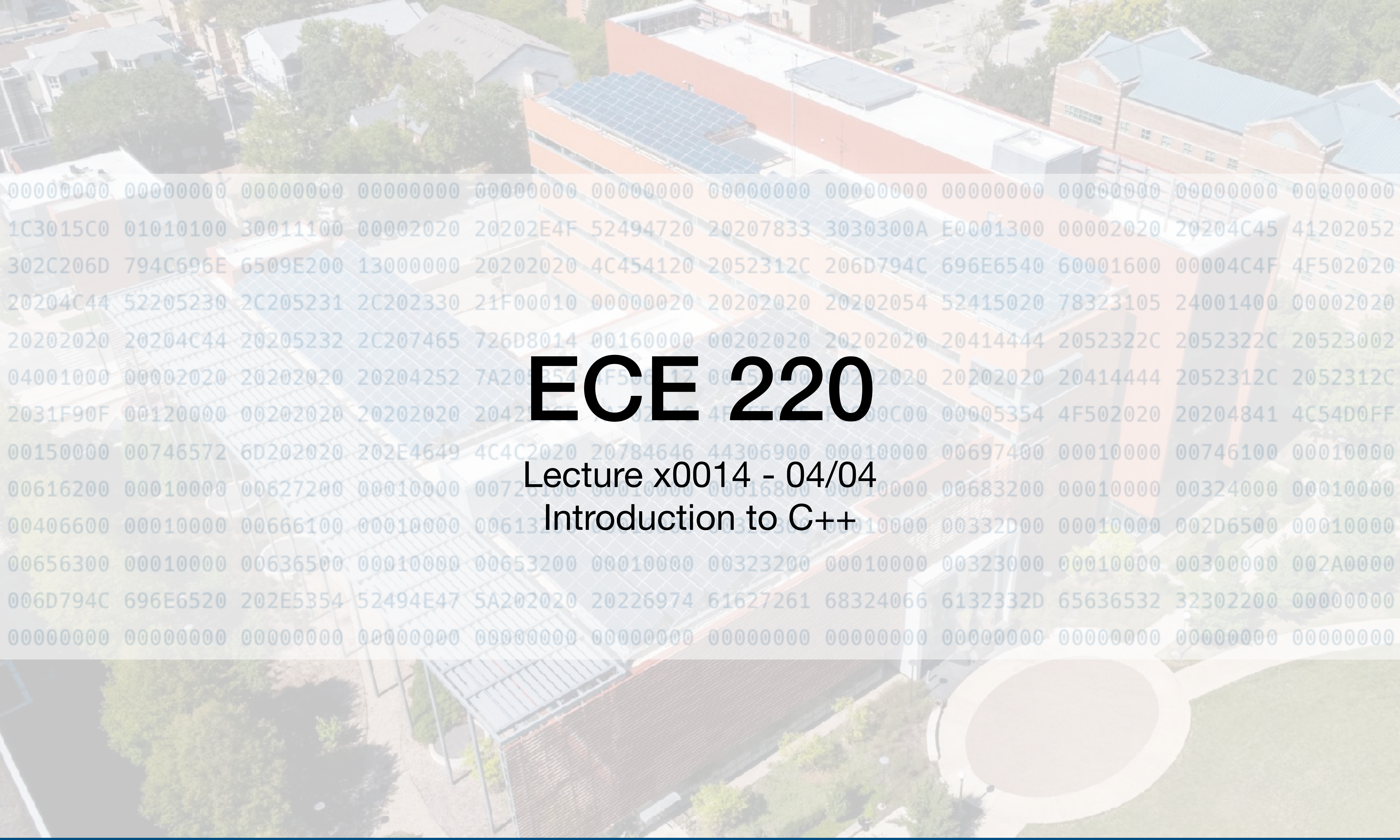


ECE 220

Lecture x0014 - 04/04

Introduction to C++

Recap + reminders

- Last class(es)
  - Doubly linked lists
  - Problem-solving with linked lists
  - Trees & traversal
- This class: Intro to C++

- Reminder(s)
  - MT2 grades have been posted
  - Regrades due by Sunday
  - CBTF reservations are now open for Quiz 5

Hello World!

