Classes
user-defined types

Michael R. Nowak
Texas A&M University

Motivating example

- Let's assume that we're reading in RGB values from a formatted file
  - At this point in the semester, we've seen how we could store these values using:
    - Three parallel vector<int>s
    - A vector<vector<int>>

```
string iname = "rgb_data.dat";
ifstream ist {iname};
if (!ist) throw runtime_error("can't open input
                               file");
vector<int> R; int r;
vector<int> G; int g;
vector<int> B; int b;
while (ist >> r >> g >> b) {
    R.push_back(r);
    G.push_back(g);
    B.push_back(b);
}
```

Motivating example

- Types are good for directly representing ideas in code
  - When we want to do
    - Integer arithmetic, `int` is a great help
    - Manipulate text, `std::string` is a great help
  - Types are helpful because they provide
    - **Representation**: A type "knows" how to represent the data needed in an object
    - **Operations**: A type "knows" what operations can be applied to objects

Motivating example

- The concept of a `color` follows this pattern:
  - A specific `color` is **represented** by three integer values
  - We can also perform various **operations** on `color`s, the result of which depends on the state of the object(s) to which it is applied; consider that,
    - We could blend two `color`s together; result depends on colors blended
    - We could update a respective color's `r`, `g`, and `b` values
    - etc.
- We would like to **represent** an **abstraction** of our notion of **color** as a *data structure* along with a *set of functions* that perform color **operations**
- The **class** language construct in the C++ language yields an ability to introduce **user-defined types** into our programs; we will leverage this construct to write a **user-defined type** `color` in this lecture

What exactly is a class?

- A class directly represents a concept in a program
  - If you can think of "it" as a separate entity, it is plausible that it could be a class or an object of a class
  - Examples: vector, matrix, input stream, string, FFT, valve controller, robot arm, device driver, picture on screen, dialog box, graph, window, temperature reading, clock
- A class is a **user-defined type**: it is composed of built-in types, other user-defined types (composition), and functions
  - Recall that a **type** simply defines a set of possible values and a set of operations for an **object**
  - That an **object** is simply some memory that holds a **value** of a given **type**
  - And that a **value** is a set of bits in memory that is interpreted according to some type

What exactly is a class?

- When we introduce a **user-defined type** to our program, a **class** provides the description for how **objects** of that **type** are to be **represented** and specifies the **operations** that can be applied to them
  - It is a **blueprint** from which **objects** are **created**, **used**, and **destroyed**

What exactly is a class?

- A **class** provides the description for how **objects** of that **type** are to be **represented**
  - The representation of the **user-defined type** is composed of built-in types and other user-defined types that are known together as data members
- To introduce a **user-defined type** for `color`, we would **declare** a **user-defined type** that contained three `integer` **data members** that would maintain a respective `color` **object**'s RGB values
  - An imperfect analogy for this would be an excel spreadsheet:
    - The definition of a table is denoted by the header columns, which provides a description for each field in each row of that column along with its data type; the column headers (metadata) is in a narrow sense like a class
    - Each row has its own storage field for each header column and stores its respectively associated data in that field; each row (data) is in a narrow sense like an object

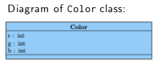
What exactly is a class?

- A **class** also specifies the operations that will be able to be applied **objects** of the **user-defined type**
  - Function members are written to provide the operations that we will be able to apply to the **objects** of our **user-defined type**
- When defining our **user-defined type** for `color`, we would also include the declaration of a number of function members defining the possible **operations** on `color` **objects**; this might include functions supporting:
  - The addition of two `color` **objects**
  - The modification of a `color` **object**'s RGB values
  - etc.

What exactly is a class?

- It is common to refer to the **representation** and **operations** for a **user-defined type** as its **attributes** and **behaviors** respectively
- For now, let's only focus on the attributes of the `color` class that we will build-up over the course of this lecture

Diagram of `Color` class:



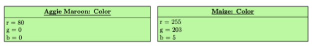
- At this point, simply think of a **class** as a template from which **objects** of that **class-type** can be created from

What exactly is an object?

