Introductory Robot Programming

ENPM809Y

Lecture 08 – Object-oriented Programming (OOP) – Part II

Zeid Kootbally
zeidk@umd.edu

Fall 2020

Overview I

Overview II

Highlights

- 2nd lecture on OOP.
- We will learn about copy and move semantics.
- This lecture is the most difficult one so far but it is also the only really difficult lecture in this course.
- Before we start, create a new project and copy main.cpp, villain.cpp, and villain.h from ELMS into this project.
  - ➤ CLion:Lecture8

Abstraction

- Reminder: Encapsulation is related to data hiding. This is usually done by making attributes `private` and then hide your method implementations to the users. You can later completely change the code for these methods while the users' code stay unchanged.
- While encapsulation happens at the implementation level, abstraction works at the design level. When you design your class, you need to decide what to provide to the users. For instance, you may give users the ability to change some attributes by providing setters for these attributes. If you do not want users to change these attributes then do not provide setters for these attributes. In abstraction, you make a decision on what to provide and what not to provide to the users.

The explicit Keyword

- The compiler can use constructors callable with a single parameter to convert from one type to another in order to get the right type for a parameter.

```
class Villain {
public:
    Villain(int life):life_{life}{}
};
```

```
void TakeVillain(game::Villain villain){
}

int main(){
    TakeVillain(1);
}
```

- The argument is not a `Villain` object, but an `int`. However, there exists a constructor for `Villain` that takes an `int` so this constructor can be used to convert the parameter to the correct type.
- `explicit` can be used to forbid such behavior.

The Hidden this Pointer

- Question: When a method is called, how does C++ keep track of which object it was called on?.
- Answer: C++ utilizes a hidden pointer named `this`.

```
int main{
    game::Villain astaroth{"Astaroth",300,3};
    astaroth.TakeDamage(25.5);
}
```

- When `astaroth.TakeDamage(25.5);` is called, C++ knows that method `TakeDamage()` should operate on object `astaroth`, and that `name_` and `health_` inside `TakeDamage()` refer to `astaroth.name_` and `astaroth.health_`

The Hidden this Pointer

- Consider the following statement from the previous slide:
  ```
  astaroth.TakeDamage(25.5);
  ```
- Although the call to `TakeDamage(25.5)` looks like it only has one argument, it actually has two!
- When compiled, the compiler converts `astaroth.TakeDamage(25.5)` into the following:
  ```
  TakeDamage(&astaroth, 25.5);
  ```
- Note that this is now just a standard function call, and the object `astaroth` (which was formerly an object prefix) is now passed by address as an argument to the function.

The Hidden this Pointer

- Since the function call now has an added argument, the member function definition needs to be modified to accept (and use) this argument as a parameter. Consequently, the following member function:

```
void Villain::TakeDamage(double damage){
    if (!IsDead()) {
        health_ -= damage;
    } else
        std::cout << name_ << " is dead!!" << std::endl;
}
```

- is converted by the compiler into:

```
void Villain::TakeDamage(Villain* const this, double damage){
    if (!this->IsDead()) {
        this->health_ -= damage;
    } else
        std::cout << this->name_ << " is dead!!" << std::endl;
}
```

The Hidden this Pointer

- Most of the time, you never need to explicitly reference the `this` pointer. However, there are a few occasions where doing so can be useful:
  - If you have a constructor (or method) that has a parameter with the same name as an attribute, you can disambiguate them by using `this`.

```
void set_name(name){
    this->name = name;
}
```

    - Note: If you follow the Google C++ guideline you should not find yourself in this situation.
  - You can check if objects are the same using `this`.

```
bool Villain::Compare(const Villain& v){
    return (this == &v);
}
```

constness with Classes

- Reminder: : The `const` keyword is used to make any element of a program constant.
- `const` objects: Ensure that no attributes of the object are changed during the object's lifetime.
- `const` methods: To ensure `const`ness, methods called for `const` objects must also be declared as `const`.
  - A method that is not specifically declared `const` is treated as one that will modify data members in an object, and the compiler will not allow you to call it for a `const` object.

constness with Classes

- We can make an object `const` by writing the `const` keyword at the beginning of the object declaration.
  ```
  const game::Villain firebrand;
  ```
- Using a non-`const` setter:
  ```
  firebrand.set_name("Firebrand");
  ```
  ```
  error: passing const game::Villain as this argument ...
  ```
- Using a non-`const` getter:
  ```
  firebrand.get_name();
  ```
  ```
  error: passing const game::Villain as this argument ...
  ```
- Question: How to fix these errors?

constness with Classes

- Answer: Tell the compiler that specific methods will not modify the object by declaring these methods as `const`.

```
class Villain{
public:
    std::string get_name() const {return name_;};
    void set_name(std::string name) const {name_ = name};
};
```

- Now we have:

```
const game::Villain firebrand;
firebrand.get_name();//--no error
firebrand.set_name("Firebrand");
```

```
error: no match for operator=...name_=name;
```

constness with Classes | Explicit Reference

- Question: What will happen with the following code?

```
class Villain{
public:
    std::string get_name() const{
        name_ = "none";
        return name_;
    }
};
```

- Question: What will happen with the following code?

```
class Villain{
public:
    std::string get_name() const{
        return name_;
    }
};
```

```
int main(){
    game::Villain astaroth;//--non-const
    astaroth.get_name();
}
```

constness with Classes | Rules

- Rule #1: If an object is declared `const` then it can only be called by `const` methods.
- Rule #2: A `const` method can be called on `const` and non-`const` objects.
- Rule #3: Keyword `const` must appear in both method declaration (.h) and definition (.cpp).
- *Best practice*: Any method that does not modify objects' attributes should be declared as `const` (all getters should be `const`).