In the tradition of programmers everywhere, we'll use a "Hello, world!" program as an entry point into the basic features of C++

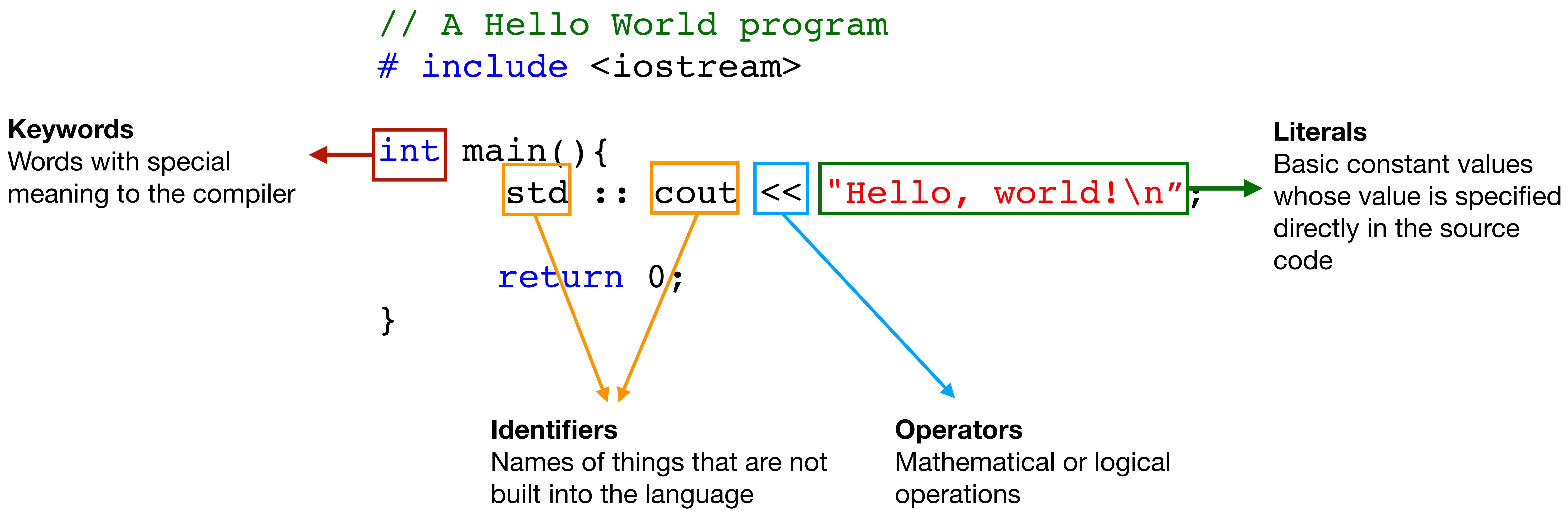
```
// A Hello World program
# include <iostream>

int main(){
      std :: cout << "Hello, world!\n";

      return 0;
}
```

Hello World!

In the tradition of programmers everywhere, we'll use a "Hello, world!" program as an entry point into the basic features of C++



```
// A Hello World program
# include <iostream>

int main(){
     std :: cout << "Hello, world!\n”;

     return 0;
}
```

Keywords
Words with special meaning to the compiler

Literals
Basic constant values whose value is specified directly in the source code

Identifiers
Names of things that are not built into the language

Operators
Mathematical or logical operations

Basic I/O

- `cout <<` - This is the syntax for outputting some piece of text to the screen
- `cin >>` - This is the syntax for inputting values
- **Namespace** - In C++, identifiers can be defined within a context – sort of a directory of names – called a namespace. When we want to access an identifier defined in a namespace, we tell the compiler to look for it in that namespace using the scope resolution operator (`::`).
- For example:

```
std :: cout << "Hello, world!\n”;
```

  Here we’re telling the compiler to look for `cout` in the `std` namespace, in which many standard C++ identifiers are defined (part of `iostream`).

