CMSC 240 Lecture 7

CMSC 240 Software Systems Development

Fall 2023

Today

- Classes and OOP
- Breakout design activity
- Coding a class in C++
- Breakout coding activity

Today

- Classes and OOP

- Breakout design activity

- Coding a class in C++

- Breakout coding activity

Procedural Programing

```
int main()
{
    procedure1();

    // ...

    procedure2();

    return 0;
}
```

```
void procedure1()
{
    // ...
}
```

```
void procedure2()
{
    // ...
    procedure3();
}
```

```
void procedure3()
{
    // ...
}
```

Procedural vs. Object-Oriented

- Procedural programming
  - Data and operations on data are *separate*
  - Requires passing data to methods
- Object-oriented programming
  - Data and operations on data are *together* in an object
  - Organizes programs like the real world
    - All objects are associated with both attributes and activities
  - Using objects improves software reusability and makes programs easier to both develop and maintain

How do we accomplish this in C++? With **classes**!

abstraction

Design that hides the details of how something works while still allowing the user to access complex functionality.

class

A class defines a new data type for our programs to use.

This sounds familiar...

```
struct Point3D
{
    double x;
    double y;
    double z;
};
```

```
struct Car
{
    int year;
    string brand;
    string model;
};
```

struct

A way to group together variables of different data types under a single name.

Then what's the difference between a **class** and a **struct**?

What is a Class?

- Examples of classes we’ve already seen:
  - **`string`**
  - **`vector`**
  - **`array`**

- Every class has two parts:
  - an **interface** specifying what operations can be performed on instances of the class (this defines the abstraction boundary)
  - an **implementation** specifying how those operations are to be performed

Classes provide their users with a **public interface** and separate this from a **private implementation**

Abstraction Boundary

Public Interface Available to Users

Private Implementation Behind the Scenes

API: Application Programming Interface

C++ | Containers library | **std::vector**

Element access	
at	access specified element with bounds checking (public member function)
operator[]	access specified element (public member function)
front	access the first element (public member function)
back	access the last element (public member function)
data	direct access to the underlying array (public member function)

```
private:
  // Constant-time move assignment when source object's memory can be
  // moved, either because the source's allocator will move too
  // or because the allocators are equal.
  void
  _M_move_assign(vector&& __x, std::true_type) noexcept
  {
    vector __tmp(get_allocator());
    this->_M_impl._M_swap_data(__tmp._M_impl);
    this->_M_impl._M_swap_data(__x._M_impl);
    std::__alloc_on_move(_M_get_Tp_allocator(), __x._M_get_Tp_allocator());
  }

  // Do move assignment when it might not be possible to move source
  // object's memory, resulting in a linear-time operation.
  void
  _M_move_assign(vector&& __x, std::false_type)
  {
    if (__x._M_get_Tp_allocator() == this->_M_get_Tp_allocator())
      _M_move_assign(std::move(__x), std::true_type());
    else
      {
        // The rvalue's allocator cannot be moved and is not equal,
        // so we need to individually move each element.
        this->assign(std::__make_move_if_noexcept_iterator(__x.begin()),
                     std::__make_move_if_noexcept_iterator(__x.end()));
        __x.clear();
      }
  }
```

Abstraction Boundary

Public Interface Available to Users

Private Implementation Behind the Scenes

API: Application Programming Interface

C++ / Containers library / **std::vector**

Element access	
at	access specified element with bounds checking (public member function)
operator[]	access specified element (public member function)
front	access the first element (public member function)
back	access the last element (public member function)
data	direct access to the underlying array (public member function)