- With respect to **user-defined types**, an **object** is simply an instance of the **user-defined type** from which it was **instantiated**

Diagram of `Color` class:

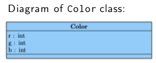
Diagrams of objects instantiated from `Color` class:



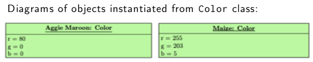
What exactly is an object?

- The values stored within the **data members** represent a respective **object**'s current **state**
  - It is the **attributes** of an **object** that *differentiates* one object **from** another of the *same* **type**
- Accordingly, each `color` **object** instance will have its *own memory space* to store its *own set of data members*

Diagram of `Color` class:



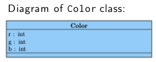
Diagrams of objects instantiated from `Color` class:



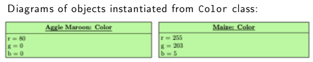
What exactly is an object?

- The **behaviors** detail what objects of a respective **class-type** can do
  - These **member functions** are not stored separately for each instance of a class
  - Instead, an *implicit instance argument* is passed when a **member function** is *called*; this *ensures* that the function is *invoked* on the *appropriate object*
- Each `color` **object** will have the same **behaviors** as all the other `color` typed **objects**

Diagram of `Color` class:



Diagrams of objects instantiated from `Color` class:



Why classes?

- One of the primary advantages of defining user-defined types is that their instances conduct themselves in nearly the same way as the built-in types
  - The objects instantiated from them follow practically the same rules as the built-in types for naming, scope, lifetime, etc.
  - The objects instantiated from them undergo static type checking at compile-time; dynamic type checking, as warranted, at run-time
  - As a type, they define the operations that can be applied to the objects instantiated from them, as well as context for common operations (such as '+', '−', etc.)

Why classes

- **Data abstraction** allows us to focus on **what** operations that will be performed on the data members **opposed to how** we will perform those operations; hidden is the underlying structure, increased is modularity and transparency
- **Encapsulates** data together with the operations that can be performed on that data
- **Data hiding** can be accomplished using member access specifiers: restrict interaction with class members across a well-defined public interface; present only the fundamental facilities that the user needs for use, and hide all implementation details within the class itself
  - Provide the user with the precise interface required to complete the job; keep the public interface to a minimum
  - A change to the implementation should not require a change to the user's code

classes and structs

- Class members are private by default:

```
class X {
    int mf();
    // …
};
```

- Means

```
class X {
private:
    int mf();
    // …
};
```

- So

```
X x;            // variable x of type X
int y = x.mf(); // error: mf is private (i.e., inaccessible)
```

classes and structs

- A struct is a class where members are public by default:

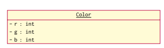
```
struct X {
    int m;
    // …
};
```

- Means

```
struct X {
public:
    int m;
    // …
};
```

- structs are primarily used for data structures where the members can take any value

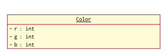
Writing a class



Color.h

```
#ifndef COLOR_H
#define COLOR_H

class Color {
public: // public access specifier

    // Declarations written here comprise Color's interface
    // Any functions declared under the public access specifier will be accessible to functions both
         inside and outside of the class

private: // private access specifier
    // Declarations written here comprise Color's implementation
    // The data members and member functions declared with private access specification
        will only be accessible by functions that are members of the class
};

#endif
```

Writing a class



Color.cpp

```
#include "Color.h"

/* definitions go here for all member functions longer than one line (irrespective of access
specifier) */
```

main.cpp

```
#include "Color.h"

int main ()
{
    Color c;
    /* do something */
    return 0;
}
```

Members and member access

- One way of looking at a class:

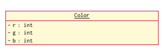
```
class X {   // this class' name is X
    // data members (they store information)
    // function members (they do things, using the information)
};
```

