OOP Classes & Objects

Dr. Anwar Shah, PhD, MBA(HR)
Assistant Professor in CS
FAST National University of Computer and Emerging Sciences CFD

Object-Oriented Programming – Classes and Objects

- What is Object-Oriented Programming?
- What are Classes and Objects?
- Declaring Classes and creating Objects
- Dot and pointer operators
- `public` and `private` access modifiers
- Methods, Constructors and Destructors
  - `class` methods
  - default and overloaded constructors
  - copy and move constructors
  - shallow vs. deep copying
  - `this` pointer

- `static` class members
- `struct` vs. `class`
- `friend` of a class

What is Object Oriented Programming

What is Object-Oriented Programming?

- Procedural Programming
- Procedural Programming limitations
- OO Programming concepts and their advantages
- OO Programming limitations

Procedural programming

- Focus is on processes or actions that a program takes
- Programs are typically a collection of functions
- Data is declared separately
- Data is passed as arguments into functions
- Fairly easy to learn

Procedural programming - Limitations

- Functions need to know the structure of the data.
  - if the structure of the data changes many functions must be changed

- As programs get larger they become more:
  - difficult to understand
  - difficult to maintain
  - difficult to extend
  - difficult to debug
  - difficult to reuse code
  - fragile and easier to break

What is Object-Oriented Programming?

- Classes and Objects
  - focus is on classes that model real-world domain entities
  - allows developers to think at a higher level of abstraction
  - used successfully in very large programs

What is Object-Oriented Programming?

- Encapsulation
  - objects contain data AND operations that work on that data
  - Abstract Data Type (ADT)

What is Object-Oriented Programming?

- Information-hiding
  - implementation-specific logic can be hidden
  - users of the class code to the interface since they don't need to know the implementation
  - more abstraction
  - easier to test, debug, maintain and extend

What is Object-Oriented Programming?

- Reusability
  - easier to reuse classes in other applications
  - faster development
  - higher quality

What is Object-Oriented Programming?

- Inheritance
  - can create new classes in term of existing classes
  - reusability
  - polymorphic classes

- Polymorphism and more...

Limitations

- Not a panacea
  - OO Programming won't make bad code better
  - not suitable for all types of problems
  - not everything decomposes to a class
- Learning curve
  - usually a steeper leaning curve, especially for C++
  - many OO languages, many variations of OO concepts
- Design
  - usually more up-front design is necessary to create good models and hierarchies
- Programs can be:
  - larger in size
  - slower
  - more complex

What are Classes and Objects?

Classes and Objects

- **Classes**
  - blueprint from which objects are created
  - a user-defined data-type
  - has attributes (data)
  - has methods (functions)
  - can hide data and methods
  - provides a public interface

- **Example classes**
  - Account
  - Employee
  - Image
  - std::vector
  - std::string

Classes and Objects

- **Objects**
  - created from a class
  - represents a specific instance of a class
  - can create many, many objects
  - each has its own identity
  - each can use the defined class methods

- **Example Account objects**
  - Frank's account is an instance of an Account
  - Jim's account is an instance of an Account
  - Each has its own balance, can make deposits, withdrawals, etc.

Classes and Objects

```
int high_score;
int low_score;

Account frank_account;
Account jim_account;

std::vector<int> scores;
std::string name;
```

Declaring Classes and Creating Objects

Declaring a Class

```
class Class_Name
{

  // declaration(s);

};
```

Player

```
class Player
{
    // attributes
    std::string name;
    int health;
    int xp;

  // methods
  void talk(std::string text_to_say);
    bool is_dead();
};
```

Creating objects

```
Player frank;
Player hero;


Player *enemy = new Player();
delete enemy;
```

Account

```
class Account
{
    // attributes
    std::string name;
    double balance;

    // methods
    bool withdraw(double amount);
    bool deposit(double amount);
};
```

Creating objects

```
Account frank_account;
Account jim_account;


Account *mary_account = new Account();
delete mary_account;
```

Creating objects

```
Account frank_account;
Account jim_account;

Account accounts[] {frank_account, jim_account};

std::vector<Account> accounts1 {frank_account};
accounts1.push_back(jim_account);
```

