Lecture 16: C++ OOP

November 27th, 2018

Topics: Object oriented programming in C++.

1 Classes in C++

A class in C++ is a user-defined type. To simplify slightly, classes may contain

1. Data attributes, and
2. Methods (function attributes).

Members must be defined at once in the class definition, and new members cannot be added to an already defined class (unlike members of namespaces).

1.1 Accessing Attributes

Let's consider a first example for how we may define our own data-attributes; from `src/class1.cpp`:

```
1  #include <string>
2
3  class user {            // Class definition.
4    // Data attributes.
5    int id;               // Uninitialized variable.
6    int zz = 0;           // Member Initialization is a C++11 feature.
7    std::string name;     // Default-initialized to "" by string-constructor.
8  };
9
10 int main() {
11   user u;               // Object (an instance of the class).
12   return 0;
13 }
```

Note that our class definition can impose default values for attributes. If we're diligent with our compiler warnings, the compiler instructs us of an unused variable; output:

```
$ g++ -std=c++11 -Wall -Wextra -Wconversion src/class1.cpp -o src/class1
src/class1.cpp:5:7: warning: private field 'id' is not used [-Wunused-private-field]
  int id;
      ^
1 warning generated.
$ ./src/class1
```

Running the program doesn't product any output, expectedly. Naturally, we might try to use (or access) an attribute, to perhaps initialize its value.

1.1.1 Classes: Member Access is Private by Default

It's correct to use dot notation (`.`) to access attributes, but we the compiler guards against (malicious) users; `src/class2.cpp`:

```
1  #include <iostream>
2  #include <string>
3
4  class user {
5    int id;
6    std::string name;
7  };
8
9  int main() {
10   user u;
11   u.id = 7; // Member access via dot notation.
12   std::cout << "u.id = " << u.id << std::endl;
13   return 0;
14 }
```

We've earned a nice error (available even without the warning flags); output:

```
$ g++ -Wall -Wextra -Wconversion src/class2.cpp -o src/class2
src/class2.cpp:13:5: error: 'id' is a private member of 'user'
  u.id = 7; // Member access via dot notation
    ^
src/class2.cpp:6:7: note: implicitly declared private here
  int id;
      ^
src/class2.cpp:14:31: error: 'id' is a private member of 'user'
  std::cout << "u.id = " << u.id << std::endl;
                              ^
src/class2.cpp:6:7: note: implicitly declared private here
  int id;
      ^
2 errors generated.
```

1.1.2 Structs: Member Access is Public by Default

Structs and classes are interchangeable, they simply have different defaults for access specifiers. Whereas a class defaults to having members be private, a struct defaults to public access; `src/struct1.cpp`:

```
1  #include <iostream>
2  #include <string>
3
```

```
4  struct user {
5    int id;
6    std::string name;
7  };
8
9  int main() {
10   user u;
11   u.id = 7;
12   std::cout << "u.id = " << u.id << std::endl;
13   return 0;
14 }
```

```
$ g++ -Wall -Wextra -Wconversion src/struct1.cpp -o src/struct1
$ ./src/struct1
u.id = 7
```

1.2 Public, Private, and Access Specifiers

C++ is strict about member access: we need to understand the default behaviors, and also how to override defaults via access specifiers.

- `private`: attribute or method only accessible from within member(s) of the same class.
- `public`: attribute or method accessible by any-user via dot notation.
- Default access specifier for `class` is `private`.
- Default access specifier for `struct` is `public`.

It's even possible to provision that a private attribute be accessible from select methods (themselves not a part of the class) using friends.

1.2.1 Overriding Default Access, a `class` Example

`src/class3.cpp`:

```
1  #include <iostream>
2  #include <string>
3
4  class user {
5   public: // everything after this will be public
6    int id;
7    std::string name;
8  };
9
10 int main() {
11   user u;
12   u.id = 7;
13   u.name = "Leland";
14   std::cout << "u.id = " << u.id << std::endl;
15   std::cout << "u.name = " << u.name << std::endl;
16   return 0;
17 }
```

```
$ g++ -Wall -Wextra -Wconversion src/class3.cpp -o src/class3
$ ./src/class3
u.id = 7
u.name = Leland
```