friend Function and Class

- One of the important concepts of OOP is data hiding, i.e., a non-member function cannot access an object's `private` or protected attributes.
- This restriction sometimes force programmers to write long and complex codes.
- There is mechanism built in C++ to access `private` data from non-member functions.
- This is done using a friend function or/and a friend `class`.
- Friendship must be granted, not taken. Friendship must be declared explicitly in the class that is granting friendship.

friend Function and Class | friend Function

- A friend function is a function that can access the private members of a class as though it were a member of that class.
- A friend function may be either a non-member function or a method of another class.
- To declare a friend function, simply use the `friend` keyword in front of the prototype of the function you wish to be a friend of the class. It does not matter whether you declare the friend function in the `private` or `public` section of the class.
- We will first look at how to create a non-member `friend` function and then how to make a method of another class a `friend` function.

friend Function and Class | friend Function | Normal Function

- In villain.h

```
namespace game{
    class Villain{
        public:
        friend void Print(const Villain& villain);//--either public or private
    };//--class Villain
}//--namespace game
```

- In villain.cpp

```
#include "villain.h"
namespace game {
    void Print(const Villain& villain){
        std::cout << "Name: " << villain.name_ << std::endl;
    }
}//--namespace game
```

- In main.cpp

```
#include "villain.h"
int main(){
    const game::Villain firebrand{"Firebrand"};
    Print(firebrand);
}
```

friend Function and Class

Todo

Declare a `friend` function `Print()` in the `Villain` class and implement it in villain.cpp

friend Function and Class | friend Function | Function from Class

- In villain.h

```
namespace game {
    class Villain;//--forward declaration

    class Display{
    public:
        void PrintName(const Villain& villain);
    };//--class Display

    class Villain {
    public:
        friend void Display::PrintName(const Villain& villain);
    };//--class Villain
}//--namespace game
```

- In villain.cpp

```
#include "villain.h"
namespace game {
    void Display::PrintName(const Villain& villain){
        std::cout << "Name: " << villain.name_ << std::endl;
    }
}//--namespace game
```

friend Function and Class

Todo

Implement a class `Display` with the method `PrintName()`
Make `PrintName()` a `friend` function of `Villain`

friend Function and Class | friend Class

- When a class is made a friend class, all the methods of that class become friend functions.
- In the next slide, `Villain` will make `Display` a friend class.
- All methods of `Display` will be friend functions of `Villain`.
- Thus, any method of `Display` can access the private attributes of `Villain`.
- However, methods of class `Villain` cannot access the members of `Display` because `Display` did not grant friendship to `Villain`.

friend Function and Class | friend Class

- In villain.h

```
namespace game {
    class Villain;//--forward declaration

    class Display{
    public:
        void PrintName(const Villain& villain);
        void PrintHealth(const Villain& villain);
    };//--class Display

    class Villain {
    public:
        friend class Display;
    };//--class Villain
}//--namespace game
```

- In villain.cpp

```
#include "villain.h"
namespace game {
    void Display::PrintName(const Villain& villain){
        std::cout << "Name: " << villain.name_ << std::endl;
    }
    void Display::PrintHealth(const Villain& villain){
        std::cout << "Health: " << villain.health_ << std::endl;
    }
}//--namespace game
```

friend Function and Class

Todo

Implement a class `Display` with two methods: `PrintName()` and `PrintHealth()`

Make `Display` a `friend` class of `Villain`

Operator Overloading

- Operators have different implementations depending on their arguments.
- Operator overloading is generally defined by a programming language, a programmer, or both.
- By overloading operators for user-defined type (`class` and `struct`) you can make `class` and `struct` objects behave like primitive data types.
- Besides the assignment operator (=), operators are not overloaded automatically for user-defined types. You have to overload them explicitly.

Operator Overloading

- Suppose we have a class `Number` that models any number.

```
int main() {
    Number a{3}, b{2,3}, c{3.14};
    Number result = (a+b)/c;
}
```

- To use the operators `+` and `/` with objects from `Number` you have to tell the compilers what to do when you use `+` and `/` on two `Number` objects.
- You have to overload `+` and `/` for `Number` objects.

Operator Overloading | Basic Rules

- The first and basic rule of operator overloading is: we can overload unary operator as only unary operator, it cannot be overload as binary operator and vice versa.
- We cannot overload those operators that are not a part of C++ language, e.g., $.
- We can perform operator overloading in only user defined classes. We cannot change the operators existing functionality.
- The following operators cannot be overloaded:
  - `::` Scope resolution operator.
  - `.` Class membership operator.
  - `?:` Ternary or conditional operator.
  - `.*` Pointer to member operator.
  - `->*` Pointer to member operator.

