Discussion 1B Notes (Week 9): Class

TA: Zhou Ren

Acknowledgement: Brian Choi's material

Classes

We have seen different types of variables so far – `int`, `double`, `string`, etc. But these basic types are not enough to represent everything. We might want to throw in things as random as cats, dogs, bears, apples, and trees into my program (quite literally). You will soon see these animals and plants in your program.

We'll call these customized types **classes**. We will start with a class with minimal functionalities, and keep adding more components onto it to make it more complete.

Basics

A **class** is a construct used to group related fields (variables) and methods (functions). More technically, it is a collection of variables (which may be of different types) and possibly some functions associated with them, put together to serve some specific purposes. This may not sound intuitive in words. Let us create a cat for an example.

```
class Cat
{
  public:
    int m_age;
    void meow();
};
```

Ignore the line `public:` for now. Within this Cat class, there are two **members** – a *member variable* `m_age` and a *member function* named `meow()`. One can access these members by using a dot, as done with structures. See the example below:

```
class Cat
{
  public:
    int m_age;        // Stores the age.
    void meow();      // Prints "MEOW!" and increments the age by 1.
};                    // Don't forget the semicolon!

int main()
{
  Cat kitty1;           // A Cat instance.
  Cat kitty2;           // Another Cat instance.
  kitty1.m_age = 1;     // This cat is 1 year old.
  kitty2.m_age = 3;     // This cat is 3 years old.

  cout << "Kitty1 is " << kitty1.m_age << " years old." << endl;
  cout << "Kitty2 is " << kitty2.m_age << " years old." << endl;
  kitty1.meow();
  cout << "Kitty1 is " << kitty1.m_age << " years old." << endl;
  kitty1.meow();
  cout << "Kitty1 is " << kitty1.m_age << " years old." << endl;
}
```

Output:

Hopefully the above example is intuitive. A few things to note:

• Note that every **instance** of a class (in the example, `kitty1` and `kitty2`) has its own copy of members.
• To add to the above bullet, `m_age` and `meow()` by themselves do not make much sense in `main()`. You must indicate which instance these members belong to. (e.g., `kitty1.m_age`)
• By convention, we capitalize the first letter of class name. (e.g., `Cat`, `Dog`, `OakTree`)

Wait, isn't something missing? The above class definition is incomplete without the definition of the function `meow()`. Let us define it now.

Member Functions

Here is the definition of `meow()` (to be filled in in class).

```
void Cat::meow()
{




}
```

Notice the new syntax for function header. This is really no different from defining other functions, but the name of the class is included to indicate the membership of the function. The general syntax for it is:

```
return_type Class_name::function_name( argument_list )
{



}
```

Within the definition of a **member function**, you <u>can</u> access and manipulate all **member variables** of the class freely. We'll see cases where you can't access some members from outside in the following section.

Public/Private Members

Let's direct our attention to this line "`public:`" now. This simply means, all functions and variables defined under this line are "public" to others, such as `main()`. But there's a danger to making everything public. Suppose I do this:

```
kitty1.m_age = -20;
```

In C++, there's nothing wrong about setting an integer to `-20`. But how about the logic? It is sensical to say my cat's age turned negative? Even if this does make sense, it still bothers me (from the perspective of the cat) to let others to set my cat's age freely. The pet might want to keep her age information private, so it has more control over this value. This leads us to the concept of **private** members of a class. A private member cannot be accessed by an external entity (e.g. main function, other functions and classes), but the class itself should have the total control over these private members.

```
class Cat
{
  public:
    void meow();            // prints "Meow!" and increment m_age by 1
  private:
    int m_age;              // age
};

void Cat::meow()
{
  /* definition omitted */
}

int main()
{
  Cat kitty1;

  kitty1.m_age = 5;         // ERROR!
}
```

Accessors

Well, perhaps the cat wants to disclose her age information, but does not want to allow anyone to change it. Let us add a public function that "accesses" `m_age`.

```
class Cat
{
  public:
    int age();          // returns m_age
    void meow();
  private:
    int m_age;
};

int Cat::age()
{
  return m_age;
}
```

Now in `main()`, we can call `kitty1.age()` (because the function is public!) to access `kitty1`'s age, without violating the privacy rule. But since the function only returns it, we still cannot modify it. Such function is called an **accessor**.

Modifiers (Mutators)

The function that lets us change the internal value of a class instance is called a **modifier** or **mutator**. `meow()` is a modifier, because it changes the value of `m_age`. Perhaps this allows too little freedom for the user of the Cat class. But we still do not want to make `m_age` public, since some dumb user might assign a negative number and mess up the program. We'll provide a function that lets users to modify `m_age`, with some restrictions. In particular, we would like to keep it between 0 and 100. If the user enters some age outside this range, we simply won't change any value.

```
void Cat::setAge(int newAge)
{






}
```

The user can now read and change `m_age`, but this is not the same as keeping `m_age` public. As you will see throughout your engineering career, this is a common practice in pretty much every system design -- providing an **interface** for the users of the system to prevent users from messing with your system and maintain system reliability. You'll see more of this in CS32.