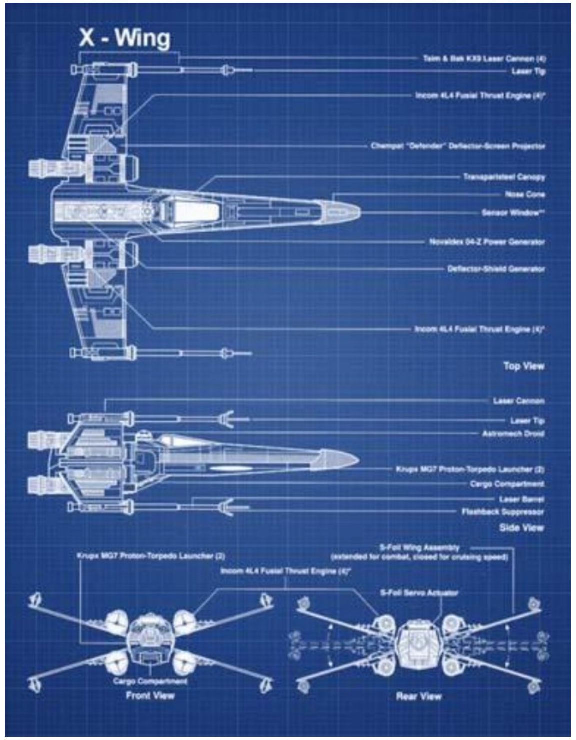
Information Hiding

encapsulation

The process of grouping related information and relevant functions into one unit and defining where that information is accessible.

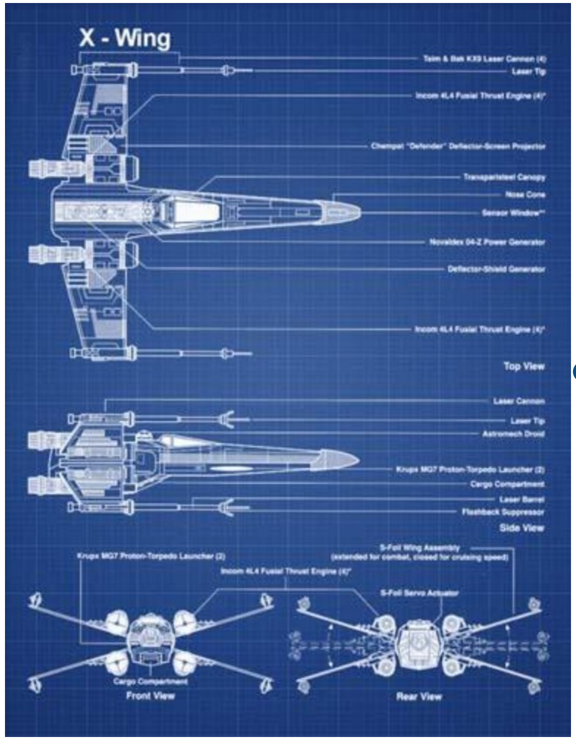
Another way to think about classes…

- A blueprint for a new type of C++ **object**
  - The blueprint describes a general structure



Another way to think about classes…

- A blueprint for a new type of C++ **object**
    - The blueprint describes a general structure
    - We can create specific **instances** of our class using this structure



Another way to think about classes…

- A blueprint for a new type of C++ **object**
  - The blueprint describes a general structure
  - We can create specific **instances** of our class using this structure

instance

When we create an object that is our new type, we call this creating an instance of our class.

Another way to think about classes…

- A blueprint for a new type of C++ **object**
  - The blueprint describes a general structure
  - We can create specific **instances** of our class using this structure

Class	Instance
Student	A specific student at the University of Richmond
University	Richmond University in Richmond, VA, USA
Bank	First National Bank of Richmond

Another way to think about classes…

- A blueprint for a new type of C++ **object**
  - The blueprint describes a general structure
  - We can create specific **instances** of our class using this structure

```
vector<int> numbers;
```

Creates an **instance** of the **`vector`** **class**
(i.e. an object of the type **`vector`**)

How do we design C++ classes?

Three main parts

- Member variables
- Member functions (methods)
- Constructors

Three main parts

- Member variables
  - These are the variables stored within the class
  - Usually not accessible outside the class implementation
- Member functions (methods)
- Constructors

Three main parts

- Member variables

- Member functions (methods)
  - Functions you can call on the object
  - `numbers.push_back(3)`, `numbers.length()`, `numbers.at()`, etc.