1.2.2 Overriding Default Access, a `struct` Example

From `src/struct2.cpp`:

```
#include <iostream>
#include <string>

struct user {
  int id;
 private: // everything after this will be private
  std::string name;
};

int main() {
  user u;
  u.id = 7;
  u.name = "Leland";
  std::cout << "u.id = " << u.id << std::endl;
  std::cout << "u.name = " << u.name << std::endl;
  return 0;
}
```

```
$ g++ -Wall -Wextra -Wconversion src/struct2.cpp -o src/struct2
src/struct2.cpp:13:5: error: 'name' is a private member of 'user'
  u.name = "Leland";
    ^
src/struct2.cpp:7:15: note: declared private here
  std::string name;
              ^
src/struct2.cpp:15:33: error: 'name' is a private member of 'user'
  std::cout << "u.name = " << u.name << std::endl;
                                ^
src/struct2.cpp:7:15: note: declared private here
  std::string name;
              ^
2 errors generated.
```

1.2.3 Mix and Match: Selectively Making Members Public

`src/class4.cpp`:

```
#include <iostream>
#include <string>

class user {
```

```
5     int id;
6    public:
7     std::string name;
8    private:
9     int age;
10 };
11
12 int main() {
13   user u;
14   u.id = 7;
15   u.name = "Leland";
16   u.age = 12;
17
18   std::cout << "u.id = " << u.id << std::endl;
19   std::cout << "u.name = " << u.name << std::endl;
20   std::cout << "u.age = " << u.age << std::endl;
21
22   return 0;
23 }
```

Output:

```
$ g++ -Wall -Wextra -Wconversion src/class4.cpp -o src/class4
src/class4.cpp:14:5: error: 'id' is a private member of 'user'
  u.id = 7;
    ^
src/class4.cpp:5:7: note: implicitly declared private here
  int id;
      ^
src/class4.cpp:16:5: error: 'age' is a private member of 'user'
  u.age = 12;
    ^
src/class4.cpp:9:7: note: declared private here
  int age;
      ^
src/class4.cpp:18:31: error: 'id' is a private member of 'user'
  std::cout << "u.id = " << u.id << std::endl;
                              ^
src/class4.cpp:5:7: note: implicitly declared private here
  int id;
      ^
src/class4.cpp:20:32: error: 'age' is a private member of 'user'
  std::cout << "u.age = " << u.age << std::endl;
                                ^
src/class4.cpp:9:7: note: declared private here
  int age;
      ^
4 errors generated.
```

1.3 Size and Layout of Classes are Static (i.e. known at compile time)

It's important to recognize that unlike Python, we can't simply add members to a class "on-the-fly". Recognize that a class is a (compound) *type*, and the compiler must be able to reason about it (and its size, for example). It has a particular layout in memory, which we'll learn more about in CME 212, which naturally depends on the number of objects being stored and their type. `src/struct3.cpp`:

```
1  #include <iostream>
2
3  struct user { int id; };
4
5  int main() {
6    user u;
7    u.id = 7;
8    u.age = 12;   // Leads to an error: no attribute named 'age'.
9    return 0;
10 }
```

```
$ g++ -Wall -Wextra -Wconversion src/struct3.cpp -o src/struct3
src/struct3.cpp:10:5: error: no member named 'age' in 'user'
  u.age = 12;
  ~ ^
1 error generated.
```

1.4 Methods

Let's define our first method; `src/class5.cpp`:

```
1  #include <iostream>
2
3  class user {
4    // data member initialization is a C++11 feature
5    int id = 7;
6    int getId(void) { return id; }
7  };
8
9  int main() {
10   user u;
11   std::cout << "u.getId() = " << u.getId() << std::endl;
12   return 0;
13 }
```

We might be at first surprised by the output; we must recall that `class`es default to private access. Our method `getId` is not accessible from *outside* of class `user`.

```
$ g++ -std=c++11 -Wall -Wextra -Wconversion src/class5.cpp -o src/class5
src/class5.cpp:13:36: error: 'getId' is a private member of 'user'
  std::cout << "u.getId() = " << u.getId() << std::endl;
                                   ^
src/class5.cpp:6:7: note: implicitly declared private here
  int getId(void) {
```

```
      ^
1 error generated.
```

If we take care to override the default access behavior, `src/class6.cpp`:

```
1  #include <iostream>
2
3  class user {
4    int id = 7;
5   public:
6    int getId(void) { return id; }
7  };
8
9  int main() {
10   user u;
11   std::cout << "u.getId() = " << u.getId() << std::endl;
12   return 0;
13 }
```