Accessing Class Members

- We can access
  - class attributes
  - class methods
- Some class members will not be accessible (more on that later)
- We need an object to access instance variables

Dot and pointer Operator

Accessing Class Members

If we have an object (dot operator)

- **Using the dot operator**

```
Account frank_account;

frank_account.balance;
frank_account.deposit(1000.00);
```

Accessing Class Members

If we have a pointer to an object (member of pointer operator)

- Dereference the pointer then use the dot operator.

```
Account  *frank_account = new Account();

(*frank_account).balance;
(*frank_account).deposit(1000.00);
```

- Or use the member of pointer operator (arrow operator)

```
Account *frank_account = new Account();

frank_account->balance;
frank_account->deposit(1000.00);
```

Private and public access Modifiers

Class Member Access Modifiers

`public`, `private`, and `protected`

- `public`
  - accessible everywhere
- `private`
  - accessible only by members or friends of the class
- `protected`
  - used with inheritance – we'll talk about it in the next section

Class Member Access Modifiers

public

```
class Class_Name
{
public:

  // declaration(s);

};
```

Class Member Access Modifiers

private

```
class Class_Name
{
private:

  // declaration(s);

};
```

Class Member Access Modifiers

protected

```
class Class_Name
{
protected:

  // declaration(s);

};
```

Declaring a Class

Player

```
class Player
{
private:
    std::string name;
    int health;
    int xp;
public:
  void talk(std::string text_to_say);
    bool is_dead();
};
```

Creating objects

```
Player frank;
frank.name = "Frank";       // Compiler error
frank.health = 1000;      // Compiler error
frank.talk("Ready to battle"); // OK


Player *enemy = new Player();
enemy->xp = 100;          // Compiler error
enemy->talk("I will hunt you down"); // OK

delete enemy;
```

Declaring a Class

Account

```
class Account
{
private:
    std::string name;
    double balance;

public:
  bool withdraw(double amount);
  bool deposit(double amount);
};
```

Creating objects

```
Account frank_account;
frank_account.balance = 10000000.00; // Compiler error
frank_account.deposit(10000000.0);     // OK
frank_account.name = "Frank's Account";  // Compiler error

Account *mary_account = new Account();

mary_account->balance = 10000.0;  // Compiler error
mary_account->withdraw(10000.0);  // OK

delete mary_account;
```

Methods, Constructors and Destructors
class methods

Implementing Member Methods

- Very similar to how we implemented functions

- Member methods have access to member attributes
  - So you don't need to pass them as arguments!

- Can be implemented inside the class declaration
  - Implicitly inline

- Can be implemented outside the class declaration
  - Need to use `Class_name::method_name`

- Can separate specification from implementation
  - .h file for the class declaration
  - .cpp file for the class implementation

Implementing Member Methods

Inside the class declaration

```
class Account {

private:
    double balance;
public:
    void set_balance(double bal) {
        balance = bal;
    }
    double get_balance() {
        return balance;
    }
};
```

Implementing Member Methods

Outside the class declaration

```
class Account {

private:
   double balance;
public:
   void set_balance(double bal);
   double get_balance();
};

void Account::set_balance(double bal) {
   balance = bal;
}
double Account::get_balance() {
   return balance;
}
```

Separating Specification from Implementation

Account.h

```
class Account {

private:
    double balance;
public:
    void set_balance(double bal);
    double get_balance();
};
```

Separating Specification from Implementation

Include Guards

```
#ifndef _ACCOUNT_H_
#define _ACCOUNT_H_

    // Account class declaration

#endif
```

Separating Specification from Implementation

Account.h

```
#ifndef _ACCOUNT_H_
#define _ACCOUNT_H_

class Account {

private:
   double balance;
public:
   void set_balance(double bal);
   double get_balance();
};

#endif
```

Separating Specification from Implementation

Account.h - #pragma once

```
#pragma once

class Account {

private:
    double balance;
public:
    void set_balance(double bal);
    double get_balance();
};
```

Separating Specification from Implementation

Account.cpp

```
#include "Account.h"

void Account::set_balance(double bal) {
    balance = bal;
}

double Account::get_balance() {
    return balance;
}
```