Basic I/O

```
#include <iostream>

using namespace std;

int main(){

        char name[20];
        cout << "Enter your name: ";
        cin >> name;
        cout << "Your name is: " << name << endl;

        return 0;
}
```

Note the lack of `.h` extension. In C++ standard header files have no extensions, but user defined header files *should*.

This is a declaration for convenience. It allows us to not have to specify `std::cout`, `std::cin`, etc.

How do we save/run this file?

File extensions are now `.cpp` rather than `.c`
Use `g++` rather than `gcc` for compilation.

The changes …

- `*.c` became `*.cpp`
- Compiler is now `g++` instead of `gcc`
- `iostream` vs. `stdio.h`
- Functions can have *default* arguments
- Dynamic memory allocation is different
- Functions and operators can be *overloaded*
- Structs get superpowers to become objects via *classes*
- Paradigm change: procedural programming to *object-oriented programming*.
- Etc.

Default arguments

```
float bmi_si(float hcm, float kg){
    return kg / (hcm/100 * hcm/100);
}

float bmi_usa(float hin, float lbs){
    return lbs / (hin * hin) * 703;
}
```

$\longrightarrow$ **C:** Write two functions and use appropriate one depending on units at hand.

C++: Write one function which can accept an optional flag for the rare case an European reports their weight and height in centimeters and kilograms

```
float bmi(float ht, float wt, bool si=false){
    float val = wt/(ht*ht);
    if (si)
        return val*10000;
    else
        return val*703;
}
```

$\downarrow$ **Default value is false**

Dynamic allocation in C & C++

C	C++
Dynamic allocation is accomplished by `malloc`	Dynamic allocation is accomplished by `new`
Deallocation accomplished by `free`	Deallocation accomplished by `delete`
Both `malloc` and `free` are library functions	Both `new` and `delete` are keyword/operators

```
# include <iostream>

int main(){
  int *p;

  // Allocating an integer's worth of space
  p = new int;

  .
  .
  .
  // Deallocating
  delete p;
}
```

How about an array of ints?

Function overloading

- C++ allows multiple functions with the same name but **different** parameters.

- **<u>Note</u>**: The return value cannot be different

    - Why?

```
double volume(float r){
  return 22.0/7*r*r*r*4/3;
}

double volume(float r, float l){
  return 22.0/7*r*r*l;
}

double volume(float w, float h, float l){
  return width * height * length;
}
```

Introduction to classes in C++

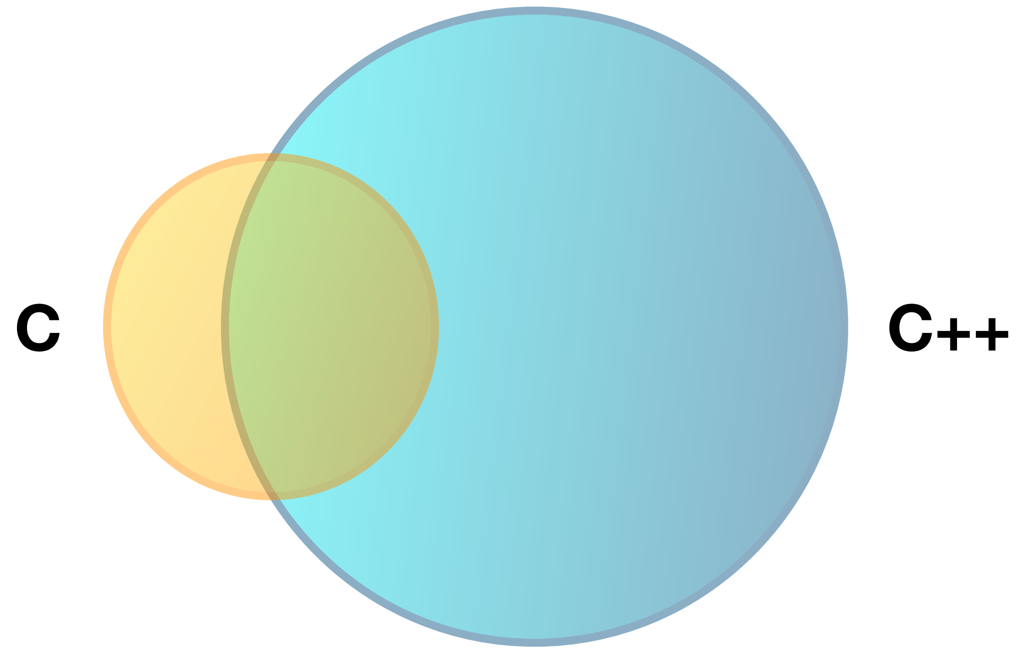
C++: created in 1979 by Bjarne Stroustrup at Bell Labs, as an extension to C.