Operator Overloading | Stream Insertion and Extraction

- The stream insertion << and stream extraction >> operators also can be overloaded to perform input and output for user-defined types like an object.
- These operators will allow us to insert and extract our midstream objects to and from streams exactly as we've been doing with all of the other builtin types.

```
int main() {
    game::Villain firebrand;
    std::cin >> firebrand;
    std::cout << firebrand << std::endl;
}
```

```
Enter name: Firebrand
Enter health points: 100
Enter number of lives: 1
Name: Firebrand, Health: 100, Life: 1
```

Operator Overloading | Insertion Operator

- `operator <<` is a non-member function that has two parameters:
  - The first parameter is an output stream object by reference `std::ostream`.
  - The second parameter is a reference to the object whose data we want to insert in the output stream. Note it's a `const` because it's read only.

```
std::ostream& operator << (std::ostream &out, const Villain& v){
    out << "Name: " << v.name_
    << ", Health: " << v.health_
    << ", Life: " << v.life_;
    return out;
}
```

Operator Overloading | Extraction Operator

- `operator >>` is a non-member function that has two parameters.
  - The first parameter is a reference to an input stream whose type is `std::istream`.
  - The second parameter is a reference to the object we want to extract data into. Note that it's not `const` because we want to modify its attributes.
  - We can also make it a `friend` function so we can work directly with the attributes.

```
std::istream& operator >> (std::istream &in,  Villain &v){
    std::cout << "Enter name: ";
    in >> v.name_;
    std::cout << "Enter health points: ";
    in >> v.health_;
    std::cout << "Enter number of lives: ";
    in >> v.life_;
    return in;
}
```

Operator Overloading

Todo

Overload the insertion and the extraction operators for the class `Villain`

Rule of Five

- The rule of five (also known as the Law of The Big Five or The Big Five) is a rule of thumb in C++ that claims that if a class defines any of the following then it should probably explicitly define all five:
  1. Copy constructor
  2. Copy assignment operator
  3. Move constructor
  4. Move assignment operator
  5. Destructor

Copy Semantics

- So far we have created objects directly by calling the correct constructor.

```
int main(){
    game::Villain firebrand{"Firebrand"};
    game::Villain zombie{"Zombie", 50, 1};
}
```

- There are situations where you want to make a copy of an object. In C++, objects are copied when:
  - They are passed by value to functions (copy constructor).
  - They are returned by value from functions (copy constructor).
  - They are created from other objects of the same class (copy constructor).
  - They are assigned from other objects (copy assignment operator).

Copy Semantics | Copy Constructor

```
game::Villain CreateFinalBoss(){
    game::Villain boss{"Lucifer",2000,5};//--boss is a variable local to the function
    return boss;//--copy of boss is returned.
}
void PrintVillain(game::Villain villain){
    //--villain is a copy of the object firebrand in this example.
    //--villain is used in the body of the function.
    std::cout << villain << std::endl;
    //--Destructor for villain will be called when the function ends.
}

int main(){
    game::Villain firebrand{"Firebrand"};
    PrintVillain(firebrand);//--firebrand is passed by value, copy is made in the function
    game::Villain boss = CreateFinalBoss();//--result of CreateFinalBoss() is copied into boss
    game::Villain lucifer{boss};//--lucifer is a copy of boss
}
```

Copy Semantics | Copy Constructor | Shallow Copy

- In the previous slide, copying and object was performed 3 times.
- Each time a copy was made the compiler implicitely called a copy constructor which copied each attribute from the source object to the new object.
- The default method of the copy constructor used by the compiler is called shallow copy:

```
Villain (const Villain& source);//prototype
```

  - Note: We pass the `source` by `const &`, why?
- A shallow copy duplicates as little as possible.
  - This is perfectly fine in most cases.
  - Issues arise when your class has raw pointer attributes which allocate memory on the heap.

Copy Semantics | Copy Constructor

```
//--returning an object from a function
game::Villain CreateFinalBoss(){
    game::Villain boss{"Lucifer",2000,5};
    return boss;
}
```

```
//--passing an object by value to a function
//--calling PrintVillain(firebrand)
void PrintVillain(game::Villain villain){
    std::cout << firebrand << std::endl;
}
```

```
//--instatiating an object from another object
game::Villain boss{"Lucifer"};
game::Villain lucifer{boss};
```

```
//--implicit copy constructor
Villain (const Villain &boss):
tmp.name_{boss.name_},
tmp.health_{boss.health_},
tmp.life_{boss.life_},
tmp.original_health_{boss.health_}{}
```

```
//--implicit copy constructor
Villain (const Villain &firebrand):
villain.name_{firebrand.name_},
villain.health_{firebrand.health_},
villain.life_{firebrand.life_},
villain.original_health_{firebrand.health_}{}
```

```
//--implicit copy constructor
Villain (const Villain &boss):
lucifer.name_{boss.name_},
lucifer.health_{boss.health_},
lucifer.life_{boss.life_},
lucifer.original_health_{boss.original_health_}{}
```

Copy Semantics | Copy Constructor | Copy-Elision and RVO

- Return Value Optimization (RVO) and Copy-Elision are in C++ since C++98.
- RVO basically means the compiler is allowed to avoid creating temporary objects for return values, even if they have side effects.
- In the example below, an object is returned but is nevertheless not copied.

```
//--returning an object from a function
game::Villain CreateFinalBoss(){
    game::Villain boss{"Lucifer",2000,5};
    return boss;
}
```

- RVO is part of a larger group of optimizations called copy-elision.
- More examples can be found here.

Copy Semantics | Copy Constructor

- As mentioned earlier:
  - A shallow copy duplicates as little as possible and this is perfectly fine in most cases.
  - Issues arise when your class has raw pointer attributes which allocated memory on the heap.
- Consider the addition of a new attribute which is a raw pointer to an `int` on the heap.

```
private:
    std::string name_;
    double health_;
    int life_;
    double original_health_;
    int *strength_ptr_;//--represent the strength of a Villain
```

- The first thing we need to do is to change the constructor to initialize this new attribute.
- The second thing is to update the destructor to deallocate memory when the pointer is destroyed.