Separating Specification from Implementation

main.cpp

```
#include <iostream>
#include "Account.h"

int main() {
    Account frank_account;
    frank_account.set_balance(1000.00);
    double bal = frank_account.get_balance();

    std::cout << bal << std::endl; // 1000
    return 0;
}
```

Methods, Constructors and Destructors default and overloaded constructors

Constructors

- Special member method
- Invoked during object creation
- Useful for initialization
- Same name as the class
- No return type is specified
- Can be overloaded

Player Constructors

```
class Player
{
private:
    std::string name;
    int health;
    int xp;
public:
  // Overloaded Constructors
  Player();
  Player(std::string name);
  Player(std::string name, int health, int xp);
};
```

Account Constructors

```
class Account
{
private:
    std::string name;
    double balance;
public:
  // Constructors
  Account();
  Account(std::string name, double balance);
  Account(std::string name);
  Account(double balance);
};
```

Methods, Constructors and Destructors: Destructors

Destructors

- Special member method
- Same name as the class proceeded with a tilde (~)
- Invoked automatically when an object is destroyed
- No return type and no parameters
- Only 1 destructor is allowed per class – cannot be overloaded!
- Useful to release memory and other resources

Player Destructor

```
class Player
{
private:
    std::string name;
    int health;
    int xp;
public:
  Player();
  Player(std::string name);
  Player(std::string name, int health, int xp);
  // Destructor
  ~Player();
};
```

Account Destructor

```
class Account
{
private:
    std::string name;
    double balance;
public:
  Account();
  Account(std::string name, double balance);
  Account(std::string name);
  Account(double balance);
  // Destructor
  ~Account();
};
```

Creating objects

```
{
    Player slayer;
    Player frank {"Frank", 100, 4 };
    Player hero {"Hero"};
    Player villain {"Villain"};
    // use the objects
}   // 4 destructors called

Player *enemy = new Player("Enemy", 1000, 0);
delete enemy; // destructor called
```

Methods, Constructors and Destructors
Default Constructor

The Default Constructor

- Does not expect any arguments
  - Also called the no-args constructor
- If your write no constructors at all for a class
  - C++ will generate a Default Constructor that does nothing
- Called when you instantiate a new object with no arguments

```
Player frank;
Player *enemy = new Player;
```

Declaring a Class

Account – using default constructor

```
class Account
{
private:
    std::string name;
    double balance;
public:
  bool withdraw(double amount);
  bool deposit(double amount);
};
```

Creating objects

Using the default constructor

```
Account frank_account;
Account jim_account;


Account *mary_account = new Account;
delete mary_account;
```

Declaring a Class

Account – user-defined no-args constructor

```
class Account
{
private:
    std::string name;
    double balance;
public:
    Account() {
        name = "None";
        balance = 0.0;
    }
  bool withdraw(double amount);
  bool deposit(double amount);
};
```

Declaring a Class

Account – no default constructor

```
class Account
{
private:
    std::string name;
    double balance;
public:
    Account(std::string name_val, double bal) {
        name = name_val;
        balance = bal;
    }
  bool withdraw(double amount);
  bool deposit(double amount);
};
```

Creating objects

Using the default constructor

```
Account frank_account;                          // Error
Account jim_account;                            // Error

Account *mary_account = new Account;            // Error
delete mary_account;

Account bill_account {"Bill", 15000.0}; // OK
```

Methods, Constructors and Destructors
Overloading Constructors

Overloading Constructors

- Classes can have as many constructors as necessary
- Each must have a unique signature
- Default constructor is no longer compiler-generated once another constructor is declared

Constructors and Destructors

Overloaded Constructors

```
class Player
{
private:
    std::string name;
    int health;
    int xp;
public:
  // Overloaded Constructors
  Player();
  Player(std::string name_val);
  Player(std::string name_val, int health_val, int xp_val);
};
```

Constructors and Destructors

Overloaded Constructors

```
Player::Player() {
    name = "None";
    health = 0;
    xp = 0;
}

Player::Player(std::string name_val) {
    name = name_val;
    health = 0;
    xp = 0;
}
```