- Constructors

Three main parts

- Member variables

- Member functions (methods)

- Constructors
  - Gets called when you create the object
  - `vector<string> mascots;`

Three main parts

- Member variables
  - These are the variables stored within the class
  - Usually not accessible outside the class implementation
- Member functions (methods)
  - Functions you can call on the object
  - `numbers.push_back(3)`, `numbers.length()`, `numbers.at()`, etc.
- Constructors
  - Gets called when you create the object
  - `vector<string> mascots;`

How do we design a class?

We must specify the 3 parts:

1. Member variables: What variables make up this new type?
   - Information associated with the new class of objects

2. Member functions: What functions can you call on a variable of this type?
   - Behavior associated with the new class of objects

3. Constructor: What happens when you make a new instance of this type?

Classes are useful in helping us with complex programs where information and behavior can be grouped into objects.

Design a Toaster Class

1. Member variables: What variables make up this new type?

2. Member functions: What functions can you call on a variable of this type?

3. Constructor: What happens when you make a new instance of this type?

Breakout design activity

We must specify the 3 parts:

1. Member variables: What variables make up this new type?
   - Information associated with the new class of objects

2. Member functions: What functions can you call on a variable of this type?
   - Behavior associated with the new class of objects

3. Constructor: What happens when you make a new instance of this type?

Today

- ~~Classes and OOP~~

- ~~Breakout design activity~~

- Coding a class in C++

- Breakout coding activity

Creating our own class

Classes in C++

- Defining a class in C++ (typically) requires two steps:

Classes in C++

- Defining a class in C++ (typically) requires two steps:
  1. Create a **header file** (typically suffixed with `.h`) describing what operations the class can perform and what internal state it needs

Classes in C++

- Defining a class in C++ (typically) requires two steps:
  1. Create a **header file** (typically suffixed with `.h`) describing what operations the class can perform and what internal state it needs
  2. Create an **implementation file** (typically suffixed with `.cpp`) that contains the implementation of the class

Classes in C++

- Defining a class in C++ (typically) requires two steps:
  1. Create a **header file** (typically suffixed with `.h`) describing what operations the class can perform and what internal state it needs
  2. Create an **implementation file** (typically suffixed with `.cpp`) that contains the implementation of the class

- Clients of the class can then include (using the `#include` directive) the header file to use the class.

Design a Toaster Class

1. Member variables: What variables make up this new type?
   - heat level
   - is it currently toasting

2. Member functions: What functions can you call on a variable of this type?
   - set/get heat level
   - start/stop toasting
   - get toasting status

3. Constructor: What happens when you make a new instance of this type?
   - initial heat level

Header files

What’s in a header?

Toaster.h

Toaster.h > ...

```
#ifndef TOASTER_H
#define TOASTER_H


















#endif
```

This boilerplate code is called a **`#include` guard**. It's used to make sure weird things don't happen if you include the same header twice.

Toaster.h > ...

```
#ifndef TOASTER_H
#define TOASTER_H


class Toaster
{











};

#endif
```

This is a **class definition**. We're creating a new class called **`Toaster`**. Like a **`struct`**, this defines the name of a new type that we can use in our programs.

Toaster.h > ...

```
#ifndef TOASTER_H
#define TOASTER_H


class Toaster
{











};

#endif
```

Don't forget to add the semicolon!

You'll run into some scary compiler errors if you leave it out!

C Toaster.h > ...

```
#ifndef TOASTER_H
#define TOASTER_H


class Toaster
{

public:




private:




};

#endif
```

C Toaster.h > ...

```cpp
#ifndef TOASTER_H
#define TOASTER_H


class Toaster
{

public:




private:




};

#endif
```

The **public interface** specifies what functions you can call on objects of this type.