Copy Semantics

Todo

1. Update the constructor to initialize `strength_ptr_`
2. Update the destructor to deallocate the memory pointed by `strength_ptr_`

Copy Semantics | Copy Constructor

1. Update the constructor.

```
Villain(std::string name="Villain", double health=100, int life=1, int strength=10 ):
    name_{name},
    health_{health},
    life_{life},
    original_health_{health_},
    strength_{nullptr}{
        strength_ptr_ = new int;
        strength_ptr_* = strength;
        }
```

2. Update the destructor.

```
~Villain(){
    delete strength_ptr_;
}
```

- When a `Villain` object goes out of scope, the destructor is called, which in turn deallocates memory pointed by `strength_ptr_`.

Copy Semantics | Copy Constructor | Shallow Copy

- The following code snippet creates two objects: `firebrand` and `firebrand_copy`. The constructor will be called to create `firebrand` and the copy constructor will be called to create `firebrand_copy`.

```
game::Villain firebrand{"Firebrand"};
game::Villain firebrand_copy{firebrand};
```

- The following happens in the default copy constructor:

```
Villain (const Villain& firebrand):
    firebrand_copy.name_{firebrand.name_},
    firebrand_copy.health_{firebrand.health_},
    firebrand_copy.life_{firebrand.life_},
    firebrand_copy.original_health_{firebrand.original_health_},
    firebrand_copy.strength_ptr_{firebrand.strength_ptr_}{
    /*body*/
}
```

- To visualize what happens when a copy is made, let's replicate the default copy constructor provided by the compiler.

Copy Semantics

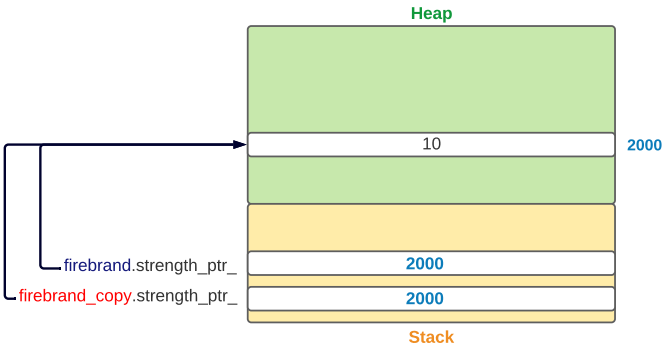
Todo

Let's write the exact same copy constructor which would be provided by the compiler if we did not provide one.

Note: This copy constructor does shallow copy.

Copy Semantics | Copy Constructor | Shallow Copy

`firebrand_copy.strength_ptr_{firebrand.strength_ptr_}`



- With shallow copy, only a copy of the pointer is made and the data the pointer is pointing to is not copied. So what?

Copy Semantics | Copy Constructor | Shallow Copy

- The copy constructor will also be called in the following case because a copy of `firebrand` needs to be made in the body of `PrintVillain()`.

```
game::Villain firebrand{"Firebrand"};
PrintVillain(firebrand);
```

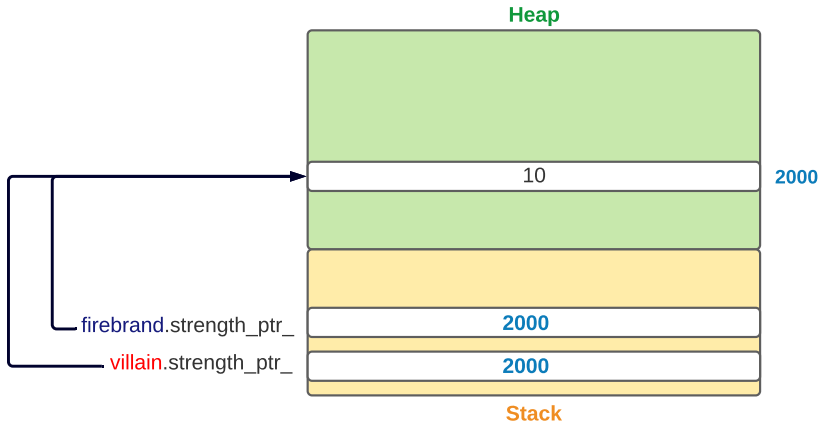
- The copy constructor is called as follows (based on the definition of `PrintVillain()` in slide 36):

```
1 Villain (const Villain& firebrand):
2     villain.name_{firebrand.name_},
3     villain.health_{firebrand.health_},
4     villain.life_{firebrand.life_},
5     villain.original_health_{firebrand.original_health_},
6     villain.strength_ptr_{firebrand.strength_ptr_}{
7     /*body*/
8 }
```

- Let's take a closer look at line 6.