Constructors and Destructors

Overloaded Constructors

```
Player::Player(std::string name_val, int health_val, int
xp_val) {
    name = name_val;
    health = health_val;
    xp = xp_val;
}
```

Creating objects

```
Player empty;                                    // None, 0, 0

Player hero {"Hero"};                             // Hero, 0, 0
Player villain {"Villain"};                       // Villain, 0, 0

Player frank {"Frank", 100, 4};                   // Frank, 100, 4

Player *enemy = new Player("Enemy", 1000, 0); // Enemy, 1000, 0
delete enemy;
```

Methods, Constructors and Destructors
Constructors Initialization Lists

Constructor Initialization Lists

- So far, all data member values have been set in the constructor body

- Constructor initialization lists
  - are more efficient
  - initialization list immediately follows the parameter list
  - initializes the data members as the object is created!
  - order of initialization is the order of declaration in the class

Constructor Initialization Lists

```
class Player
{
private:
    std::string name;
    int health;
    int xp;
public:
  // Overloaded Constructors
  Player();
  Player(std::string name_val);
  Player(std::string name_val, int health_val, int xp_val);
};
```

Constructor Initialization Lists

Player()

Previous way:

```
Player::Player() {
    name = "None";    // assignment not initialization
    health = 0;
    xp = 0;
}
```

Better way:

```
Player::Player()
    : name{"None"}, health{0}, xp{0} {
}
```

Constructor Initialization Lists

Player(std::string)

Previous way:

```
Player::Player(std::string name_val) {
    name = name_val;   // assignment not initialization
    health = 0;
    xp = 0;
}
```

Better way:

```
Player::Player(std::string name_val)
    : name{name_val}, health{0}, xp{0} {
}
```

Constructor Initialization Lists

Player(std::string, int, int)

Previous way:

```
Player::Player(std::string name_val, int health_val, int xp_val) {
    name = name_val;                // assignment not initialization
    health = health_val;
    xp = xp_val;
}
```

Better way:

```
Player::Player(std::string name_val, int health_val, int xp_val)
        : name{name_val}, health{health_val}, xp{xp_val} {
}
```

Constructor Initialization Lists

```
Player::Player()
    : name{"None"}, health{0}, xp{0} {
}

Player::Player(std::string name_val)
    : name{name_val}, health{0}, xp{0} {
}

Player::Player(std::string name_val, int health_val, int
xp_val)
        : name{name_val}, health{health_val}, xp{xp_val} {
}
```

Methods, Constructors and Destructors
Delegating Constructors

Delegating Constructors

- Often the code for constructors is very similar
- Duplicated code can lead to errors
- C++ allows delegating constructors
  - code for one constructor can call another in the initialization list
  - avoids duplicating code

Delegating Constructors

```
class Player
{
private:
    std::string name;
    int health;
    int xp;
public:
  // Overloaded Constructors
  Player();
  Player(std::string name_val);
  Player(std::string name_val, int health_val, int xp_val);
};
```

Delegating Constructors

```
Player::Player()
    : name{"None"}, health{0}, xp{0} {
}

Player::Player(std::string name_val)
    : name{name_val}, health{0}, xp{0} {
}

Player::Player(std::string name_val, int health_val, int
xp_val)
    : name{name_val}, health{health_val}, xp{xp_val} {
}
```

Delegating Constructors

```
Player::Player(std::string name_val, int health_val, int xp_val)
    : name{name_val}, health{health_val}, xp{xp_val} {
}

Player::Player()
    : Player {"None", 0, 0} {
}

Player::Player(std::string name_val)
    : Player { name_val, 0, 0} {
}
```

Methods, Constructors and Destructors
Default Constructor Parameters

Default Constructor Parameters

- Can often simplify our code and reduce the number of overloaded constructors
- Same rules apply as we learned with non-member functions

Default Constructor Parameters

```
class Player
{
private:
    std::string name;
    int health;
    int xp;
public:
  // Constructor with default parameter values
  Player(std::string name_val = "None",
         int health_val = 0,
         int xp_val = 0);
};
```