It's called an **Object Oriented** language.

Object Oriented Programming (OOP)

Programming style associated with *classes* and *objects* and other concepts like

- Encapsulation
- Inheritance $\longrightarrow$ **More next week**
- Polymorphism, etc.



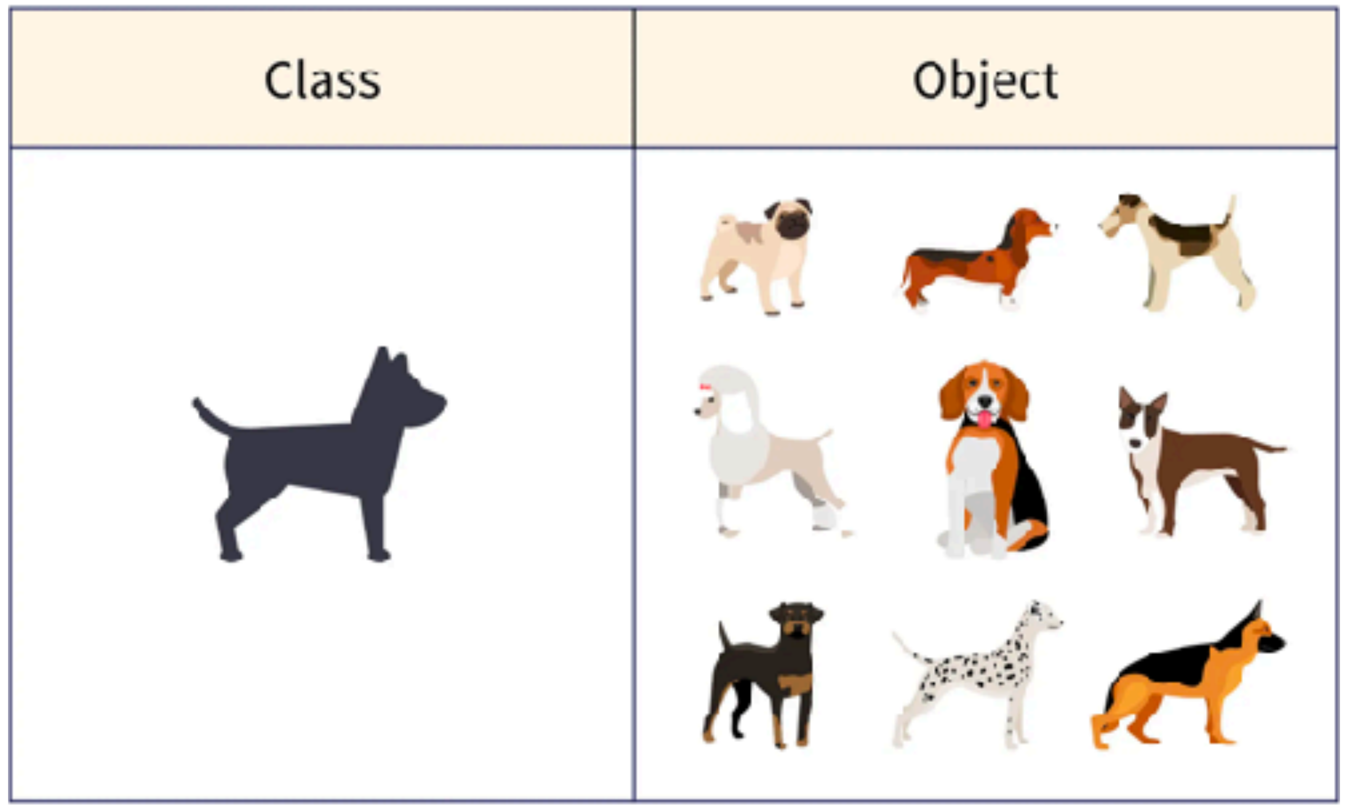
A *class* in C++ is similar to *struct* in C except it defines

