Exercise

• Using demo code from Recitation 3, implement
  
  • Hierarchical inheritance (hint: create some new classes that inherit from Student - feel free to modify Student)
  
  • Constructor initializer list
  
  • Function overloading
Part 2/2
A bit about compiling
Recap: Compiling Classes

- Run `g++ -o exec_name main.cpp rest.cpp ...`

- Include all the cpp files in the g++ command

- Ignore header files in compilation command as they should be included in the cpp files

- Only one program should contain the main function (in the above example, main.cpp)
Journey of C++ Compilation

• Step 1: The preprocessor

• Before the C++ compiler compiles, the source code file is processed by a preprocessor

• The compiler automatically invokes the preprocessor

• Preprocessor commands start with “#”, e.g., #include <iostream>
Journey of C++ Compilation

• Step 2: The compiler

  • By now, the compiler has included all header files and expanded `#include` statements

  • Compiler transforms C++ source code into **object code** file (*.o) containing binary version of source code

  • Object code file is not directly executable
Journey of C++ Compilation

• Step 3: The linker

• Separate program called ld akin to preprocessor (also invoked automatically by compiler like preprocessor program)

• Links together object files (including object files created from source code and pre-compiled object files collected into library files with *.a or *.so extension) into a single binary executable
Build Files and Generate Executables

Makefile

- Makefile is a special file containing shell commands executed by running the ‘make’ command inside the Makefile directory.

```
CC = g++
CFLAGS = -g -Wall
TARGET = output
all: $(TARGET)
$(TARGET): main.o hello.o
    $(C) $(CFLAGS) -o $(TARGET) main.o hello.o
main.o: main.cpp hello.hpp
    $(C) $(CFLAGS) -c main.cpp
hello.o: hello.hpp hello.cpp
    $(C) $(CFLAGS) -c hello.cpp
```
Build Files and Generate Executables  CMake

• Why CMake?
  
  • Makefiles are low-level, clunky creatures
  
  • CMake is a higher level language to automatically generate Makefiles
  
  • CMake contains more features, such as finding library, files, header files; it makes the linking process easier, and gives readable errors

• What is CMake?
  
  • CMake is an extensible, open-source system that manages the build process in an operating system and in a compiler-independent manner.
  
  • CMakeLists.txt files in each source directory are used to generate Makefiles
Build Files and Generate Executables

CMake

Example

- Build and Run
  - Navigate to the source directory, and create a build directory
    - `$ cd ./myproject` & `$ mkdir build`
  - Navigate to the build directory, and run CMake to configure the project and generate a build system
    - `$ cd build` & `$ cmake`
  - Call build system to compile/link the project
    - Either run `$ make`
    - Or run `$ cmake-build`
Cmake
3. Cmake with subdirectory

- CMakeLists.txt files placed in each source directory are used to generate standard build files (e.g., makefiles on Unix and projects/workspaces in Windows MSVC).
- CMake supports in-place and out-of-place builds, and can therefore support multiple builds from a single source tree.
- add_subdirectory(source_dir [binary_dir] [EXCLUDE_FROM_ALL])

  Adds a subdirectory to the build. The source_dir specifies the directory in which the source CMakeLists.txt and code files are located.

```cmake
add_subdirectory(say-hello)
add_subdirectory(hello-exe)
```

```cmake
add_library(
  say-hello
  hello.hpp
  hello.cpp
)
```

```cmake
target_include_directories
(say-hello PUBLIC
"${CMAKE_CURRENT_SOURCE_DIR}"
)
```

```cmake
target_compile_definitions
(say-hello PUBLIC
SAY_HELLO_NUM=5)
```
Cmake

- target_include_directories(<target> [SYSTEM] [AFTER|BEFORE] <INTERFACE|PUBLIC|PRIVATE> [items1...] )

- Set include directory properly

- The PUBLIC, PRIVATE and INTERFACE keywords can be used to specify both the link dependencies and the link interface in one command.
  - PUBLIC(default): All the directories following PUBLIC will be used for the current target and the other targets that have dependencies on the current target
  - PRIVATE: All the include directories following PRIVATE will be used for the current target only
  - INTERFACE: All the include directories following INTERFACE will NOT be used for the current target but will be accessible for the other targets that have dependencies on the current target
References

1. https://www.youtube.com/watch?v=HcESuwmlHEY