Default Constructor Parameters

```
Player::Player(std::string name_val, int health_val, int
xp_val)
    : name {name_val}, health {health_val}, xp {xp_val} {

}
Player empty;                          // None, 0, 0
Player frank {"Frank"};                 // Frank, 0, 0
Player villain {"Villain", 100, 55}; // Villain, 100, 55
Player hero {"Hero", 100};             // Hero, 100, 0

// Note what happens if you declare a no-args constructor
```

Methods, Constructors and Destructors copy and move constructors

Copy Constructor

- When objects are copied C++ must create a new object from an existing object

- When is a copy of an object made?
  - passing object by value as a parameter
  - returning an object from a function by value
  - constructing one object based on another of the same class

- C++ must have a way of accomplishing this so it provides a compiler-defined copy constructor if you don't

Pass object by-value

```
Player hero {"Hero", 100, 20};

void display_player(Player p) {
    // p is a COPY of hero in this example
    // use p
    // Destructor for p will be called
}

display_player(hero);
```

Return object by value

```
Player enemy;

Player create_super_enemy() {
    Player an_enemy{"Super Enemy", 1000, 1000};
    return an_enemy;  // A COPY of an_enemy is returned
}

enemy = create_super_enemy();
```

Construct one object based on another

```
Player hero {"Hero", 100, 100};

Player another_hero {hero};  // A COPY of hero is made
```

Copy Constructor

- If you don't provide your own way of copying objects by value then the compiler provides a default way of copying objects

- Copies the values of each data member to the new object
  - default memberwise copy

- Perfectly fine in many cases

- Beware if you have a pointer data member
  - Pointer will be copied
  - Not what it is pointing to
  - Shallow vs. Deep copy

Best practices

- Provide a copy constructor when your class has raw pointer members
- Provide the copy constructor with a **const reference** parameter
- Use STL classes as they already provide copy constructors
- Avoid using raw pointer data members if possible

Declaring the Copy Constructor

```
Type::Type(const Type &source);


Player::Player(const Player &source);

Account::Account(const Account &source);
```

Implementing the Copy Constructor

```
Type::Type(const Type &source) {
    // code or initialization list to copy the object
}
```

Implementing the Copy Constructor

```
Player
```

```
Player::Player(const Player &source)
    : name{source.name},
      health {source.health},
      xp {source.xp} {
}
```

Implementing the Copy Constructor

Account

```
Account::Account(const Account &source)
    : name{source.name},
      balance {source.balance} {
}
```

Methods, Constructors and Destructors shallow and deep copying
Shallow copy

Shallow vs. Deep Copying

- Consider a class that contains a pointer as a data member
- Constructor allocates storage dynamically and initializes the pointer
- Destructor releases the memory allocated by the constructor
- What happens in the default copy constructor?

Default copy constructor

- memberwise copy
- Each data member is copied from the source object
- The pointer is copied NOT what it points to (shallow copy)
- **Problem**: when we release the storage in the destructor, the other object still refers to the released storage!

Copy Constructor

Shallow

```
class Shallow {
private:
    int *data;                                  // POINTER
public:
   Shallow(int d);                             // Constructor
   Shallow(const Shallow &source);              // Copy
Constructor
   ~Shallow();                                 // Destructor
};
```

Copy Constructor

Shallow constructor

```
Shallow::Shallow(int d) {
     data = new int;  // allocate storage
     *data = d;
}
```

Copy Constructor

Shallow destructor

```
Shallow::~Shallow() {
    delete data;   // free storage
    cout << "Destructor freeing data" << endl;
}
```

Copy Constructor

Shallow copy constructor

```
Shallow::Shallow(const Shallow &source)
     : data(source.data) {
          cout << "Copy constructor - shallow"
               << endl;
}
```

Shallow copy - only the pointer is copied - not what it is pointing to!

Problem: `source` and the newly created object BOTH point to the SAME `data` area!

Copy Constructor

Shallow – a simple method that expects a copy

```
Shallow::Shallow(const Shallow &source)
     : data(source.data) {
          cout << "Copy constructor - shallow"
               << endl;
}
```

Shallow copy - only the pointer is copied - not what it is pointing to!