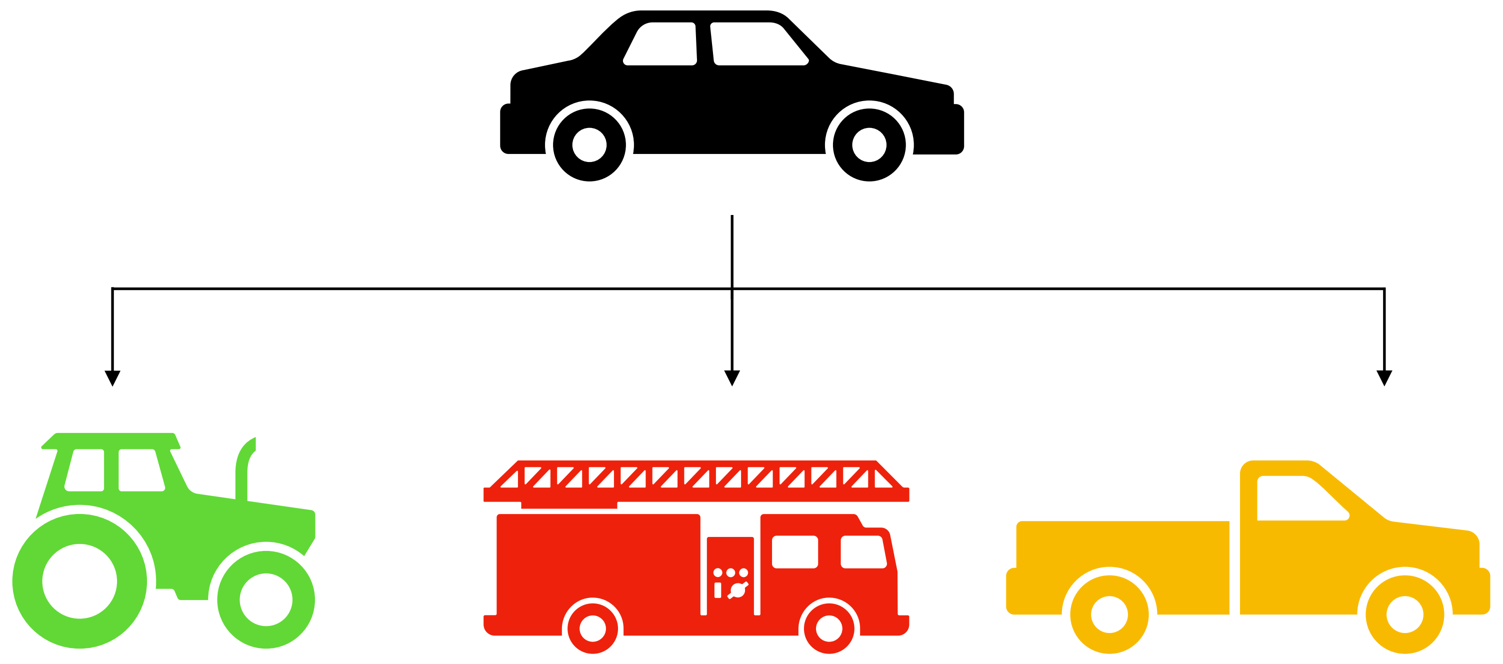
- control "who" can access the data $\longrightarrow$ **Today: classes**
- provide functions specific for the class & its data