Exercise Now, provide a boolean function `dead()` that returns true if the age is greater than or equal to 100. Also, modify the function `meow()`, so that it does not increase the age if the cat is dead already.

```
bool Cat::dead()
{


}

void Cat::meow()
{







}
```

With the addition of `dead()`, our class now looks as follows:

```
class Cat
{
  public:
    void meow();
    int age();                   // Accessor.
    void setAge(int newAge);     // Mutator.
    bool dead();                 // Returns true if m_age == 100.
  private:
    int m_age;
};

/* Member function definitions are omitted. */
```

Constructors

If you look closely again at the Cat class we have built so far, there is still something missing – there's no initialization! We did all the dirty work to keep our age within the range of 0 and 100, but if it is initialized to an invalid number, we are out of hope. To guarantee that the age never becomes negative, the user of this class must call `setAge(0)` to set `m_age` to 0 as soon as she creates the instance. This is inconvenient, and we can't expect every user of the Cat class to know and do this. We want some mechanism to "automatically" initialize the class when it gets created.

```
class Cat
{
  public:
    Cat();         // default constructor
    void meow();
    int age();
    void setAge(int newAge);

  private:
    int m_age;
};

Cat::Cat()
{
  setAge(0);
  cout << "A cat is born" << endl;
}
```

That type-less function called `Cat()` (its name must be the same as the class name) is our **constructor**, and will be called automatically when a Cat instance is created. It is still a function, without any arguments. When the constructor has no arguments, it is called the **default constructor**.

```
Cat kitty;            // uses default constructor -- Cat()
Cat *p1 = new Cat;    // uses Cat()
Cat *p2 = new Cat();  // uses Cat()
```

The above statements will each create a `Cat` instance, using the default constructor. As hinted by the term "default", we can have multiple constructors for a class, by overloading it.

```
class Cat
{
  public:
    Cat();               // default constructor
    Cat(int initAge);    // constructor with an initial age
    void meow();
    int age();
    void setAge(int newAge);
  private:
    int m_age;
};

Cat::Cat(int initAge)
{
  Cat();             // I can call the default constructor,
  setAge(initAge);   // and then set the age to initAge
}
```

Question Can you guess what this new constructor will do?

You can call this new constructor by creating your classes in the following way:

```
Cat kitty1(3);
Cat kitty2(-15);     // what is the age going to be?
```

Initialization List

When there are more than one properties that need to be initialized, it might look a bit dirty to use assignment operations to every one of them. Suppose Cat had 3 variables, `m_age`, `m_weight`, and `m_gender`. In our constructor, we would do the following:

```
Cat::Cat()
{
  m_age = 0;
  m_weight = 10;
  m_gender = 1;    // e.g. 1 for female, 2 for male
  /* some code */
}
```

In this example this doesn't look that bad, but when there are more lines of code to the constructor, it sometimes looks better to keep these initialization statements separately. An **initialization list** is intended to serve this purpose.

```
Cat::Cat()
: m_age(0), m_weight(10), m_gender(1)
{
  /* some code */
}
```

It begins with a single colon after the argument list. Make sure you separate each element with a comma.

Note that you need not have all the private member variables initialized, though. Just include ones that you have to initialize.

Dynamic Allocation

Let's get back to the topic of dynamic allocation. If you understood how it worked for `int` and `double`, then it should be straightforward with classes.

```
Cat *pKitty = new Cat();
Cat *pKitty2 = new Cat(10);
```

These statements should be straightforward to understand, if you've mastered your pointers. If not, review pointers and come back to this.

As with structures, you can use `pKitty->meow()` as a shortened form of `(*pKitty).meow()`. (Warning: This only works if `pKitty` is a pointer!!!)

Destructors

Finally, as with other dynamically allocated variables, you must remove the dynamic instance from memory when you are done using it:

```
delete pKitty;
delete pKitty2;
```

However, some classes need special treatment when deleted. For example, the class might have created some dynamic arrays that need to be deleted, or you might want to print out something right before the instance gets deleted. A **destructor**, a counterpart of constructors, is the function that gets called when deleted.

There can't be multiple destructors though – there is no point of having several destructors since the instance will be gone anyways.

```
class Cat
{
  public:
    Cat();
    Cat(int initAge);
    ~Cat();           // destructor
    void meow();
    int age();
    void setAge(int newAge);

  private:
    int m_age;
};

Cat::~Cat()
{
  if (m_age > 5)
    cout << "R.I.P." << endl;
}
```

A destructor's name starts with **~**, followed by the name of the class, with no return type or arguments. The above destructor doesn't do much, but prints out our message to the poor cat whose life ended if she lived long enough.

Cat, Complete

We have come a long way to get this Cat implemented. Can you predict the output?

```
class Cat
{
  public:
    Cat();
    Cat(int initAge);
    ~Cat();
    void meow();
    int age();
    void setAge(int newAge);
    bool dead();

  private:
    int m_age;
};

Cat::Cat()
:m_age(0)
{
  cout << "A cat is born" << endl;
}

Cat::Cat(int initAge)
{
  Cat();
  setAge(initAge);
}

Cat::~Cat()
{
  if (age > 5)
    cout << "R.I.P." << endl;
}

void Cat::meow()
{
  // see above
}
```

```
int Cat::age()
{
  return m_age;
}

void Cat::setAge(int newAge)
{
  // see above
}

bool Cat::dead()
{
  // see above
}

int main()
{
  Cat* kitty1 = new Cat();
  Cat* kitty2 = new Cat(99);

  kitty1->meow();
  kitty2->meow();

  cout << "The age of kitty1 = "
       << kitty1->age() << endl;
  cout << "The age of kitty2 = "
       << kitty2->age() << endl;
  cout << “Is kitty2 dead? “;

  if (kitty2->dead())
    cout << “YES” << endl;
  else
    cout << “NO” << endl;

  delete kitty1;
  delete kitty2;
}
```

Classes vs. Structures

Structures are remnants of C, and in C++. Originally, structures could not have member functions -- its sole purpose was to group variables. As the paradigm of Object Oriented Programming emerged, classes were introduced, and structures in C are just a special case of classes (i.e. classes with public member variables and nothing else). The only difference between classes and structures is that, if you declare a member without indicating the publicness/privateness of it, it will be considered public in a structure, and private in a class. In most cases, you won’t see that many cases where structures are used to represent an object, as most people just use classes. Structures are mostly used to serve their original purpose of grouping related variables.