```
//================================
// Section 13: OOP - Classes and Objects
//================================

// What is OOP
Procedural programming
- Focus on processes or actions the program takes
- Collection of functions
- Data is declared separately
- Passed into functions as arguments
- Fairly easy to learn

Limitations:
functions need to know about the structure of the data
If it changes the functions probably need to change too

As programs get bigger they get:
- Difficult to understand, maintain, extend, debug, etc
- Hard to reuse code
- Fragile and easier to break

// Classes and objects
- Classes model real world domain entities
- higher levels of abstraction
- best for large programs

// Encapsulation
- Objects contain data and operations that work on the data
- ADT - Abstract data type

// Information hiding
- Implementation-specific logic can be hidden
- Users of the class code to the interface since they know nothing else
- More abstraction
- Easier to test maintain and debug.

// Reusability
- Easier to reuse classes in other applciation
- Faster development
- Higher quality

// Inheritance
- Create a new class based on an existing class
- Helps us reuse classes
- Polymorphic classes

// Limitations
Not a cure all
- Won't make bad code better, it probably makes it worse
- Not suitable for all problems
- Not everything decomposes to a class

Learning curve
- Steeper learning curve, especially for C++
- Many OO languages and variations of OO concepts

Design
- Usually more up front design is needed for good models and hierarchies

Programs can be:
- Large in size
- slower
- more complex
```

```
// What are classes and objects?
Classes
- Blueprints from which objects are created
- User defined
- has data and functions (methods and attributes)
- Can hide data and methods
- provide a public interface


Objects
- Created from classes
- Represents an instance of the class
- We can have lots of objects
- Each has its own identity
- Each can use the defined class methods

// Declaring a class and creating objects
class ClassName{ // capitalize class names
    // declarations
}; // semicolon is required

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

    // methods
    void talk(std::string textToSay);
    bool isDead();
}; // Remember the semicolon after the class

// After a class is declared we can create objects from it
Player Yaya;
Player hero;
Player * enemy = new Player(); // can create pointers on the heap
delete enemy; // delete it then

// Accessing class members
We can access:
- attribbutes
- methods

some class members will not be accessible
We need an object to access instance variables

If we have an object(dot operator)
Account myAccount;
myAccount.balance;
myAccount.deposit(1000.00);

If we have a pointer (two ways)
- Dereference then use the dot operator

Account * myAccount = new Account();

(*myAccount).balance; // dot operator has higher precedence
(*myAccount).deposit(1000.00);

Alternatively, use the member of pointer operator (->)
myAccount->balance;
myAccount->deposit(1000.00);
```

```
//====================
// Example
//====================
#include <iostream>
#include <vector>
#include <string>
#include <cmath>
#include <ctime>

class Player {
public: // by default all members are private
    // Attributes
    std::string name;
    int health;
    int xp;

    // methods
    void talk(std::string incomingString) {
        std::cout << name << " says: " << incomingString << std::endl;
    };
    bool isDead() {};
};


int main(){

    Player yaya;
    yaya.name = "Yaya";
    yaya.health = 200;
    yaya.xp = 512;
    std::cout << yaya.name << " " << yaya.health << " " << yaya.xp << std::endl;
    yaya.talk("I am the mighty yaya");
    Player* babs = new Player();
    babs->name = "Bablet";
    (*babs).health = 200; // Dereference syntax - use the arrow instead
    babs->xp = 34;
    std::cout << babs->name << " " << babs->health << " " << babs->xp << std::endl;
    babs->talk("Babs is the best");

    return 0;
}
//====================
// Example
//====================

// Public and private
public
- accessible everywhere

private
- accessible only by members or friends of the class

protected
- used with inheritance

class ClassName{
    private:
    // Stuff in here is private

    public:
    // Stuff in here is public

    protected:
    // Stuff in here is protected
}

Compiler will not let you access private class members
```

```
// Implementing member methods
Similar to how we implemented functions
Membre methods have access to member attributes
-No need to pass arguments

Can be implemented inside the class declaration
- Implicitly inline

Can be implemented outside
- Use scope resolution to access: ClassName::methodName

Separate specification from the implementation