- Example

```
class X {
public:
    int m;                                             // data member
    int mf(int v) { int old = m; m = v; return old; } // function member
};

X var;              // var is a variable of type X
var.m = 7;          // access var's data member m
int x = var.mf(9);  // call var's member function mf()
```

Writing a class : data members

```
sketch.cpp: In function 'int main()':
sketch.cpp:21:7: error: 'int Color::r' is private within this context
     c.r = 80;
       ^
sketch.cpp:13:9: note: declared private here
     int r;
         ^
sketch.cpp:22:7: error: 'int Color::g' is private within this context
     c.g = 0;
       ^
sketch.cpp:14:9: note: declared private here
     int g;
         ^
sketch.cpp:23:7: error: 'int Color::b' is private within this context
     c.b = 0;
       ^
sketch.cpp:15:9: note: declared private here
     int b;
         ^
```



Color.h

```
#ifndef COLOR_H
#define COLOR_H
class Color {
public:

private:
    int r;
    int g;
    int b;
};
#endif
```

main.cpp

```
#include "Color.h"
int main ()
{
    Color c;
    c.r = 80;
    c.g = 0;
    c.b = 0;

    return 0;
}
```

Color.cpp

```
#include "Color.h"
```

Why bother with the public/private distinction?

- Why not make everything public?
  - To provide a clean interface
    - Data and messy functions can be made private
  - To maintain an invariant
    - Only a fixed set of functions can access the data
  - To ease debugging
    - Only a fixed set of functions can access the data
    - (known as the “round up the usual suspects” technique)
  - To allow a change of representation
    - You need only to change a fixed set of functions
    - You don’t really know who is using a public member

Data members, public or private?

- Should an attribute be public or private?
  - If the value assigned to an object will work regardless, you could declare that data member under the public access specifier
  - If the value assigned to an object needs to be checked, or must conform to some requirements, it should be declared under the private access specifier

Private data members

- Making attributes private can help maintain the integrity of our data members by inhibiting the direct manipulation of their values; interactions with private data members are limited to the extent provided by the public interface
- If you want private attributes to be theoretically "public",
  - Then provide public mutators/accessors that can mitigate setting/getting those values

Accessors and mutators

- An **accessor** is a function that returns the value stored in a private data member
- A **mutator** is a function that stores a value in a private data member or mutates its state

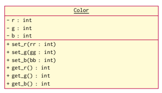
Writing a class : accessors and mutators

Color.h

```
#ifndef COLOR_H
#define COLOR_H
class Color {
public:
    void set_r(int);
    void set_g(int);
    void set_b(int);
     int get_r() const;
     int get_g() const;
     int get_b() const;
private:
    int r;
    int g;
    int b;
};
#endif
```

Color.cpp

```
#include "Color.h"
void Color::set_r(int rr) { r = rr; }
void Color::set_g(int gg) { g = gg; }
void Color::set_b(int bb) { b = bb; }
int Color::get_r() const { return r; }
int Color::get_g() const { return g; }
int Color::get_b() const { return b; }
```




```
~/Desktop
% ./a.out
80      0       0
```

main.cpp

```
#include "Color.h"

int main ()
{
    Color c;
    c.set_r(80);
    c.set_g(0);
    c.set_b(0);
    cout << c.get_r()
         << '\t'
         << c.get_g()
         << '\t'
         << c.get_b()
         << endl;
    return 0;
}
```

Defining member functions

- When writing a member functions for a class,
  - You must provide a declaration for that function within the class declaration (which should be written in the class's header file, `ClassName.h`)
    - If the member function is part of the interface, its declaration should be written under the public access specifier
    - If the member function is part of the implementation, its declaration should be written under the private access specifier
  - Unless the member function should be inlined, define the behavior after the class declaration, preferably inside the class's source file (`ClassName.cpp`)
    - Don't forget that you must prefix the function's name with the class name followed by the scope resolution operator (`::`)