Copy Semantics | Copy Constructor | Shallow Copy

villain.strength_ptr_{firebrand.strength_ptr_}

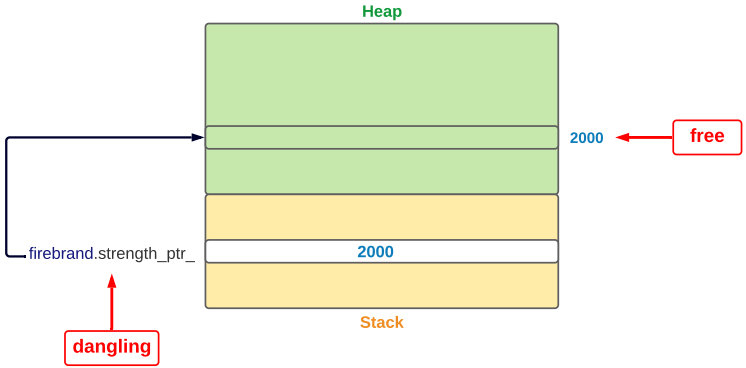


Copy Semantics | Copy Constructor | Shallow Copy

- `PrintVillain(firebrand)` creates the local variable `villain`, which is a copy of `firebrand`.
- `villain.strength_ptr_` points to the same memory pointed by `firebrand.strength_ptr_`
- When the call to `PrintVillain(firebrand)` terminates, `villain` goes out of scope and `~Villain()` is called (`delete` is called on `villain.strength_ptr_`).
  - Memory pointed by `villain.strength_ptr_` is deallocated.
  - `firebrand.strength_ptr_` is still pointing at this address.
  - `firebrand.strength_ptr_` becomes a dangling pointer.
  - Your program will probably still write to and access data from that memory address.
  - Your program will call `delete` a second time when `firebrand` goes out of scope.

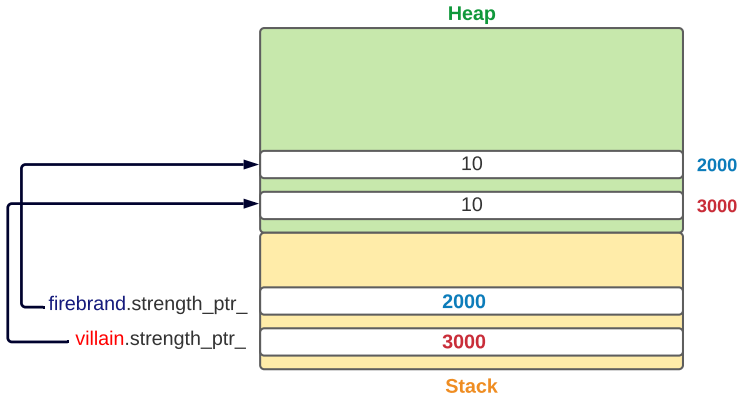
Copy Semantics | Copy Constructor | Shallow Copy

- Result when the call to `PrintVillain(firebrand)` terminates.



Copy Semantics | Copy Constructor | Deep Copy

- What we really want: Copy both the pointer (`villain.strength_ptr_`) and the data it points to (10). This is called deep copy.



Copy Semantics | Copy Constructor | Deep Copy

- Deep copy is a user-provided copy constructor.
- Each copy will have a pointer to unique storage in the heap.
- Write your own copy constructor using deep copy when you have raw pointers as class attributes.
- Next slide shows the implementation of our own copy constructor which performs deep copy.
- When a copy of a `Villain` object is made, the custom copy constructor is called.

Copy Semantics | Copy Constructor | Deep Copy

```
//--explicit copy constructor definition
Villain(const Villain& source):
        name_{source.name_}, health_{source.health_}, life_{source.life_},
        original_health_{source.health_}, strength_ptr_{nullptr}{
            strength_ptr_ = new int;
            *strength_ptr_ = *source.strength_ptr_;
}
```

```
//--in main()
game::Villain firebrand{"Firebrand"};
game::Villain firebrand_copy{firebrand};//--making a copy
```

```
//--behind the scenes...
Villain(const Villain& firebrand):
        firebrand_copy.name_{firebrand.name_},
        firebrand_copy.health_{firebrand.health_},
        firebrand_copy.life_{firebrand.life_},
        firebrand_copy.original_health_{firebrand.health_},
        firebrand_copy.strength_ptr_{nullptr}{
            .firebrand_copystrength_ptr_ = new int;
            *firebrand_copy.strength_ptr_ = *firebrand.strength_;
}
```

Copy Semantics

Todo

Let's write our custom copy constructor which performs deep copy.

Copy Semantics | Copy Assignment Operator

- A very similar situation arises with the assignment operator if you wish to assign to one instance of the class the values of another instance of the class.

```
int main(){
    const game::Villain firebrand{"Firebrand"};
    game::Villain firebrand2{"Firebrand 2"};
    firebrand2 = firebrand;
}
```

- As with the copy constructor, C++ automatically overloads the assignment operator = for new classes, and its action is to copy the member values of the object on the right side of the operator to those on the left side.

Copy Semantics | Copy Assignment Operator

```
int main(){
    game::Villain firebrand{"Firebrand"};
    game::Villain firebrand2{"Firebrand 2"};
    firebrand2 = firebrand;
}
```

- Implicit assignment operator provided by the compiler.

```
Villain& operator= (const Villain& rhs){
    std::cout << "Copy assignment operator called..." << std::endl;
    if (this == &rhs){return *this;} //--if you assign an object to itself
    //--same as copy constructor from here
    name_ = rhs.name_;
    health_ = rhs.health_;
    life_ = rhs.life_;
    original_health_ = rhs.original_health_;
    strength_ptr_ = rhs.strength_ptr_;
    return *this;
}
```

Copy Semantics

Todo:

Write the copy assignment operator as provided by the compiler.