.h file for the class declaration
.cpp for the implementation

//Separating classes out

Put in Account.h file
#ifndef _ACCOUNT_H_
#define _ACCOUNT_H_
class Account{
    private:
    double balance;
    public:
    void setBalance(double bal);
    double getBalance();
}
#endif

//Put in Account.cpp file
#include "Account.h"
void Account::setBalance(double bal){
    balance = bal;
}

double Account::getBalance(){
    return balance;
}

// Include guards to make sure the compiler processes this only once
#ifndef _ACCOUNT_H_
#define _ACCOUNT_H_

#endif

or use #pragma once

// Example Account.h file
#pragma once
#include <string>
class Account {
private:
    // attributes
    std::string name;
    double balance;
public:
    // Methods
    void setBalance(double bal);
    double getBalance();
    void setName(std::string n);
    std::string getName();
    bool deposit(double amount);
    bool withdraw(double amount);
};
```

```cpp
// Example Account.cpp file
#include "Account.h"
void Account::setBalance(double bal) {
    balance = bal;
}
double Account::getBalance() {
    return balance;
}

void Account::setName(std::string n) {
    name = n;
}

std::string Account::getName() {
    return name;
}
bool Account::deposit(double amount) {
    balance += amount;
    return true;
}
bool Account::withdraw(double amount) {
    if (balance - amount > 0) {
        balance -= amount;
        return true;
    } else {
        return false;
    }
}

// Example main.cpp file
#include <iostream>
#include <vector>
#include <string>
#include <cmath>
#include <ctime>
#include "Account.h"

int main(){

    Account josieAccount;
    josieAccount.setName("Josie Revisited");
    josieAccount.setBalance(1000.00);

    if (josieAccount.deposit(200)) {
        std::cout << "Deposit ok" << std::endl;
    } else {
        std::cout << "Deposit not allowed" << std::endl;
    }

    if (josieAccount.withdraw(500.00)) {
        std::cout << "Withdrawal okay" << std::endl;
    } else {
        std::cout << "Not sufficient funds" << std::endl;
    }

    if (josieAccount.withdraw(15000)) {
        std::cout << "Withdrawal okay" << std::endl;
    } else {
        std::cout << "Not sufficient funds" << std::endl;
    }
    return 0;
}
```

```
// Constructors and destructors
Constructors
- Special member method
- Invoked during object creation
- Used for initialization
- Same name as the class
- No return type
- Can be overloaded - multiple constructors

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

Destructors
- special member method
- Same name as class preceded with a ~
- Invoked automatically when an object is destroyed
- No return type and no parameters
- Only one destructor is allowed per class - we can't overload these
- Useful to release memory and other resources

class Player{
    private:
    // stuff
    public:
    // Overloaded constructors
    Player();
    Player(std::string name);
    Player(std::string name, int health, int xp);

    // Destructor
    ~Player();
}

// Creating objects
{
Player slayer;                      // no args constructor called
Player Yaya{"Yaya", 100, 10};       // Three args constructor called
Player Babs{"Babs"};                // One arg constructor called
Player * enemy = new Enemy("Fiteme", 1000, 0); //Works the same with pointers
delete enemy; // destructor called
} // Destructor called when the other objects go out of scope

NOTE: If we don't provide one, C++ will give us a constructor and destructor for free!
But they are empty.

// Default constructors
Default constructor
- Does not expect any arguments (aka the no-args constructor)
- If you write no constructors C++ will generate one
- Called when you create an object with no arguments
- Best practice is to define our own no-args constructor
- If you define a constructor that has argument, you no longer get a free default constructor from C++
- If we need it, we need to define it ourselves
- We would no longer be able to create objects with no arguments
```

```
class Player{
    private:
     std::string name;
     int health;
     int xp;
    public:
    // Overloaded constructors
    Player(){ // default no-args constructor
        name = "none";
        health = 100;
        xp = 0;
    }
    Player(std::string name){
        name =  name;
        health = 100;
        xp = 0;
    }
    Player(std::string name, int health, int xp){
        name = name;
        health = health;
        xp = xp;
    }

    // Destructor
    ~Player(){};
}

// 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
Make sure that you initialize all member variables so that nothing contains garbage data

// Constructor initialization lists
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
So far what we have been doing isn't initialization. It is assignment.