Using private member functions

- A private member function can only be called by other member functions
- Private member functions are commonly used to implement aspects of the public interface and for internal processing completed by the class
  - It is encouraged to have private member functions in your classes; you should keep the public interface as minimal as possible
    - Use private member functions to break code up into conceptual units, such that each function does only one thing

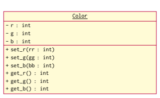
Using `const` with member functions

- Directly proceeding the call operator in both the declarations and definitions of my accessors, you will see the keyword `const`
  - When `const` appears after directly after the call operator, it specifies that that function will not change the state of the object for which it is called

```
int Color::get_r() const { return r; }
```

  - If you're interacting with a constant reference to an object or a constant object, you will only be able to call member functions that are marked `const`

Writing a class : `inline` accessors and mutators




```
~/Desktop
% ./a.out
80      0       0
```

Color.cpp

```
#include "Color.h"
```

Color.h

```
#ifndef COLOR_H
#define COLOR_H
class Color {
public:
    void set_r(int rr) { r = rr; }
    void set_g(int gg) { g = gg; }
    void set_b(int bb) { b = bb; }
    int get_r() const { return r; }
    int get_g() const { return g; }
    int get_b() const { return b; }
private:
    int r;
    int g;
    int b;
};
#endif
```

main.cpp

```
#include "Color.h"

int main ()
{
    Color c;
    c.set_r(80);
    c.set_g(0);
    c.set_b(0);
    cout << c.get_r() << '\t'
         << c.get_g() << '\t'
         << c.get_b() << endl;
    return 0;
}
```

Inline member functions

- When you provide a definition for a member function inside of a class declaration (as I did in the code presented on the previous slide)
  - You're asking the compiler to substitute the body of the function inline at each call to it, in hope of saving the overhead of a function call
    - This is appropriate for simple functions, with short bodies
  - Therefore, when the function definition `int get_r() const {return r;}` is provided inside the class declaration,
    - A function call to `get_r()`, may then be inlined by the compiler by copying the function's code in place of the function call

Writing a class : adding more behaviors

main.cpp

```cpp
#include "Color.h"
int main ()
{
    Color c;
    c.set_r(80);
    c.set_g(0);
    c.set_b(0);
    c.to_cout();

    Color c2;
    c2.set_r(35);
    c2.set_g(80);
    c2.set_b(185);
    c2.to_cout();

    Color c3 = c.blend(c2);
    c3.to_cout();

    return 0;
}
```

Color.h

```cpp
#ifndef COLOR_H
#define COLOR_H

class Color {
public:
    void set_r(int rr) { r = rr; }
    void set_g(int gg) { g = gg; }
    void set_b(int bb) { b = bb; }
    int get_r() const { return r; }
    int get_g() const { return g; }
    int get_b() const { return b; }
    Color blend(Color const&);
    bool is_gray() const;
    void to_cout() const;
private:
    int r;
    int g;
    int b;
};
#endif
```

```
Color
- r : int
- g : int
- b : int
+ set_r(rr : int)
+ set_g(gg : int)
+ set_b(bb : int)
+ get_r() : int
+ get_g() : int
+ get_b() : int
+ blend(other : Color const&) : Color
+ is_gray() : bool
+ to_cout()
```

```
~/Desktop
% ./a.out
c       80      0       0
c       35      80      185
c       57      40      92
```

Color.cpp

```cpp
#include "Color.h"
Color Color::blend(Color const&
                              other)
{
    Color c_blended;
    c_blended = (other.r + r) / 2;
    c_blended = (other.g + g) / 2;
    c_blended = (other.b + b) / 2;
    return c_blended;
}
void Color::to_cout() const
{
    cout << "c\t" << r << '\t' << g
         << '\t' << b << endl;
}

bool Color::is_gray() const {
    return (r == g && g == b);
}
```