Concepts related to classes

An **object** is an *instance* of a class. An object

- shares the same functions with other objects of the same class
- but each object has its **own** copy of the data





Introduction to classes

```
# include <stdio.h>

struct student{
    char name[74];
    unsigned long UIN;
    unsigned int year;
    float GPA;
};

int main(void){
  struct student s1 = {"Garfield", 123456, 6, 3.5};
  printf("%s is an excellent student!\n", s1.name);
  s1.GPA = 1.5;
  printf("Their GPA is %f", s1.GPA);
}
```

Anyone can modify the records!

- Classes provide more structured or granular access to *members*.
  - Two access types, **private** (default) and **public.**
  - Members can also (necessarily) be functions.

Actually in C++ (**but not in C**), structs can also have member functions, but that is an advanced topic.

Introduction to classes

- How to declare an *instance* of a class?

```
# include <iostream>
using namespace std;

class Student{
    char name[74];
    unsigned long UIN;
    unsigned int year;
    float GPA;
};

int main(void){
  Student s1 = {"Garfield", 123456, 6, 3.5};
  cout<<s1.name<<" is an excellent student!"<<endl;
  s1.GPA = 2.4;
  cout<<"Their GPA is "<<s1.GPA<<endl;
}
```

Class members are private. Only the class itself can access them!

Typically this is accomplished using the class *methods.*

Also applies to initialization, i.e. we need to write a class method to initialize an instance.

✗

Constructors

- There are two functions that should be implemented for all classes: ***constructs*** and ***destructors***.
- Constructors are used to initialize instances of a class.
- If we don't declare one, compiler implicitly produces a *default one.*

These are private.

```
class Student{
    char name[74];
    unsigned long UIN;
    unsigned int year;
    float GPA;

public:
  Student(char const *name, unsigned int UIN,
          unsigned int year, float GPA);
};
```

Everything after this will be public.

```
Student::Student(char const *name,
                 unsigned int UIN,
                 unsigned int year,
                 float GPA){
  strcpy(this->name, name);
  this->UIN = UIN;
  this->year = year;
  this->GPA = GPA;
}
```

1. **A constructor has no return type.**
2. **A constructor must have the same name as its class.**

This pointer

- Remember *methods* are shared between **all** instances of a class. However, each instance keeps its **own** copy of the data.

- When we invoke a method on a *particular* object/instance of a class, we need a way to refer to *that* particular instance's copy of the data.

- This is accomplished using the **this** pointer.

```
class Student{
    char name[74];
    unsigned long UIN;
    unsigned int year;
    float GPA;

public:
  Student(char const *name, unsigned int UIN,
        unsigned int year, float GPA);
};

Student::Student(char const *name,
                 unsigned int UIN,
                 unsigned int year,
                 float GPA){
  strcpy(this->name, name);
  this->UIN = UIN;
  this->year = year;
  this->GPA = GPA;
}
```

Constructors

```
class Student{
    char name[74];
    unsigned long UIN;
    unsigned int year;
    float GPA;

public:
  Student(char const *name,
          unsigned int UIN,
          unsigned int year,
          float GPA);

};

Student::Student(char const *name,
                 unsigned int UIN,
                 unsigned int year,
                 float GPA){
  strcpy(this->name, name);
  this->UIN = UIN;
  this->year = year;
  this->GPA = GPA;
}
```

```
int main(void){
  Student s1 = Student(“Garfield", 123456, 6, 3.5);
  cout << s1.name << " is an excellent student!" << endl;
  cout << "Their GPA is: " << s1.GPA << endl;
}
```

Still not correct. We cannot access the private members.