Think things like the **`vector.length()`** function or the **`string.find()`**

C Toaster.h > ...

```
#ifndef TOASTER_H
#define TOASTER_H


class Toaster
{

public:




private:




};

#endif
```

The **private implementation** contains information that objects of this class type will need in order to do their job properly. This is invisible to people using the class.

C Toaster.h > ...

```
#ifndef TOASTER_H
#define TOASTER_H


class Toaster
{

public:




private:




};

#endif
```

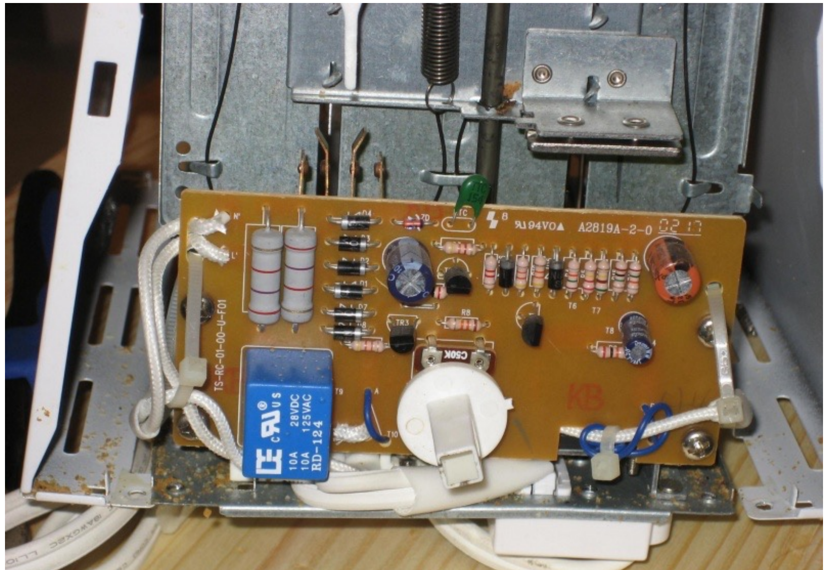
Abstraction Boundary

Toaster.h > ...

```cpp
#ifndef TOASTER_H
#define TOASTER_H


class Toaster
{

public:




private:




};

#endif
```

Public Interface
(What it looks like)

Private Implementation
(How it works)



C Toaster.h > ...

```cpp
#ifndef TOASTER_H
#define TOASTER_H


class Toaster
{

public:
    Toaster(int initialLevel);
    void toast();
    void cancel();
    bool isOn();
    int getLevel();
    void setLevel(int newLevel);
private:



};

#endif
```

The public **member functions** of the `Toaster` class are functions you can call on objects of type `Toaster`.

All member functions must be defined in the class definition. We will implement these functions in the C++ file.

C Toaster.h > ...

```
#ifndef TOASTER_H
#define TOASTER_H


class Toaster
{

public:
    Toaster(int initialLevel);
    void toast();
    void cancel();
    bool isOn();
    int getLevel();
    void setLevel(int newLevel);
private:
    int heatLevel;
    bool isToasting;
    bool isValidLevel(int level);
};

#endif
```

The private **data members** of the **`Toaster`** class. This tells us how the class is implemented. Internally we are storing a heat level and an on/off value for toasting. The only code that can access or modify these values is the **`Toaster`** implementation.

C Toaster.h > ...

```cpp
#ifndef TOASTER_H
#define TOASTER_H


class Toaster
{

public:
    Toaster(int initialLevel);
    void toast();
    void cancel();
    bool isOn();
    int getLevel();
    void setLevel(int newLevel);
private:
    int heatLevel;
    bool isToasting;
    bool isValidLevel(int level);
};

#endif
```

Class definition and name

Methods

Member variables

Toaster.h > ...

```cpp
#ifndef TOASTER_H
#define TOASTER_H


class Toaster
{

public:
    Toaster(int initialLevel);
    void toast();
    void cancel();
    bool isOn();
    int getLevel();
    void setLevel(int newLevel);
private:
    int heatLevel;
    bool isToasting;
    bool isValidLevel(int level);
};

#endif
```

Implementation files

Toaster.cpp > ...

```
#include "Toaster.h"
```

Toaster.cpp > ...

```cpp
#include "Toaster.h"
```

If we are going to implement the **`Toaster`** type, the **`.cpp`** file needs to have the class definition available.