// Using an initializer list
Player::Player()
    : name{"None"}, health{0}, xp{0}
{// whatever other statements necessary in here}

class Player{
    private:
     std::string name;
     int health;
     int xp;
    public:
    // Overloaded constructors
    Player()
        : name{"None"}, health{0}, xp{0}{ // default no-args constructor
    }
    Player(std::string name)
        : name{name}{
    }
    Player(std::string name, int health, int xp)
        : name{name}, health{health}, xp{xp}{
    }

    // Destructor
    ~Player(){};
}
```

```
// 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

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

    Player()
        : Player{"None", 0,0}{ // This calls the three args constructor with these values
    }
    Player(std::string name)
        : Player{name, 0, 0}{ // Again, calls the three args constructor and passes in these values
    }

    // Destructor
    ~Player(){};
}

// Constructor parameters and default
This can simplify the code and reduce the number of overloaded constructors
Same rules apply as with non-member functions
You can't always have default values, but many times you can



class Player{
    private:
     std::string name;
     int health;
     int xp;
    public:
    // Constructor prototype with default parameters. We can now create objects with any number of args
    Player(std::string name = "None", int health = 100, int xp = 0)
    }

    // Destructor
    ~Player();
}

// Constructor implementation outside the class
Player::Player(std::String name, int health, int xp)
    :name{name}, health{health}, xp{xp}{
}

// Copy constructor
When objects are copied, C++ must create a new object from an existing one
When do we do this?
- Passing an 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 to do this so if we don't specify, the compiler will make one for us
```

```
Use cases
When we pass an object to a function by value (copy is made)
// Create an object
Player hero{"Hero", 100, 20};

// Some random display function
void displayPlayer(Player playerObjectByValue){
    // playerObjectByValue is a COPY of hero in this example
    // the function will do whatever it needs to with the copy
    // The destructor for playerObjectByValue will then be called
}

// Call our function - it needs to make a copy of hero to do its job
displayPlayer(hero);

// What do they do?
If you don't provide one, the compiler will
Copies the values of each data member to the new object
- Defaults to memberwise copy - which is each member is just directly copied over

Perfectly fine in many cases
Beware if you have a pointer as a data member
- The pointer will be copied, but not what it is pointing to
- This is a shallow copy and you will run into issues with two pointers pointing to the same data

// Best practices
If you are using raw pointers, provide your own copy constructor
Create it with a const reference parameter
Use STL classes when you can - they already have their own copy constructors
Avoid using raw pointers if you can

// Declaring the copy constructor
Type::Type (const Type &source); // it passes in a single object as a constant reference so we don't damage the source
Player::Player(const Player&source);

// Now we can do whatever initialization we need
Player::Player(const Player&source){
    // We can put in whatever code or "assignment" style initialization we need
}

// Or we can use the initializer list (preferred)
Player::Player(const Player&source)
: name{source.name}, health{source.health}, xp{source.xp}{
    // Any other stuff here
}
```

```cpp
//====================================
// Copy constructor example program
//====================================
#include <iostream>
#include <vector>
#include <string>
#include <cmath>
#include <ctime>

class Player {
private:
    std::string name;
    int health;
    int xp;

public:
    std::string getName() { return name; }
    int getHealth() { return health; }
    int getXp() { return xp; }
    // Constructor
    Player(std::string name = "None", int health = 100, int xp = 0);

    // Copy Constructor
    Player(const Player& source);

    // Destructor
    ~Player() {
        std::cout << "Destructor called for: " << name << std::endl;
    };
};

// Create a function that expects a Player object - this will need to use the copy constructor
void displayPlayer(Player p) {
    std::cout << "Name: " << p.getName() << std::endl;
    std::cout << "Health: " << p.getHealth() << std::endl;
    std::cout << "XP: " << p.getXp() << std::endl;
}

Player::Player(std::string name, int health, int xp)
    : name{ name }, health{ health }, xp{ xp } {
    std::cout << "Three args constructor with initiliazer list to take care of everything" << std::endl;
}

Player::Player(const Player& source)
    : name(source.name), health(source.health), xp(source.xp) {
    std::cout << "Copy constructor - made a copy of: " << source.name << std::endl;
}

int main(){
    Player empty;
    displayPlayer(empty);
    Player Josie{ "Josie" };
    Player hero{ "Hero", 100 };
    Player villain{ "Villain", 1000, 1000 };


    return 0;
 }
```

```cpp
//====================================
// Copy constructor example program
//====================================

// Note that we could also use a delegating constructor
Player::Player(const Player& source)
    //: name(source.name), health(source.health), xp(source.xp) {
    :Player{source.name, source.health, source.xp}{
    std::cout << "Copy constructor - made a copy of: " << source.name << std::endl;
}

// Shallow copying
If a class has a raw pointer, the constructor allocates storage and initializes the pointer
The destructor then releases that memory

Default copy constructor:
memberwise copy - each object attribute copied as-is
The pointer is copied, but NOT what it's pointing to.