Copy Semantics | Copy Assignment Operator

- Explicit assignment operator provided by the user.

```
Villain& operator= (const Villain& rhs){
    if (this == &rhs){ return *this; } //--assigning object to itself
    //same as deep copy constructor from here
    name_ = rhs.name_;
    health_ = rhs.health_;
    life_ = rhs.life_;
    original_health_ = rhs.original_health_;
    delete strength_;//--get rid of the old data
    strength_ = new int;
    *strength_ = *rhs.strength_;
    return *this//--this is a pointer to an object
                //--*this returns a reference to the object
}
```

Copy Semantics

Todo:

Overload the copy assignment operator to perform deep copy.

Move Semantics

- During the execution of a C++ program the compiler sometimes creates unnamed temporary values.

```
int x{};
x = 1 + 2;
```

- `1 + 2` is evaluated and `3` is stored in an unnamed temporary variable (let's call it `tmp`).
- The value from `tmp`, 3 in this example, is copied into `x` then `tmp` is destroyed. Conceptually, this r-value evaporates by the time the compiler reaches the semicolon at the end of the full expression.
- These `tmp` variables can be very big (e.g., class objects). If these classes contain raw pointer attributes it would be more efficient if `x` could point to this memory, rather than doing deep copy. This will reduce the overhead and speed up our program.

Move Semantics

- C++11 introduced move semantics and the move constructor.
- Move semantics allows an object, under certain conditions, to take ownership of some other object's external resources. This is important in two ways:
  - Turning expensive copies into cheap moves. Note: If an object does not manage at least one external resource (either directly, or indirectly through its attributes), move semantics will not offer any advantages over copy semantics. Highly recommended if you have raw pointer attributes.
  - Implementing safe "move-only" types; that is, types for which copying does not make sense, but moving does. Examples include locks, file handles, and smart pointers with unique ownership semantics.
- Move constructor moves an object rather than copy it. If you do not provide a move constructor the copy constructor will be used.

Move Semantics | l-value and r-value

- l-values and r-values are concepts you do not worry about too much when you are first learning C++.
- As you learn more advanced features of the language (r-value reference and move semantics) they become very important.
- *"A useful heuristic to determine whether an expression is an l-value is to ask if you can take its address. If you can, it typically is. If you can't, it's usually an r-value."* – Effective Modern C++
- l-value : An l-value is an expression that refers to a memory location and allows us to take the address of that memory location via the & operator.
- r-value : We can define r-values by exclusion. Everything that is not an l-value is an r-value.

Move Semantics | l-value and r-value

- Most of the variables are l-values. `i` and `name` are l-values. We know they are addressable since we can use them on the left-hand side of an assignment expression. The compiler will look for their memory address on the stack and will change the value located there.

```
int main() {
    int i{7};                       //--i is an l-value
    std::cout << &i << std::endl; //--i is addressable
    i = 10;                         //--we can modify i

    std::string name{};
    name = "Bjarne";                //--name is an l-value

    200 = i;                        //--200 is NOT an l-value
    (200 +2) = i                    //--200+2 is NOT an l-value
}
```

Move Semantics | l-value and r-value

- Usually r-values are on the right-hand side of an assignment expression.

```
int main() {
    int i{7};               //--7 is an r-value
    int j{i+3};             //--i+3 is an r-value
    int max = Max(20,30);   //--Max(20,30) is an r-value

    std::string name{"Bjarne"};//--Bjarne is an l-value
}
```

- r-values can be assigned to l-values explicitly. l-values can appear both on the left and right sides of an assignment statement.

```
int main() {
    int i{2};
    int j{};

    i = 100; // i is an l-value and 100 is an r-value
    j = i + j; // j is an l-value and i + j is an r-value
}
```

Move Semantics | l-value and r-value | l-value Reference

- All the references we have seen so far are called l-value references, because they reference l-values.

```
void Square(int &n){
    n *= n;
}

int main() {
    int i{2};
    int &ref_i = i;//--ok: i is an l-value
    int &ref_j = 200;//--error: 200 is not an l-value

    Square(i);//--ok: i is an l-value
    Square(200);//--error: expects and l-value for 1st argument
}
```

Move Semantics | l-value and r-value | r-value Reference

- If X is any type, then X&& is called an r-value reference to X.
- Move semantics is all about r-value references.
- You can think of r-value references being references to all the temporary objects (that are not addressable).
- r-value references are used by move constructor and move assignment operator to efficiently move an object rather than copy it.

```
int x{5};             //--x is an l-value

int &l_ref = x;       //--l_ref is an l-value reference
l_ref = 10;           //--change x to 10

int &&r_ref = 200;    //--r_ref is an r-value reference
r_ref = 300;          //--change 200 to 300

int &&x_ref = x;      //--error: cannot bind rvalue reference of type
                      //int&& to lvalue of type int
```

Move Semantics | l-value and r-value | l-value Reference Parameter

- We know how to use l-value references as function parameters.

```
void Function(int& value){};

int main(){
    int x{5};       //--x is an l-value
    Function(x);    //--ok, x is an l-value
    Function(200); //--error: cannot bind non-const lvalue reference
                   //of type int& to an rvalue of type int
}
```

Move Semantics | l-value and r-value | r-value Reference Parameter

- Lets' look at r-value references in the context of function parameters.

```
void Function(int&& value){};

int main(){
    int x{5};       //--x is an l-value
    Function(200); //--ok, 200 is an r-value
    Function(x);   //--error: cannot bind rvalue reference of type
                   //int&& to lvalue of type int
}
```

Move Semantics | l-value and r-value | l-value and r-value Reference Parameter

- We can have overloaded functions for r-value and l-value reference parameters. The compiler will call the correct one based on the argument passed to the function.

```
1  void Function(int& value){};
2  void Function(int&& value){};
3
4  int main(){
5      int x{5};       //--x is an l-value
6      Function(x);    //--Function defined at line 1 is called
7      Function(200); //--Function defined at line 2 is called
8  }
```