Our program now compiles and executes without error.

```
$ g++ -std=c++11 -Wall -Wextra -Wconversion src/class6.cpp -o src/class6
$ ./src/class6
u.getId() = 7
```

1.5 Naming Conventions and Passing/Initializing Arguments

Suppose we have a set of attributes associated with a class. We may want to provide methods which facilitate initialization/modification and retrieval. As an implementation detail, we must take care to disambiguate between the argument passed to the method and the underlying attribute we wish to modify; `src/class7.cpp`:

```
1  #include <iostream>
2
3  class user {
4    int id = 1;
5   public:
6    int getId(void) { return id; }
7    void setId(int id) { id = id; }   // Oops, this doesn't assign/over-write out attribute.
8  };
9
10 int main() {
11   user u;
12   u.setId(7);
13   std::cout << "u.getId() = " << u.getId() << std::endl;
14   u.setId(42);
15   std::cout << "u.getId() = " << u.getId() << std::endl;
16   return 0;
17 }
```

Output:

```
$ g++ -std=c++11 -Wall -Wextra -Wconversion src/class7.cpp -o src/class7
src/class7.cpp:7:27: warning: explicitly assigning value of variable of type
                             'int' to itself [-Wself-assign]
```

```
  void setId(int id) { id = id; }
                       ~~ ^ ~~
1 warning generated.
$ ./src/class7
u.getId() = 1
u.getId() = 1
```

1.5.1 Total Disambiguation via Distinct Identifiers

We could simply choose different variable names for arguments used to initialize corresponding attributes; `src/class8.cpp`:

```
1  #include <iostream>
2
3  class user {
4    int id = 1;
5   public:
6    int getId(void) { return id; }
7    void setId(int id_) { id = id_; }
8  };
9
10 int main() {
11   user u;
12   u.setId(7);
13   std::cout << "u.getId() = " << u.getId() << std::endl;
14   u.setId(42);
15   std::cout << "u.getId() = " << u.getId() << std::endl;
16   return 0;
17 }
```

```
$ g++ -std=c++11 -Wall -Wextra -Wconversion src/class8.cpp -o src/class8
$ ./src/class8
u.getId() = 7
u.getId() = 42
```

1.5.2 Keyword `this`

We can also use keyword `this`, whose value is an address of the object in context. We can use the `->` operator as a syntactic sugar for de-referencing the memory address and accessing the corresponding member attribute; see `src/class9.cpp`:

```
1  #include <iostream>
2
3  class user {
4    int id = 1;
5   public:
6    int getId(void) { return id; }
7    void setId(int id) { this->id = id; }  // Note use of this->
8  };
9
10 int main() {
11   user u;
12   u.setId(7);
```

```
13    std::cout << "u.getId() = " << u.getId() << std::endl;
14    u.setId(42);
15    std::cout << "u.getId() = " << u.getId() << std::endl;
16    return 0;
17  }
```

Now this behaves as intended.

```
$ g++ -std=c++11 -Wall -Wextra -Wconversion src/class9.cpp -o src/class9
$ ./src/class9
u.getId() = 7
u.getId() = 42
```

2 Constructor

There are a few essential ingredients which are essential to each class.[1] A constructor is a special type of member function which is used for initialization of objects of the corresponding class type. Put differently, anytime an object of a particular class is created, *some type* of constructor is called (which one depends on the context of use).

2.1 Default Constructors

In C++, there is a notion of a *default constructor* which can be called with *no* arguments. In C++, if no user-declared constructor is defined, one is generated automatically by the compiler. The behavior of the default constructor is to simply default initialize each member attribute in turn. We can replace the default constructor with a custom constructor by defining a method name with the same name as the class.

Comparison with Methods *Unlike* other methods, the constructor does *not* return anything, not even void! *Like* other methods, the constructor can take arguments; we can even define *multiple* constructors which differ in the number (and types) of arguments which are accepted. We can think of each of these constructors as being totally distinct methods for initialization, which happen to share the same name, but can be uniquely found given a well-written constructor call.