The problem is when one of the objects goes out of scope or the destructor is called, the
allocated memory is released but there is still a pointer out there that thinks it's valid

// Shallow example
class Shallow{
    private:
        int * data;
    public:
        // Constructor
        Shallow(int d);

        // Copy constructor
        Shallow(const Shallow &source);

        // Destructor
        ~Shallow(){}
}

// Constructor implementation
Shallow::Shallow(int d){
    data = new int;     // Allocate space on the heap for it
    *data = d;          // Assign the value we passed in
}

// Destructor implementation
Shallow::~Shallow(){
    delete data;
}

// Copy constructir implementation
Shallow:Shallow(const Shallow & source)
    : data(source.data){
    // Both the copy and the source will point to the same object!
}
```

```cpp
//===================================
// Shallow copy constructor example program
//===================================
#include <iostream>
#include <vector>
#include <string>
#include <cmath>
#include <ctime>
#include "../../std_lib_facilities.h"


class Shallow {
private:
    int* data;

public:
    void setData(int d) { *data = d; }
    int getDataValue() {
        return *data;
    }

    //Constructor
    Shallow(int d);

    // Copy Constructor
    Shallow(const Shallow& source);

    // Destructor
    ~Shallow();
};

Shallow::Shallow(int d) {
    data = new int;
    *data = d;
}

Shallow::Shallow(const Shallow& source)
    :data(source.data) {
    std::cout << "Copy Constructor - shallow copy" << std::endl;
}

Shallow::~Shallow() {
    delete data;
    std::cout << "Destructor freeing data" << std::endl;
}

void displayShallow(Shallow shallowObjectToCopy) {
    std::cout << shallowObjectToCopy.getDataValue() << std::endl;
}


int main(){

    Shallow object1{ 100 };
    displayShallow(object1);

    Shallow object2{ object1 };
    object2.setData(1000);

    return 0;
}
```

```
// Deep copying with the copy constructor

This copies the pointer and the data it points to
Each copy has a pointer to unique storage in the heap
Always use when you have a raw C++ pointer

class Deep{
    private:
        int * data;
    public:
        Deep (int d);
        Deep (const Deep &source);
        ~Deep();
}

Deep::Deep(int d){
    data = new int;
    *data = d;
}

Deep::Deep(const Deep &source){
    data = new int;
    *data = *source.data;
    std::cout << "Deep copy of the data" << std::endl;
}

Deep::Deep(){
    delete data; // Free storage
}

// We can also use delegation
Deep::Deep(const Deep &source)
:Deep{*source.data}{
    std::cout << "Copy constructor - deep" << std::endl;
}

//=====================================
// Deep copy constructor example program
//=====================================
#include <iostream>
#include <vector>
#include <string>
#include <cmath>
#include <ctime>
#include "../../std_lib_facilities.h"

class Deep {
private:
    int* data;

public:
    void setData(int d) { *data = d; }
    int getDataValue() {
        return *data;
    }

    //Constructor
    Deep(int d);

    // Copy Constructor
    Deep(const Deep& source);

    // Destructor
    ~Deep();
};
```

```cpp
Deep::Deep(int d) {
    data = new int;
    *data = d;
}

Deep::Deep(const Deep& source)
    :Deep{ *source.data } { // In this case we do a deep copy by delegating to the main constructor
    std::cout << "Copy Constructor - shallow copy" << std::endl;
}

Deep::~Deep() {
    delete data;
    std::cout << "Destructor freeing data" << std::endl;
}

void displayDeep(Deep shallowObjectToCopy) {
    std::cout << shallowObjectToCopy.getDataValue() << std::endl;
}

int main() {

    Deep object1{ 100 };
    displayDeep(object1);

    Deep object2{ object1 };
    object2.setData(1000);

    return 0;
}

// Move Constructor
Sometimes when we execute code the compiler will create unnamed temp values
int total{0};
total = 100 + 200;

100 + 200 is evaluated and the result 300 is stored in an unnamed temp value
Then the 300 is stored in the variable total
Then the temp value is discarded
The same happens with objects
It's important to tell whether an expression is an L-Value or an R-Value

// When are move semantics useful?
