Object-Oriented Programming

- Classes are the key construct for object-oriented programming
- Object-oriented programming involves a community of interacting objects
- Classes are similar to a struct
  - but has member functions (also called methods) in addition to member variables
Object-Oriented Programming

Important concepts:
• Encapsulation
• Abstraction
• Inheritance (not today)
• Polymorphism (not today)

Classes

Encapsulation
• Separate the interface (what the function does) from the implementation (how it actually does this)
• Should be able to change the implementation without changing the interface
  – Don’t need to change code that uses your code
Classes

Abstraction

• See the big picture without needing to see all the details
• You can work with different levels of abstraction
• We do this in real life (see example on next slide)
Classes

Example: student.hpp

class student {
public:
    void print();
    std::string name;

private:
    int id;
};

Note the order of things:
• (Enforced) Public things first, then private. Why? Public member variables/functions are often all you care about
• (Optional) Within public or private: member functions then member variables
Classes

Example: student.cpp

```cpp
#include <iostream>
#include "student.hpp"

int main(int argc, char** argv)
{
    student s;
    s.print();
}

void student::print() {
    std::cout << "Student name: " << name << std::endl;
    std::cout << "Student id: " << id << std::endl;
}
```

Classes

- The class is a student, while the variable `s` is an object (of type student)
- Think of a class like a cookie cutter
- Think of an object like a cookie
Classes

• **Dot operator** eg. `s.print()` says that print() is a member of object s
  – Used with the object name (ie. s)
• **Scope resolution operator** eg.
  `student::print()` says print() is a member of class student
  – Used with the class name (ie. student)
  – Used inside the class definition

Classes

• **Private**: can only reference this method/variable within the class
  – Why? Control access to these variables/methods
  – Member variables are usually private
• **Public**: can reference this method/variable anywhere
  – This is how other objects interact with your class / object
Constructors

- Suppose you run the code in student.cpp
- You will see something like:

  Student name: 
  Student id: 0

  What just happened??? We will show you in the next slide…

```cpp
int main(int argc, char** argv) {
    student s;
    s.print();
}
```

- This line creates an object of type student.
- To do so, it needs to invoke a constructor for the student class.
- Because we didn’t define any constructors, C++ creates a default one for you, which does nothing and leaves all member variables uninitialized (hence the weird id and name values)
Constructors

- Automatically called when an object of a class is declared
- Used to initialize member variables and other things related to the object

Constructors

Rules
- Must have same name as the class
- Cannot have a return value
- Constructors are public
- If you don’t define any constructors of any kind, C++ will automatically create a default constructor for you that does nothing
- TIP: Always define a default constructor. Letting C++ leave member variables uninitialized leads to lots of errors
Constructors

Examples of constructors (in student.cpp):

```cpp
student::student() {
    id = -1;
    name = NULL;
}
student::student(string nameValue) {
    id = -1;
    name = nameValue;
}
student::student(int idValue, string nameValue) {
    id = idValue;
    name = nameValue;
}
```

The constructor that doesn’t take any arguments is called a default constructor.

You will also need to add these declarations to student.hpp

```cpp
class student {
public:
    student();
    student(string name);
    student(int id, string name);
    void print();
    string name;

private:
    int id;
};
```
Constructors

Alternate (and better) way of defining constructors using the initialization list:

```cpp
student::student() : id(-1), name(NULL) {
}

student::student(string nameValue) : id(-1), name(nameValue) {
}

student::student(int idValue, string nameValue) :
  id(idValue), name(nameValue) {
}
```

Constructors

How do you invoke a constructor? There are multiple ways:

1. Declare a variable of type student

   ```cpp
   student s; // default constructor
   student s("Bob");
   student s(42,"Bubba");
   ```
Constructors

2. Explicit constructor call
   student s;
   s = student(1,"Adam");

3. Memory allocation
   student* s = new
   student(2,"Eve");

Accessors / Mutators

- **Accessor functions**: get the values of member variables
  - Usually prefixed by get eg. getName()
- **Mutator functions**: set the values of member variables
  - Usually prefixed by set eg. setName(name)
Accessors / Mutators

Accessors (example)

```cpp
int student::get_id() {
    return id;
}

string student::get_name() {
    return name;
}
```

Mutators (example)

```cpp
void student::set_name(string newname) {
    name = newname;
}
```

Note: if we don’t want to allow the student id to change, we choose NOT to create a mutator for it
**Accessors / Mutators**

This is what the header file now looks like

```cpp
class student {
public:
    student();
    student(string name);
    student(int id, string name);
    int get_id();
    string get_name();
    void set_name(string newname);
    void print();

private:
    int id;
    char* name;
};
```

**Inline Functions**

- For short functions, you can declare them as inline functions
- The code for an inline function declaration is inserted at each spot where the function is called
- More efficient: saves the cost of calling a function
Inline Functions

class student {
    public:
        int get_id() { return id; }
        string get_name() { return name; }
        // etc.
    private:
        int id;
        char* name;
};

Notice the semi-colon is missing at the end of the brace

Const

bool student::isOlder(student s1, student s2) {
    return(s1.id > s2.id);
}

- The isOlder function above doesn’t need to alter the values of s1 and s2
- In cases where we don’t modify the parameters, we can declare them const
- Why? Form of error-checking and also more efficient
- Commonly used for call-by-reference parameters for large objects eg. classes
**Const**

```cpp
bool student::isOlder(const student& s1, const student& s2) {
    return(s1.id > s2.id);
}
```

You also need the header file to have the declaration:

```cpp
class student {
public:
    bool isOlder(const student& s1, const student& s2);
    // etc..
};
```

---

**Const**

Ways to use `const`:

1. Declare that a function doesn’t change a parameter
   
   This was the previous example

2. Declare that a function doesn’t change an object (eg. its member variables)
   
   We can make the print function `const` (see next slide for example)
void student::print() const {
    std::cout << "Student name: " << name <<
    std::endl;
    std::cout << "Student id: " << id <<
    std::endl;
}

You also need the header file to have the declaration:

class student {
public:
    void print() const;
    // etc.
private:
};

Static Variables

• A static variable is shared by all objects of a class
  – Eg. if we wanted to keep track of the number of student objects created, we can associate a num_students static variable

• A static variable is associated with a class (not an object)
  – Eg. student::num_students

• Need to initialize it outside the class (yes, even private static variables)
  – Can only initialize it once
class student {
public:
    // etc.
private:
    int id;
    string name;
    static int num_students;
};

int student::num_students = 0;

/* This now guarantees that the student id is unique for each student */
student::student() : name(NULL) {
    id = student::num_students;
    student::num_students++;
}

student::student(string nameValue) :
    name(nameValue) {
    id = student::num_students;
    student::num_students++;
}
Static functions

• You can also have static functions which are associated with the class
• A static function is not allowed to use any member variables or functions that depend on the calling object
  – ie. cannot use nonstatic member variables or nonstatic member functions