Move Semantics

- Sometimes copy constructors are called many times due to the copy semantics.

```
std::vector<game::Villain> vect;
vect.push_back(game::Villain{"Firebrand", 100, 1, 55});
```

- If the source must remain untouched then a copy constructor is what you want. However, in our example, `game::Villain{"Firebrand", 100, 1, 100}` is an r-value: `vect` will make a copy of it in a temporary object, place it in `vect`, and then destroy the temporary object.
- Multiple calls to `push_back()` will make multiple copies and multiple destructions. Each time a copy is made, new memory is allocated on the heap for `strength_ptr_` (because we are using a copy constructor) and the previous memory is deallocated (when `vect` destroys the temporary object).
- Since `game::Villain{"Firebrand", 100, 1, 100}` will be destroyed anyways, it would be more efficient to use the memory it allocated on the heap for `strength_ptr_` instead of copying it.

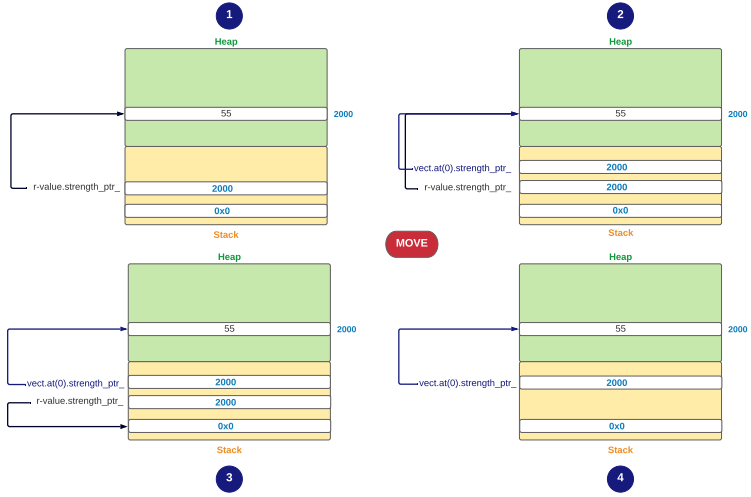
Move Semantics | Move Constructor

- A move constructor enables the resources owned by an r-value object to be moved into an l-value without copying, but by "stealing" the state of the r-value.
- Define a move constructor method that takes an r-value reference to the class type as its parameter.

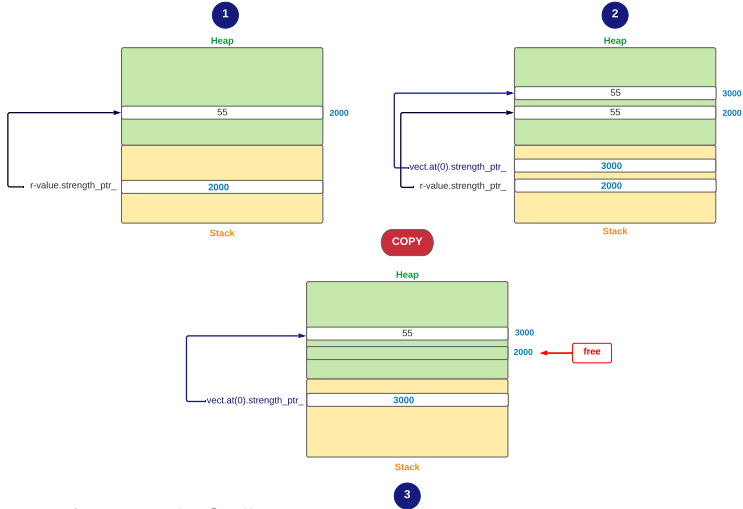
```
Villain(Villain&& source) noexcept:
    name_{source.name_},
    health_{source.health_},
    life_{source.life_},
    original_health_{source.health_},
    strength_ptr_{nullptr}{
        strength_ptr_ = source.strength_ptr_;
        source.strength_ptr_ = nullptr;
}
```

- The move constructor is much faster than a copy constructor because it doesn't allocate memory nor does it copy memory buffers.

Move Semantics | Move Constructor



Move Semantics | Move Constructor



Move Semantics | Move Assignment Operator

- The copy assignment operator that we overloaded earlier works with l-value references.
- The move assignment that we will overload in this section works with r-value references (temporary unnamed objects).
- You do not have to provide a move assignment operator and the compiler will use the copy constructor by default.
- However, you should implement the move assignment operator if you have raw pointer attributes.

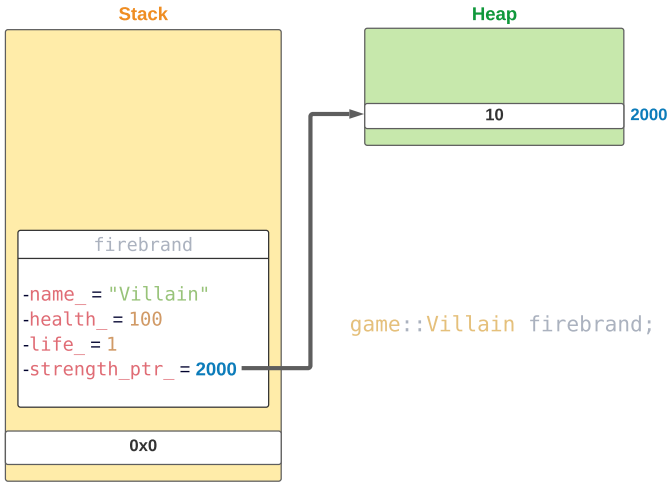
```
int main(){
    game::Villain firebrand;//--ctor is called
    //--copy assignment operator is called
    firebrand = game::Villain{"Firebrand", 100, 1, 100};//-- l-value = r-value
}
```