Sometimes copy constructors are called a LOT because of how we copy things
The ones doing deep copy can take a LOT of time
Sometimes it makes more sense to move the object rather than copy it
It's optional, but recommended when you use a raw pointer
Copy elision - C++ may automatically eliminate the unnecessary copying
RVO - Return Value Optimization - the compiler generates code that doesn't copy a return
value from a function. Makes ir more efficient

//R-Value references
Used in move semantics and perfect forwarding
Move semantics are all about r-value references - values that can't be addressed
Used by the move constructor and move assignment operator to efficiently move an object rather than copy it
R-Value reference operator is the double ampersand (&&)

// Examples
L-Value references:
int x{100}; // In this instance, x is an l-value. It's addressable and it has a name
int &lRef = x; // l-value reference
lRef = 10;

R-Value references
int &&rReference = 200; // R-value reference
tReference = 300;       // Change rReference to 300;
int &&xReferernce = x; // We can't assign an L-Value to it - it gives a compiler error
```

```
// Dealing with this in functions
void func(int &&num);
func(200);      // This is okay - it's an r-value
func(x);        // this won't be okay, because x is an l-value

// Overloading them
void func(int &num);        // Pass my integer by reference
void func(int &&num);       // Pass my integer as a r-value

Now we can either use x or 300

// Move constructor
class Move{
    private:
        int *data;

    public:
        void setData(int d){*data = d;)      // Setter
        int getData(){return *data;}         // Getter
        Move(int d);                         // Constructor
        Move(const Move &source);            // Copy Constructor
        ~Move();                             // Destructor
}

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

// The copy constructors would be called to copy these temps - this can be inefficient since it will do a LOT of copies
Vector<Move> vec;
vec.push_back(Move{10});
vec.push_back(Move{20});

// What does the move constructor do?
Instead of making deep copies, it moves resources
Copies the address of the source to the destination and nulls out the source
Very efficient

// Syntax for r-value references - note cannot be const
Type::Type (Type &&source);
Player::Player(Player &&source);

class Move{
    private:
        int *data;

    public:
        void setData(int d){*data = d;)      // Setter
        int getData(){return *data;}         // Getter
        Move(int d);                         // Constructor
        Move(const Move &source);            // Copy Constructor
        Move(Move &&source);                 // Move constructor
        ~Move();                             // Destructor
}

Move::Move(Move &&source)
    : data{source.data}{ // Steal the data
        source.data = nullptr; // null the original data - if we don't do this we end up being a shallow copy instead
        // and we will have pointers all over the place
}

// This time these will call the Move constructor since they are r-values being passed in
Vector<Move> vec;
vec.push_back(Move{10});
vec.push_back(Move{20});
```

```cpp
//====================================
// Move copy constructor example program
//====================================

#include <iostream>
#include <vector>
#include <string>
#include <cmath>
#include <ctime>
//#include "../../std_lib_facilities.h"

class Move {
private:
    int* data;
public:
    void setDataValue(int d) { *data = d; }
    int getDataValue() { return *data; }
    Move(int d);                        // constructor
    Move(const Move& source);           // Copy Constructor
    Move(Move&& source) noexcept;       // Move constructor
    ~Move();                            // Destructor
};

// Constructor to create the object
Move::Move(int d) {
    data = new int;
    *data = d;
    std::cout << "Constructor for: " << d << std::endl;
}

// Copy constructor - this is doing a deep copy
Move::Move(const Move& source)
    :Move{ *source.data } {
    std::cout << "Copy Cosntructor - deep copy for: " << *data << std::endl;
}

// Move constructor
Move::Move(Move&& source) noexcept
    :data{ source.data } {
    source.data = nullptr;
    std::cout << "Move constructor - moving the resource: " << *data << std::endl;
}

Move::~Move() {
    if (data != nullptr) {
        std::cout << "Destructor freeing data for: " << *data << std::endl;
    } else {
        std::cout << "Destructor freeing data for nullptr." << std::endl;
    }