- Solutions?
  - Write a function to print details of a student out.
  - Write *getters* and *setters*.

Getters ...

```
# include <iostream>
using namespace std;

class Student{
    char name[74];
    unsigned long UIN;
    unsigned int year;
    float GPA;

public:
  Student(char const *name,
          unsigned int UIN,
          unsigned int year,
          float GPA);

  float get_GPA();
  char const * get_name();
};
```

```
Student::Student(char const *name, unsigned int UIN,
                 unsigned int year, float GPA){
  strcpy(this->name, name);
  this->UIN = UIN;
  this->year = year;
  this->GPA = GPA;
}

float Student::get_GPA(){
  return this->GPA;
}

char const * Student::get_name(){
  return this->name;
}

int main(void){
  Student s1 = {"Garfield", 123456, 6, 3.5};
  cout<<s1.get_name()<<" is an excellent student!"<<endl;
  cout<<"Their GPA is: "<<s1.get_GPA()<<endl;
}
```

... and setters

```
# include <iostream>
using namespace std;

class Student{
    char name[74];
    unsigned long UIN;
    unsigned int year;
    float GPA;

public:
  Student(char const *name,
          unsigned int UIN,
          unsigned int year,
          float GPA);

  float get_GPA();
  char const * get_name();
  void set_GPA(float gpa);
};
```

```
Student::Student(char const *name, unsigned int UIN,
                 unsigned int year, float GPA){
  name = name;
  UIN = UIN;
  year = year;
  GPA = GPA;
}

float Student::get_GPA(){
  return this->GPA;
}

char const * Student::get_name(){
  return this->name;
}

void Student::set_GPA(float gpa){
  this->GPA = gpa;
}
```

Classes - summary so far …

Member functions

- **Member functions** also called **methods** are functions that are part of a class

Private vs. public members

- private members can only be accessed by member functions (default)
- public members can be accessed by anyone

Constructors & destructor

- special member functions that *creates* and *deletes* an object (when it goes outside of scope)

Summary - constructors

A special method which is invoked automatically at the time of object ***creation***.

- Used to initialize the data members.
- It has the same name as class.
- Two types: default constructor & user defined constructor
- Overloading and default arguments are possible.
- Has no return value; not even `void`.

Destructors

- Destructor is a member function that *destroys* an object.
- It is called automatically when the object goes out of scope.
- It has the same name as class, but prefixed with ~.
- No argument (overloading and default arguments are not possible).
- No return value
- **Primary use:** de-allocate memory!

More on this in the exercise!

Operator overloading

```
#include<iostream>
using namespace std;

class Complex{
  double real;
  double imag;

public:
  Complex(double real, double imag){
    this->real = real;
    this->imag = imag;
  }

  void print(){
    cout<<"(" <<this->real<<" + "<<this->imag<<")";
  }
};
```

Wouldn't it be nice if we could do something like that? →

```
int main(){
  Complex c1 = Complex(2, 4);
  Complex c2 = Complex(3, -5);
  Complex c3 = c1 + c2;
}
```

C++ allows you to *overload* standard operators so that you can use them with your classes.

Operator overloading

```
#include<iostream>
using namespace std;

class Complex{
  double real;
  double imag;

public:
  Complex(double real, double imag){
    this->real = real;
    this->imag = imag;
  }

  void print(){
    cout<<"(" <<this->real<<" + "<<this->imag<<")";
  }
```

```
Complex operator+(Complex c){
    return Complex(this->real + c.real, this->imag + c.imag);
  }
};
```

```
int main(){
  Complex c1 = Complex(2, 4);
  Complex c2 = Complex(3, -5);
  Complex c3 = c1 + c2;
}
```

Just write a function of this form to enable

Exercise(s)

- Overload the multiplication operator to multiply two complex numbers.

- Implement a linked lists in C++ using classes. ← Will do this next time