Toaster.cpp > ...

```cpp
#include "Toaster.h"
```

```cpp
class Toaster
{
public:
    Toaster(int initialLevel);
    void toast();
    void cancel();
    bool isOn();
    int getLevel();
    void setLevel(int newLevel);
private:
    int heatLevel;
    bool isToasting;
    bool isValidLevel(int level);
};
```

Toaster.cpp > ...

```cpp
#include "Toaster.h"

Toaster::Toaster(int initialLevel) { setLevel(initialLevel); }
```

The syntax **`Toaster::`** means "look inside of Toaster." The `::` operator is called the scope resolution operator in C++ and is used to say where to look for things.

```cpp
class Toaster
{
public:
    Toaster(int initialLevel);
    void toast();
    void cancel();
    bool isOn();
    int getLevel();
    void setLevel(int newLevel);
private:
    int heatLevel;
    bool isToasting;
    bool isValidLevel(int level);
};
```

Toaster.cpp > ...

```cpp
#include "Toaster.h"

Toaster::Toaster(int initialLevel) { setLevel(initialLevel); }
```

We don’t need to specify where the **`setLevel`** method is. The compiler knows we are inside of **`Toaster`**.

```cpp
class Toaster
{
public:
    Toaster(int initialLevel);
    void toast();
    void cancel();
    bool isOn();
    int getLevel();
    void setLevel(int newLevel);
private:
    int heatLevel;
    bool isToasting;
    bool isValidLevel(int level);
};
```

Toaster.cpp > ...

```cpp
#include "Toaster.h"

Toaster::Toaster(int initialLevel) { setLevel(initialLevel); }

void Toaster::toast() { isToasting = true; }
```

```cpp
class Toaster
{
public:
    Toaster(int initialLevel);
    void toast();
    void cancel();
    bool isOn();
    int getLevel();
    void setLevel(int newLevel);
private:
    int heatLevel;
    bool isToasting;
    bool isValidLevel(int level);
};
```

Toaster.cpp > ...

```cpp
#include "Toaster.h"

Toaster::Toaster(int initialLevel) { setLevel(initialLevel); }

void Toaster::toast() { isToasting = true; }

void Toaster::cancel() { isToasting = false; }
```

```cpp
class Toaster
{
public:
    Toaster(int initialLevel);
    void toast();
    void cancel();
    bool isOn();
    int getLevel();
    void setLevel(int newLevel);
private:
    int heatLevel;
    bool isToasting;
    bool isValidLevel(int level);
};
```

Toaster.cpp > ...

```cpp
#include "Toaster.h"

Toaster::Toaster(int initialLevel) { setLevel(initialLevel); }

void Toaster::toast() { isToasting = true; }

void Toaster::cancel() { isToasting = false; }

bool Toaster::isOn() { return isToasting; }

int Toaster::getLevel() { return heatLevel; }
```

```cpp
class Toaster
{
public:
    Toaster(int initialLevel);
    void toast();
    void cancel();
    bool isOn();
    int getLevel();
    void setLevel(int newLevel);
private:
    int heatLevel;
    bool isToasting;
    bool isValidLevel(int level);
};
```

Toaster.cpp > ...

```cpp
#include "Toaster.h"

Toaster::Toaster(int initialLevel) { setLevel(initialLevel); }

void Toaster::toast() { isToasting = true; }

void Toaster::cancel() { isToasting = false; }

bool Toaster::isOn() { return isToasting; }

int Toaster::getLevel() { return heatLevel; }

void Toaster::setLevel(int newLevel)
{
    if (isValidLevel(newLevel))
    {
        heatLevel = newLevel;
    }
}
```

```cpp
class Toaster
{
public:
    Toaster(int initialLevel);
    void toast();
    void cancel();
    bool isOn();
    int getLevel();
    void setLevel(int newLevel);
private:
    int heatLevel;
    bool isToasting;
    bool isValidLevel(int level);
};
```