    delete data;
}

int main(){
    std::vector<Move> moveObjects;
    moveObjects.push_back(Move{ 20 });
    moveObjects.push_back(Move{ 30 });
    moveObjects.push_back(Move{ 40 });
    moveObjects.push_back(Move{ 50 });
    moveObjects.push_back(Move{ 60 });
    return 0;
}
```

```
// The 'this' pointer
This
- reserved keyword
- Contains the address of the object - so it's technically a pointer to the object
- Can only be used in the class scope
- All member access is done via the this pointer
- Can be used by the programmer to:
-- Access data members and methods
-- Determine if two objects are the same
-- Deferenced to give us the current object

// ways to write classes
void Account::setBalance(double balance){
    this->balance = balance;
}

// Check if two objects are the same
int Account::compareBalance(const Account &other){
    if (this == &other){
        std::cout << "They are the same" << std::endl;
    }
}

yayaAccount.compare(yayaAccout);

//====================================
// Using const with classes
//====================================

We can pass arguments to class member methods as 'const'
We can also create const objects
What happens if we call member funtions on const objects? Maybe bad stuff. We need to be const correct

// Creating a const object
const Player villian{"Villain", 100, 55};       // this creates a const object and the attributes cannot be changed

// Calling member methods
Even when you call getter methods, the compiler sees that it could potentially change data and it won't compile
void displayPlayerName(const Player &p){
    std::cout << p.getName() << std::endl;
}

displayPlayerName(villain);          // This will fail because the compiler sees that it could potentially change something

// How do we solve this?
Mark the methods we want to use as const
class Player{
    private:
    // stuff
    public:
    std::string getName() const;
}

Now the calls to the getters will work
```

```cpp
//====================================
// Const-correctness example program
//====================================

#include <iostream>
#include <vector>
#include <string>
#include <cmath>
#include <ctime>
#include "../../std_lib_facilities.h"

class Player {
private:
    std::string name;
    int health;
    int xp;

public:
    // This could change the object - we have to qualify it as const. We promise the compiler we won't change it
    std::string getName() const { return name; }
    void setName(std::string name) { this->name = name; }

    // Constructors
    Player();
    Player(std::string name);
    Player(std::string name, int health, int xp);
    ~Player() { std:cout << "Destructor called for: " << name << std::endl; };
};

Player::Player()
    : Player{"None", 0, 0} {
    std::cout << "No args constructor called" << std::endl;
}

Player::Player(std::string name)
    :Player{ name, 0, 0 } {
    std::cout << "Two args constructor for: " << name << " called" << std::endl;
}

// remember that the constructor with the most arguments is the one everyone else delegates to
Player::Player(std::string name, int health, int xp)
    : name{ name }, health{ health }, xp{ xp } {
    std::cout << "Three args constructor for: " << name << " called" << std::endl;
}

// This function with the const qualifier - will fail
void displayPlayerName(const Player &source) {
    std::cout << source.getName() << std::endl;
}

int main(){
    // villian's attributes cannot be changed
    const Player villain{ "Villain", 100, 55 };
    Player hero{ "Hero", 100, 15 };

    std::cout << villain.getName() << std::endl;
    std::cout << "The name is: " << hero.getName() << std::endl;
    return 0;
}
```

```
// Static class members
Class members can be declared as static
- The data member belongs to the class, not any specific object
- Useful to store class-wide information

Class functions can also be static
- Independant of any objects
- Can be called using the class name

// Set up the class in the header file
class Player{
    private:
    static int numberOfPlayers;

    public:
    static int getNumberOfPlayers();

    Player()
}

// Initialize the static member in the .cpp file
int Player::numberOfPlayers = 0;

// Implement the member function as well in the .cpp file
int Player::getNumberOfPlayers(){
    return numberOfPlayers;
}

// Update the constructor to add 1 to the static member
Player::Player(std::string name, int health, int xp)
    :name{name}, health{health}, xp{xp}{
    ++numberOfPlayers;
}
NOTE: Make sure that you only increment in one place - make the other constructors delegate!