Move Semantics | Move Assignment Operator

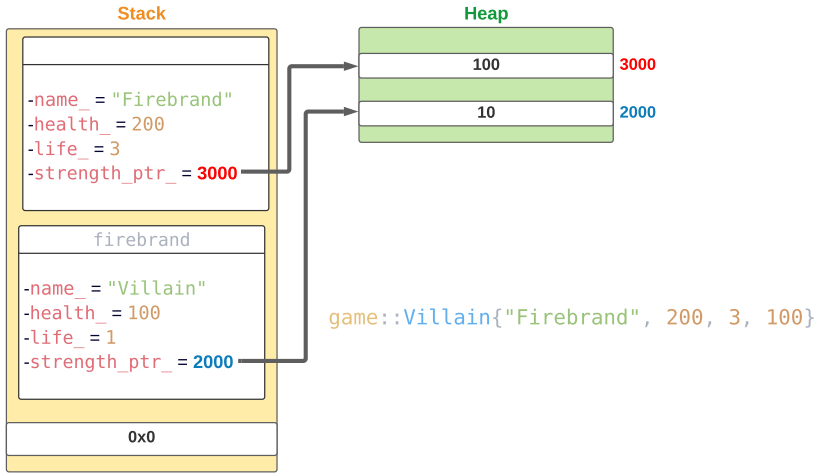
- The copy assignment operator that we overloaded earlier works with l-value references. The move assignment that we will overload in this section works with r-value references (temporary unnamed objects).
- You do not have to provide a move assignment operator and the compiler will use the copy constructor by default. However, you should implement the move assignment operator if you have raw pointer attributes.

```
Villain& operator= (Villain&& rhs) noexcept {
    if (this == &rhs){ return *this; }//--self assignment?
    name_ = rhs.name_;
    health_ = rhs.health_;
    life_ = rhs.life_;
    original_health_ = rhs.original_health_;
    delete strength_ptr_; // get rid of the old data
    strength_ptr_ = rhs.strength_ptr_;
    rhs.strength_ptr_ = nullptr;
    return *this;
}
```

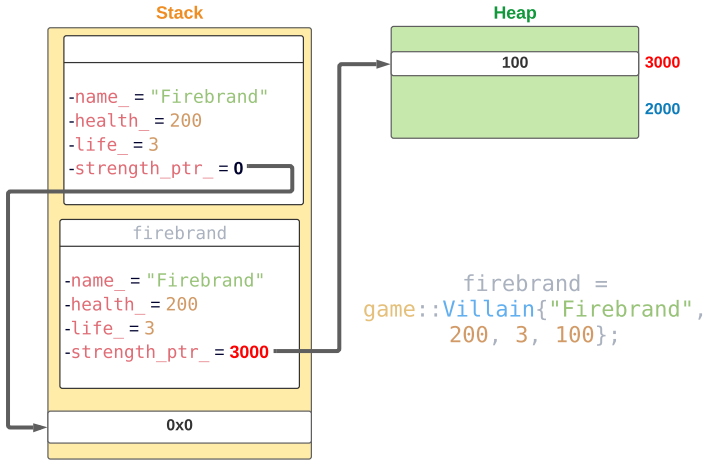
Move Semantics | Move Assignment Operator



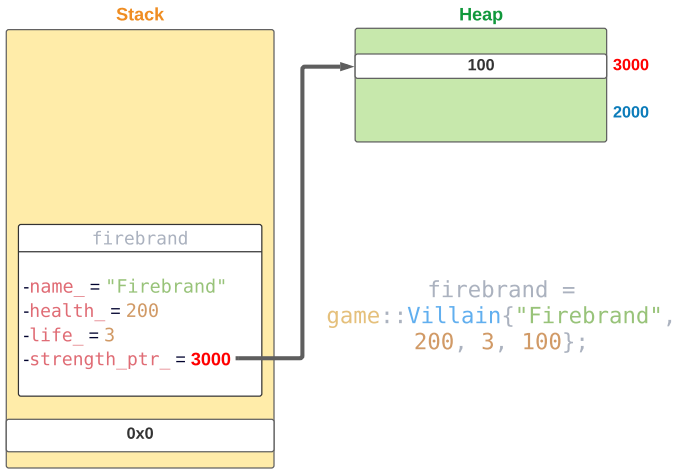
Move Semantics | Move Assignment Operator



Move Semantics | Move Assignment Operator



Move Semantics | Move Assignment Operator



Moral of the Story

- If you write a class which uses any raw pointers, it should always fulfill the rule of 5.
- If your class does not use any raw pointers, do not write these functions. Rely on the ones built into the language.
- If you need dynamic memory allocation, use smart pointers.
  - In our example, you could make `strength_ptr_` a `shared_ptr`.
- Pass objects to functions as `const` & unless you need to be able to change the object being passed in.
- Note: For next lecture we will get rid of the pointer attribute and all the methods created that take care of this pointer.

Next Class | 10/29

- Lecture09: Object Oriented Programming - Part III.
- Stay safe!