Example: Default Constructor Here, we define a non-default constructor.

```
1     int id;
2   public:
3     user() { this->id = 43; }    // This is a default-Constructor.
```

[1] These include default, copy, and move constructors, as well as copy/move assignment operators, and finally a destructor. We'll learn about how to define these and use them for performant computing in CME 212.

```
4     int getId(void) { return id; }
5   };
```

Since we have provided a definition for a default constructor (i.e. one which accepts *no* arguments), the compiler no longer defines one for us. It's also possible to create *another* constructor (separate from the default constructor); `src/class10.cpp`:

```
1   #include <iostream>
2
3   class user {
4     int id;
5    public:
6     user(int id) { this->id = id; }   // This is a Constructor.
7     int getId(void) { return id; }
8   };
9
10  int main() {
11    user u(13);
12    std::cout << "u.getId() = " << u.getId() << std::endl;
13    return 0;
14  }
```

Now, we have a one-argument constructor which can initialize `id` attribute to the value passed as argument. Output:

```
$ g++ -std=c++11 -Wall -Wextra -Wconversion src/class10.cpp -o src/class10
$ ./src/class10
u.getId() = 13
```

2.1.1 If Any User-Constructor Defined, Default Constructor *not* Auto-generated

Note that if *any* user-defined constructor is provided, then the compiler *foregoes defining* a default constructor; [2] `src/class11.cpp`:

```
1   #include <iostream>
2
3   class user {
4     int id;
5    public:
6     user(int id) { this->id = id; }
7     int getId(void) { return id; }
8   };
9
10  int main() {
11    user u;
12    std::cout << "u.getId() = " << u.getId() << std::endl;
13    return 0;
14  }
```

Output:

```
$ g++ -std=c++11 -Wall -Wextra -Wconversion src/class11.cpp -o src/class11
```

[2] We emphasize that the default constructor default-initializes each member (in this case simply an `id` variable), and since the `int` type is atomic and doesn't have a constructor, its value is left uninitialized

```
src/class11.cpp:11:8: error: no matching constructor for initialization of 'user'
  user u;
       ^
src/class11.cpp:6:3: note: candidate constructor not viable: requires single argument 'id',
                     but no arguments were provided
  user(int id) { this->id = id; }
  ^
src/class11.cpp:3:7: note: candidate constructor (the implicit copy constructor)
                     not viable: requires 1 argument, but 0 were provided
class user {
      ^
src/class11.cpp:3:7: note: candidate constructor (the implicit move constructor)
                     not viable: requires 1 argument, but 0 were provided
1 error generated.
```

Reasons for this behavior will be illuminated in CME 212 when we learn about invariants.

2.2 Circle Example

src/circle1.cpp:

```
1  #include <cmath>
2  #include <iostream>
3
4  class circle {
5    double x, y, r;
6   public:
7    circle(double x, double y, double r) {
8      this->x = x;
9      this->y = y;
10     this->r = r;
11   }
12   double getArea(void) {
13     return M_PI*r*r;
14   }
15 };
16
17 int main() {
18   circle c(1.2, 3.4, 2.);
19   std::cout << "c.getArea() = " << c.getArea() << std::endl;
20   return 0;
21 }
```

Output:

```
$ g++ -Wall -Wextra -Wconversion src/circle1.cpp -o src/circle1
$ ./src/circle1
c.getArea() = 12.5664
```

3 Program Decomposition: Organizing into Multiple Files

3.1 User-Interface (`.hpp`) vs. Implementation Details (`.cpp`)

We've discussed before that the user-interface for a class may be read by one set of users; `src/circle2.hpp`:

```
#ifndef CIRCLE2_HPP
#define CIRCLE2_HPP

class circle {
  double x, y, r;
 public:
  circle(double x, double y, double r);
  double getArea(void);
};

#endif /* CIRCLE2_HPP */
```

And implementation details may be stored in another file; `src/circle2.cpp`:

```
#include <cmath>

#include "circle2.hpp"

circle::circle(double x, double y, double r) {
  this->x = x;
  this->y = y;
  this->r = r;
}

double circle::getArea(void) {
  return M_PI*r*r;
}
```

We can call `#include` our header file within our `main` program; `src/main2.cpp`:

```
#include <iostream>
#include "circle2.hpp"

int main() {
  circle c(1.2, 3.4, 2.);
  std::cout << "c.getArea() = " << c.getArea() << std::endl;
  return 0;
}
```

And finally, instruct our compiler where to look for implementation details (for both `main` and `circle`); here we use `-I` to specify to the compiler where we shall look for external header files.

```
$ g++ -Wall -Wextra -Wconversion src/circle2.cpp src/main2.cpp -o src/main2 -I./src
$ ./src/main2
c.getArea() = 12.5664
```