Problem: `source` and the newly created object BOTH point to the SAME `data` area!

Copy Constructor

Sample main – will likely crash

```
int main() {
    Shallow obj1 {100};
    display_shallow(obj1);
  // obj1's data has been released!

    obj1.set_data_value(1000);
    Shallow obj2 {obj1};
    cout << "Hello world" << endl;
    return 0;
}
```

Methods, Constructors and Destructors
shallow and deep copying
Deep copy

User-provided copy constructor

- Create a **copy** of the pointed-to data
- Each copy will have a pointer to unique storage in the heap
- Deep copy when you have a raw pointer as a class data member

Copy Constructor

Deep

```
class Deep {
private:
    int *data;          // POINTER
public:
   Deep(int d);          // Constructor
   Deep(const Deep &source);  // Copy Constructor
   ~Deep();          // Destructor
};
```

Copy Constructor

Deep constructor

```
Deep::Deep(int d) {
     data = new int;  // allocate storage
     *data = d;
}
```

Copy Constructor

Deep constructor

```
Deep::~Deep() {
     delete data;   // free storage
     cout << "Destructor freeing data" << endl;
}
```

Copy Constructor

Deep constructor

```
Deep::Deep(const Deep &source)
{
     data = new int;   // allocate storage
     *data = *source.data;
     cout << "Copy constructor - deep"
          << endl;
}
```

Deep copy – create new storage and copy values

Copy Constructor

Deep copy constructor – delegating constructor

```
Deep::Deep(const Deep &source)
  : Deep{*source.data} {
        cout << "Copy constructor - deep"
             << endl;
}
```

Delegate to `Deep(int)` and pass in the `int (*source.data)` source is pointing to

Copy Constructor

Deep – a simple method that expects a copy

```
void display_deep(Deep s) {
    cout << s.get_data_value() << endl;
}
```

When `s` goes out of scope the destructor is called and releases `data`.
No Problem: since the storage being releases is unique to `s`

Copy Constructor

Sample main – will not crash

```
int main() {
    Deep obj1 {100};
    display_deep(obj1);

    obj1.set_data_value(1000);
    Deep obj2 {obj1};

    return 0;
}
```

Methods, Constructors and Destructors
Move Constructor

Move Constructor

- Sometimes when we execute code the compiler creates unnamed temporary values

```
int total {0};
total = 100 + 200;
```

  - `100 + 200` is evaluated and `300` stored in an unnamed temp value
  - the `300` is then stored in the variable total
  - then the temp value is discarded

- The same happens with objects as well

When is it useful?

- Sometimes copy constructors are called many times automatically due to the copy semantics of C++

- Copy constructors doing deep copying can have a significant performance bottleneck

- C++11 introduced move semantics and the move constructor
- Move constructor moves an object rather than copy it

- Optional but recommended when you have a raw pointer

- Copy elision – C++ may optimize copying away completely (RVO-Return Value Optimization)

r-value references

- Used in moving semantics and perfect forwarding
- Move semantics is all about r-value references
- Used by move constructor and move assignment operator to efficiently move an object rather than copy it
- R-value reference operator (&&)

r-value references

```
int x {100}
   int &l_ref = x;    // l-value reference
   l_ref = 10;        // change x to 10

   int &&r_ref = 200;    // r-value ref
   r_ref = 300;      // change r_ref to 300

   int &&x_ref = x;  // Compiler error
```

l-value reference parameters

```
int x {100};      // x is an l-value

void func(int &num);// A

func(x);     // calls A - x is an l-value
func(200);      // Error - 200 is an r-value
```

error: cannot bind non-const lvalue reference of type 'int&' to an rvalue of type 'int'

r-value reference parameters

```
int x {100};      // x is an l-value

void func(int &&num); // B

func(200);    // calls B - 200 is an r-value
func(x);    // ERROR - x is an l-value
```

error: cannot bind rvalue reference of type 'int&&' to lvalue of type 'int'

l-value and r-value reference parameters

```
int x {100};      // x is an l-value

void func(int &num);  // A
void func(int &&num); // B

func(x);     // calls A - x is an l-value
func(200);    // calls B - 200 is an r-value
```