Implied Attributes

- We store most of an instantiated object's data as an attribute; however, there are situations where the data is better off computed than stored
  - For instance, our `color` class defines a member function `is_gray()` that returns whether the color stored in an object is grayscale or not
  - If we stored this "attribute" as a data member, we would have to update its value each time an RGB value was changed

Object initialization

- Recall that when we define local variables of the primitive built-in types, they are not automatically initialized for us; instead, they take on whatever value is in left-over in the memory that they occupy
  - The data members of an instantiated `color` object (r, g, and b) are type int; they are not initialized when we instantiate a `color` object
  - When I wrote the r, g, and b values of a color object in my program to standard output, before setting its values, I observed the following:

```
~/Desktop
% ./a.out
         r              g              b
1352205056          32767     1352205056
```

Object initialization

- The uninitialized nature of the variables in objects instantiated from our user-defined types is problematic
- When we instantiate an object from a user-defined type it should be initialized to a valid state
  - For an object instantiated from `color`, we want r, g, and b to each be initialized with a valid value (between 0 and 255)
  - We would then write our member functions to ensure that the object's valid state is maintained throughout its lifetime
- We strive to design our types so that values are guaranteed to be valid
  - A rule for what constitutes a valid value is called an "invariant"

Constructors

- In order to initialize our data members upon object instantiation, we write a "special" member function known as a constructor
  - The constructor is implicitly invoked whenever an object of the user-defined type is created
    - Its job is to construct an object and do initialization if necessary
  - We can also acquire resources in the constructor; perhaps we allocate dynamic memory for some object or maybe even open a file
  - When we declare a constructor in our class declaration, we are declaring a function that is:
    - identified by the same name as the class;
    - has no return type specified; and,
    - has a parameter list with zero-or-more items

Constructors

- If we do not write a constructor for a class, the compiler will provide our class with a default constructor
  - The compiler-generated default constructor calls the default constructor on all of the user-defined type data members
  - This is the constructor that has been used when we instantiate a color object with the declaration `Color c;`
- It is important that you write your own default constructor when needed; you know better than the compiler how to go about object construction and data member initialization

Default constructor

Color.h

```cpp
#ifndef COLOR_H
#define COLOR_H
class Color {
public:
    Color();
private:
    int r; int g; int b;
};
#endif
```

Color.cpp

```cpp
#include "Color.h"

Color::Color() : r{0}, g{0}, b{0} {}
```

Constructor overloading

- A constructor is a member function; a class introduces a scope
  - Functions that have the same name but different parameter lists and appear in the same scope are overloaded
  - Therefore, it follows that we can overload the constructors of a user-defined type; a class can have more than one constructor
    - We can pass arguments to a parameterized constructors and those arguments can be used to initialize data members

Parameterized constructors

Color.h

```cpp
#ifndef COLOR_H
#define COLOR_H
class Color {
public:
    Color();
    Color(int, int, int);
private:
    int r; int g; int b;
};
#endif
```

Color.cpp

```cpp
#include "Color.h"

Color::Color() : r{0}, g{0}, b{0} {}
Color::Color(int r, int g, int b) :
r{r}, g{g}, b{b} {}
```

Parameterized constructors

- Once you've defined a parameterized constructor, you can pass arguments to that constructor and use them for the initialization of an instantiated object

```
Color c{80,0,0};
```

Color.h

```
#ifndef COLOR_H
#define COLOR_H
class Color {
public:
    Color();
    Color(int, int, int);
private:
    int r; int g; int b;
};
#endif
```

Color.cpp

```
#include "Color.h"

Color::Color() : r{0}, g{0}, b{0} {}
Color::Color(int r, int g, int b) : r{r},
g{g}, b{b} {}
```

```
// 1st Pre C++11
```

Color.cpp

```
#include "Color.h"

Color::Color()
{
    r = 0;
    g = 0;
    b = 0;
}

Color::Color(int r, int g, int b)
{
   this->r = r;
   this->b = b;
   this->c = c;
}
```

Values set twice.