3.2 Example: Class Decomposition with a Namespace

A `circle` is really just one of many possible `geometrys`; `src/circle3.hpp`:

```
#ifndef CIRCLE3_HPP
#define CIRCLE3_HPP

namespace geometry {

  class circle {
      double x, y, r;
    public:
      circle(double x, double y, double r);
      double getArea(void);
      double getPerimeter(void);
  };
}
#endif /* CIRCLE3_HPP */
```

Having laid out our interface, we can proceed with an implementation; `src/circle3a.cpp`:

```
#include <cmath>
#include "circle3.hpp"

namespace geometry {
  circle::circle(double x, double y, double r) {
    this->x = x;
    this->y = y;
    this->r = r;
  }
  double circle::getArea(void) {
    return M_PI*r*r;
  }
}
```

Perhaps we wish to place methods in separate files, splitting apart `getArea` from `getPerimiter`; `src/circle3b.cpp`:

```
#include <cmath>
#include "circle3.hpp"

namespace geometry {
  double circle::getPerimeter(void) {
    return 2.*M_PI*r;
  }
}
```

We've still laid everything out in a single header file, since we've learned that a class can't be dynamically added to; `src/main3.cpp`:

```
#include <iostream>
#include "circle3.hpp"

int main() {
  geometry::circle c(1.2, 3.4, 1.8);
  std::cout << "c.getArea() = " << c.getArea() << std::endl;
  std::cout << "c.getPerimeter() = " << c.getPerimeter() << std::endl;
  return 0;
}
```

We instruct our compiler where to look for implementations of `getArea` (`circle3a`, `getPerimiter` (`circle3b`), and where to look for the main sub-routine; output:

```
$ g++ -Wall -Wextra -Wconversion src/circle3a.cpp src/circle3b.cpp src/main3.cpp \
  -o src/main3 -I./src
$ ./src/main3
c.getArea() = 10.1788
c.getPerimeter() = 11.3097
```

4 Objects and Containers

Since a class is just a (user-defined, possibly compound) type, the compiler understands how much space each instantiated object will require, and we can for example store instances of our class-object as elements in a container; `src/container.cpp`:

```
1  #include <iostream>
2  #include <vector>
3  #include "circle3.hpp"
4
5  int main() {
6    std::vector<geometry::circle> circles;
7    circles.emplace_back(8.3, 4.7, 0.5);
8    circles.emplace_back(4.1, 2.3, 1.4);
9    circles.emplace_back(-3.2, 0.8, 14.4);
10
11   for(auto& c : circles)   # C++11 mechanism for a for-loop (by reference).
12     std::cout << "c.getArea() = " << c.getArea() << std::endl;
13   return 0;
14 }
```

We could of course index into our vector to retrieve each element as well, using the canonical syntax: `for (unsigned i = 0; i < circles.size(); i++)`. Notice that if we did not use an ampersand (`&`) to declare a reference, that we would be creating a new copy of each `circle` before using it in the for-loop, whence it's prefereable instead to use a reference; output:

```
$ g++ -std=c++11 -Wall -Wextra -Wconversion src/circle3a.cpp src/circle3b.cpp \
  src/container.cpp -o src/container -I./src
$ ./src/container
c.getArea() = 0.785398
c.getArea() = 6.15752
c.getArea() = 651.441
```

5 Appendix: Use of keyword `static`

We've mentioned previously a concept of *statically allocated* arrays, i.e. an array whose length is known at compile time. Such an array can be allocated on the stack (in temporary

memory, where the compiler disposes of the object when it goes out of scope) or on the heap (persistent storage, wherein the user must clean up after themselves). Totally separate from this concept, the keyword `static` appears in two *different* (and unrelated) contexts in C++.

> Inside a class, `static` declares members that are not bound to class instances.
> Outside a class definition, it refers to a type of storage duration.

Storage Duration: Automatic vs. Static All objects are associated with a kind of storage duration. The default is *automatic* storage duration, wherein the storage is allocated for the object at the beginning of the enclosing code block and deallocated at the end. In contrast, we could also request *static* storage duration, wherein the object is allocated *once* when the program begins, and deallocated *once* when the program ends: only one instance of the object exists.

Static Members If keyword `static` is used within a class definition, it indicates that the member shall *not* be associated with any particular object(s) of the class; instead, they are independent variables (or functions) which happen to have static storage duration.

6 Reading

C++ Primer, Fifth Edition by Lippman et al.: Section 1.5: Introducing Classes.