Toaster.cpp > ...

```cpp
#include "Toaster.h"

Toaster::Toaster(int initialLevel) { setLevel(initialLevel); }

void Toaster::toast() { isToasting = true; }

void Toaster::cancel() { isToasting = false; }

bool Toaster::isOn() { return isToasting; }

int Toaster::getLevel() { return heatLevel; }

void Toaster::setLevel(int newLevel)
{
    if (isValidLevel(newLevel))
    {
        heatLevel = newLevel;
    }
}

bool Toaster::isValidLevel(int level)
{
    return level >= 1 && level <= 7;
}
```

```cpp
class Toaster
{
public:
    Toaster(int initialLevel);
    void toast();
    void cancel();
    bool isOn();
    int getLevel();
    void setLevel(int newLevel);
private:
    int heatLevel;
    bool isToasting;
    bool isValidLevel(int level);
};
```

Toaster.cpp > ...

```cpp
#include "Toaster.h"

Toaster::Toaster(int initialLevel) { setLevel(initialLevel); }

void Toaster::toast() { isToasting = true; }

void Toaster::cancel() { isToasting = false; }

bool Toaster::isOn() { return isToasting; }

int Toaster::getLevel() { return heatLevel; }

void Toaster::setLevel(int newLevel)
{
    if (isValidLevel(newLevel))
    {
        heatLevel = newLevel;
    }
}

bool Toaster::isValidLevel(int level)
{
    return level >= 1 && level <= 7;
}
```

This use of the **`const`** keyword means "*I promise that this method doesn't change the state of the object.*"

```cpp
class Toaster
{
public:
    Toaster(int initialLevel);
    void toast();
    void cancel();
    bool isOn() const;
    int getLevel() const;
    void setLevel(int newLevel);
private:
    int heatLevel;
    bool isToasting;
    bool isValidLevel(int level);
};
```

Toaster.cpp > ...

```cpp
#include "Toaster.h"

Toaster::Toaster(int initialLevel) { setLevel(initialLevel); }

void Toaster::toast() { isToasting = true; }

void Toaster::cancel() { isToasting = false; }

bool Toaster::isOn() const { return isToasting; }

int Toaster::getLevel() const { return heatLevel; }

void Toaster::setLevel(int newLevel)
{
    if (isValidLevel(newLevel))
    {
        heatLevel = newLevel;
    }
}

bool Toaster::isValidLevel(int level)
{
    return level >= 1 && level <= 7;
}
```

We have to remember to add it into the implementation as well!

```cpp
class Toaster
{
public:
    Toaster(int initialLevel);
    void toast();
    void cancel();
    bool isOn() const;
    int getLevel() const;
    void setLevel(int newLevel);
private:
    int heatLevel;
    bool isToasting;
    bool isValidLevel(int level);
};
```

Toaster.cpp > ...

```cpp
#include "Toaster.h"

Toaster::Toaster(int initialLevel) { setLevel(initialLevel); }

void Toaster::toast() { isToasting = true; }

void Toaster::cancel() { isToasting = false; }

bool Toaster::isOn() const { return isToasting; }

int Toaster::getLevel() const { return heatLevel; }

void Toaster::setLevel(int newLevel)
{
    if (isValidLevel(newLevel))
    {
        heatLevel = newLevel;
    }
}

bool Toaster::isValidLevel(int level)
{
    return level >= 1 && level <= 7;
}
```

```cpp
class Toaster
{
public:
    Toaster(int initialLevel);
    void toast();
    void cancel();
    bool isOn() const;
    int getLevel() const;
    void setLevel(int newLevel);
private:
    int heatLevel;
    bool isToasting;
    bool isValidLevel(int level);
};
```

Breakout coding activity