// Now we need to make our own destructor
Player::~Player(){
    --numberOfPlayers;
}

Now we can call these static member methods
//==================================
// Static class members Example program
//==================================

// Player.h file
#pragma once
#include <string>

class Player {
private:
    std::string name;
    int health;
    int xp;
    static int numberOfPlayers; // Note that this cannot be initialized in here
public:
    // Methods
    static int getNumberOfPlayers(); // This only has access to other static things

    // three arg constructor- delegate to this one, also using defaults
    Player(std::string name = "None", int health = 0, int xp = 0);

    // Copy constructor
    Player(const Player& source);

    // Destructor
    ~Player();
};
```

```cpp
// Player.cpp file
#include "Player.h"
#include <iostream>

// Initialize the static variable
int Player::numberOfPlayers{ 0 };

// Add to the static member in the constructor
Player::Player(std::string name, int health, int xp)
    : name{ name }, health{ health }, xp{ xp } {
    ++numberOfPlayers; // Add one to the number of players
}

// Delegate from the copy constructor to the three args constructor
Player::Player(const Player& source)
    : Player{source.name, source.health, source.xp} {
}

// Destructor - delete one player when this is called
Player::~Player() {
    std::cout << "Calling destructor for: " << name << std::endl;
    --numberOfPlayers; // Make sure that we clean up the players
}

int Player::getNumberOfPlayers() {
    return numberOfPlayers;
}

// Main.cpp file
#include <iostream>
#include <vector>
#include <string>
#include <cmath>
#include <ctime>
#include "../../std_lib_facilities.h"
#include "Player.h"

void displayActivePlayers() {
    std::cout << "Active players: " << Player::getNumberOfPlayers() << std::endl;
}

int main(){
    displayActivePlayers();
    Player josie{ "Josie", 100, 100 };
    displayActivePlayers();
    {
        Player yaya{ "Yaya", 10,10 };
        displayActivePlayers();
    }
    displayActivePlayers();
    Player Bab{}; // Now with the defaults a no args object will work

    Player* enemy = new Player{ "Enemy", 1000, 1000 };

    displayActivePlayers();
    delete enemy;

    displayActivePlayers();
    return 0;
}

// Structs versus classes
Structs are from C
Essentially the same as classes, but public by default

struct Person{
    std::string name;
    std::string getName(); // this is public be default
}
```

```
// When to use which?
struct:
- Use for passive objects with public access
- don't declare methods in a struct

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

// Friends of a class
Friend:
- A function or class that has access to private class members
- that function 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

// Controversy
Do friends enhance or detract from encapsulation

// The big picture:
Friendship must be granted. It cannot be taken
- Declared explicitly in the class that is granting it
- Declared in the function prototype with the keyword friend

Friendship is not symmetric
- Must be explicitly granted
- A can be a friend of B, but that doesn't necessarily mean B is a friend of A

Friendship is also not transitive
- A is a friend of B
- B is a friend of C
- A is not automatically a friend of C

Friendship is also not inherited

// Examples
class Player{
    // This stuff defaults to private
    friend void displayPlayer(Player &source);

    public:
    // stuff here
}

That function can now directly access the private attributes
void displayPlayer(){
    std::cout << p.name << std::endl;
} //It can change private data members as well.

// Declaring the method of another class as a friend:
Player{
    friend void OtherClass::displayPlayer(Player &source);
}

Now the displayPlayer method in the other class can access things in this class

// Declare an entire class as a friend
Class Player{
    friend class OtherClass;
}
```