Example - Move class

```
class Move {
private:
   int *data;           // raw pointer
public:
   void set_data_value(int d) { *data = d; }
   int get_data_value()     { return *data; }
   Move(int d);         // Constructor
   Move(const Move &source);  // Copy Constructor
   ~Move();           // Destructor
};
```

Move class copy constructor

```
Move::Move(const Move &source) {
    data = new int;
    *data = *source.data;
}
```

Allocate storage and copy

Inefficient copying

```
Vector<Move> vec;

vec.push_back(Move{10});
vec.push_back(Move{20});
```

Copy Constructors will be called to copy the temps

Inefficient copying

```
Constructor for: 10
Constructor for: 10
Copy constructor  - deep copy for: 10
Destructor freeing data for: 10
Constructor for: 20
Constructor for: 20
Copy constructor  - deep copy for: 20
Constructor for: 10
Copy constructor  - deep copy for: 10
Destructor freeing data for: 10
Destructor freeing data for: 20
```

What does it do?

- Instead of making a deep copy of the move constrictor
  - ‘moves’ the resource
  - Simply copies the address of the resource from source to the current object
  - And, nulls out the pointer in the source pointer

- Very efficient

syntax - r-value reference

```
Type::Type(Type &&source);

Player::Player(Player &&source);

Move::Move(Move &&source);
```

Move class with move constructor

```
class Move {
private:
    int *data;          // raw pointer
public:
    void set_data_value(int d) { *data = d; }
    int get_data_value()       { return *data; }
    Move(int d);        // Constructor
    Move(const Move &source);// Copy Constructor
    Move(Move &&source);    // Move Constructor
    ~Move();           // Destructor
};
```

Move class move constructor

```
Move::Move(Move &&source)
    : data{source.data} {
        source.data = nullptr;
}
```

'Steal' the data and then null out the source pointer

efficient

```
Vector<Move> vec;

vec.push_back(Move{10});
vec.push_back(Move{20});
```

Move Constructors will be called for the temp r-values

efficient

```
Constructor for: 10
Move constructor - moving resource: 10
Destructor freeing data for nullptr
Constructor for: 20
Move constructor - moving resource: 20
Move constructor - moving resource: 10
Destructor freeing data for nullptr
Destructor freeing data for nullptr
Destructor freeing data for: 10
Destructor freeing data for: 20
```

Methods, Constructors and Destructors
this pointer

`this` pointer

- `this` is a reserved keyword
- Contains the address of the object - so it's a pointer to the object
- Can only be used in class scope
- All member access is done via the `this` pointer
- Can be used by the programmer
  - To access data member and methods
  - To determine if two objects are the same (more in the next section)
  - Can be dereferenced (`*this`) to yield the current object

`this` pointer

```
void Account::set_balance(double bal) {
    balance = bal;  // this->balance is implied
}
```

`this` pointer

To disambiguate identifier use

```
void Account::set_balance(double balance) {
   balance = balance;   // which balance? The parameter
}

void Account::set_balance(double balance) {
   this->balance = balance;  // Unambiguous
}
```

`this` pointer

To determine object identity

- Sometimes its useful to know if two objects are the same object

```
int Account::compare_balance(const Account &other) {
    if (this == &other)
        std::cout << "The same objects" << std::endl;
    ...
}
frank_account.compare_balance(frank_account);
```

- We'll use the `this` pointer again when we overload the assignment operator

Methods, Constructors and Destructors
Using Const with Classes

Using const with classes

- Pass arguments to class member methods as `const`
- We can also create `const` objects
- What happens if we call member functions on `const` objects?
- `const`-correctness

Creating a const object

- `villain` is a const object so it's attributes cannot change

```
const Player villain {"Villain", 100, 55};
```

Using const with classes

What happens when we call member methods on a const object?

```
const Player villain {"Villain", 100, 55};

villain.set_name("Nice guy");          // ERROR

std::cout << villain.get_name() << std::endl; // ERROR
```