1. Initialization via default constructor:
   a. r is set to 0
   b. g is set to 0
   c. b is set to 0
2. Assignment in constructor with parameters:
   a. r is set to value passed as parameter
   b. g is set to value passed as a parameter
   c. b is set to value passed as a parameter

```
// 2nd C++ 11 and later
```

Color.cpp

```
#include "Color.h"

Color::Color() : r{0}, g{0}, b{0} {}

Color::Color(int r, int g, int b) : r{r},
g{g}, b{b} {}
```

Values set once.

1. Initialization in constructor with parameters
   a. r is set to value passed as a parameter
   b. g is set to value passed as a parameter
   c. b is set to value passed as a parameter

Classes without a default constructor

- If we define any constructor for a class (whether its parameterized or not), the compiler will NOT generate a default constructor
  - If we define a parameterized constructor for a class, but not a default constructor, we will not be able to instantiate an object without providing the necessary arguments
    - If we decided not to write a default constructor for `Color`, we would not be able to declare a `vector<Color> colors(100)`
      - `Color`, in this case, would not be default constructible; therefore, we would need to explicitly initialize all 100 elements

Destructors

- The destructor is another "special" member function and is automatically called when an object's lifetime is up
  - One of the primary uses of the destructors is to release resources that the object had acquired during construction
    - For instance, if the constructor allocated dynamic memory, the destructor should deallocate that memory
    - Likewise, if the object acquired some resource from somewhere during construction, it should release that resource as it is being torn down
  - There is only one destructor per class (it cannot be overloaded)

Destructors

Color.h

```
#ifndef COLOR_H
#define COLOR_H
class Color {
public:
    ~Color();
private:
    int r; int g; int b;
};
#endif
```

Color.cpp

```
#include "Color.h"

Color::~Color() {}
```

What makes a good interface?

- Minimal
  - As small as possible
- Complete
  - And no smaller
- Type safe
  - Beware of confusing argument orders
  - Beware of over-general types (e.g., int to represent a month)
- Const correct

Interfaces and "helper functions"

- Keep a class interface (the set of public functions) minimal
  - Simplifies understanding
  - Simplifies debugging
  - Simplifies maintenance

- When we keep the class interface simple and minimal, we need extra "helper functions" outside the class (non-member functions)
  - E.g. == (equality) , != (inequality)

Helper functions

- Helper functions are a design concept, not a programming language construct
  - A helper function will (usually) takes arguments of the classes for which they are helpers of

- A function that can be simply, elegantly, and efficiently implemented as a non-member function should be implemented as a helper function
  - This will keep that function from accessing an instantiated object's representation; the function cannot directly corrupt the data in a class
  - If the representation changes, only the functions that directly access the representation need to be rewritten

Operator overloading

- You can define almost all C++ operators (but can't create your own) for user-defined type operands
  - You can define only existing operators
    - E.g., `+ - * / % [] () ^ ! & < <= > >=`
  - You can define operators only with their conventional number of operands
    - E.g., no unary `<=` (less than or equal) and no binary `!` (not)
  - An overloaded operator must have at least one user-defined type as operand
    - `int operator+(int,int); // error: you can't overload built-in +`
    - `Vector operator+(const Vector&, const Vector &);    // ok`
  - Advice (not language rule):
    - Don't overload unless you really have to
    - Overload operators only with their conventional meaning
    - + should be addition, * be multiplication, [] be access, () be call, etc.

References

- Lippman, B., Lajoie, Josee, & Moo, B. E. (2016). *C++ primer* (5th ed.). Addison-Wesley.
- Stroustrup, B. (2014). *Programming: principles and practice using C++* (2nd ed.). Addison-Wesley.