Using const with classes

What happens when we call member methods on a const object?

```
const Player villain {"Villain", 100, 55};

void display_player_name(const Player &p) {
    cout << p.get_name() << endl;
}

display_player_name(villain);      // ERROR
```

Using const with classes

const methods

```
class Player {
  private:
     . . .
  public:
     std::string get_name() const;
     . . .
  };
```

Using const with classes

const-correctness

```
const Player villain {"Villain", 100, 55};

villain.set_name("Nice guy");          // ERROR

std::cout << villain.get_name() << std::endl; // OK
```

Using const with classes

const methods

```
class Player {
  private:
     . . .
  public:
     std::string get_name() const;
     // ERROR if code in get_name modifies this object
     . . .
  };
```

Static class members

Static Class Members

- Class data members can be declared as static
  - A single data member that belongs to the class, not the objects
  - Useful to store class-wide information

- Class functions can be declared as static
  - Independent of any objects
  - Can be called using the class name

Player class -static members

```
class Player {
private:
    static int num_players;

public:
    static int get_num_players();
    . . .
};
```

Player class – initialize the static data

Typically in Player.cpp

```
#include "Player.h"

int Player::num_players = 0;
```

Player class – implement static method

```
int Player::get_num_players() {
    return num_players;
}
```

Player class –update the constructor

```
Player::Player(std::string name_val, int health_val,
int xp_val)
    : name{name_val}, health{health_val}, xp{xp_val} {
        ++num_players;
}
```

Player class - Destructor

```
Player::~Player() {
    --num_players;
}
```

main.cpp

```
void display_active_players() {
    cout << "Active players: "
         << Player::get_num_players() << endl;
}

int main() {
    display_active_players();

    Player obj1 {"Frank"};
    display_active_players();
    . . .
```

Struct vs. class

Structs vs Classes

- In addition to define a `class` we can declare a `struct`
- `struct` comes from the C programming language
- Essentially the same as a `class` expect
  - members are `public` by default

class

```
class Person {
    std::string name;
    std::string get_name();
};

Person p;
p.name = "Frank";    // compiler error - private
std::cout << p.get_name(); // compiler error - private
```

struct

```
struct Person {
    std::string name;
    std::string get_name(); // Why if name is public?
};

Person p;
p.name = "Frank";    // OK - public
std::cout << p.get_name(); // OK - public
```

Some general guidelines

- `struct`
  - Use `struct` for passive objects with public access
  - Don’t declare methods in `struct`

- `class`
  - Use class for active objects with private access
  - Implement getters/setters as needed
  - Implement member methods as needed

Friend of a class

Friends of a Class

- Friend
  - A function or class that has access to private class member
  - And, that function of or class is NOT a member of the class it is accessing

- Function
  - Can be regular non-member functions
  - Can be member methods of another class

- Class
  - Another class can have access to private class members

Friends of a Class

- Friendship must be granted NOT taken
  - Declared explicitly in the class that is granting friendship
  - Declared in the function prototype with the keyword `friend`

- Friendship is not symmetric
  - Must be explicitly granted
    - if A is a friend of B
    - B is NOT a friend of A

- Friendship is not transitive
  - Must be explicitly granted
    - if A is a friend of B AND
    - B is a friend of C
    - then A is NOT a friend of C

non-member function

```
class Player {
    friend void display_player(Player &p);
    std::string name;
    int health;
    int xp;
public:
    . . .
};
```

non-member function

```
void display_player(Player &p) {
    std::cout << p.name << std::endl;
    std::cout << p.health << std::endl;
    std::cout << p.xp << std::endl;
}
```

`display_player` may also change private data members

member function of another class

```
class Player {
    friend void Other_class::display_player(Player &p);
    std::string name;
    int health;
    int xp;
public:
    . . .
};
```

member function of another class

```
class Other_class {
    . . .
public:
    void display_player(Player &p) {
        std::cout << p.name << std::endl;
        std::cout << p.health << std::endl;
        std::cout << p.xp << std::endl;
    }
};
```

Another class as a friend

```
class Player {
    friend class Other_class;
    std::string name;
    int health;
    int xp;
public:
    . . .
};
```

Good Luck!