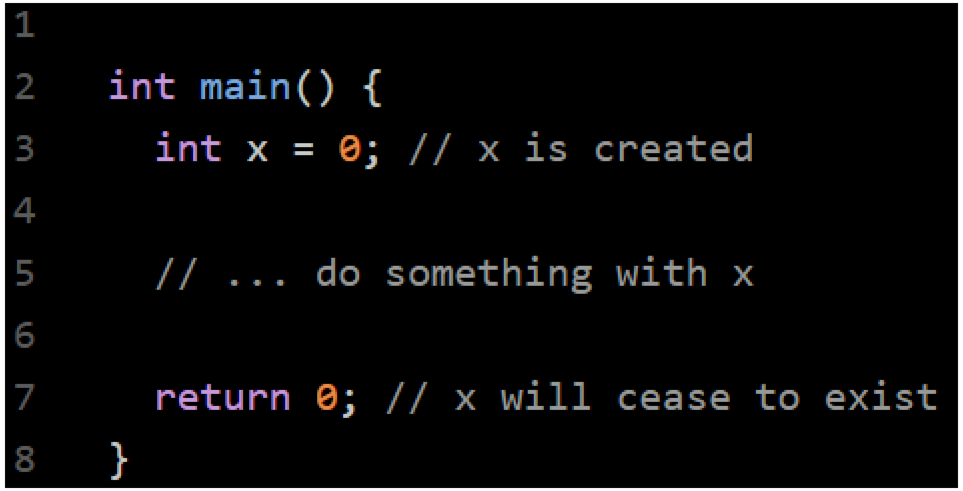
Happy Belated Thanksgiving

Today we will be learning about **classes** and **lab 3**

You can email me: mario.badr@mail.utoronto.ca

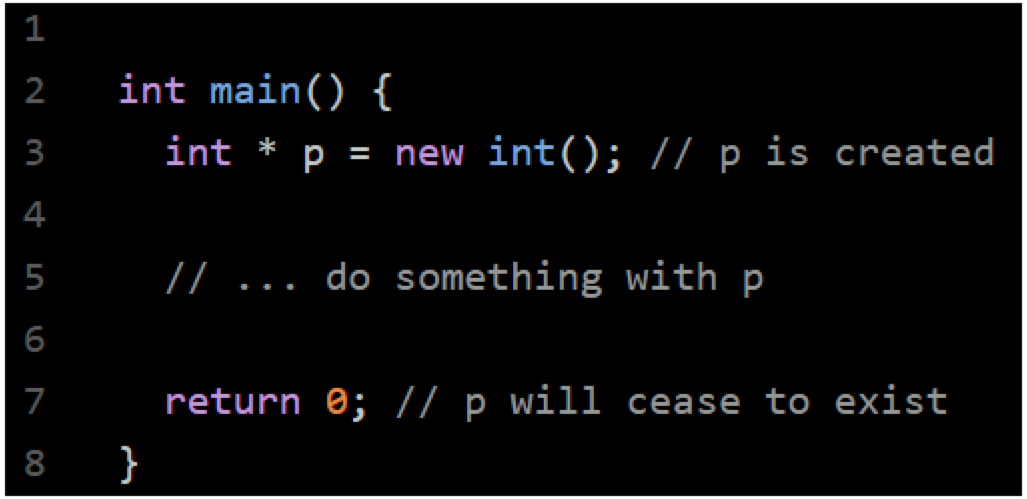
Automatic Variables: Plain Old Data



```

int main() {
  int x = 0; // x is created

  // ... do something with x

  return 0; // x will cease to exist
}
```

- Where does **int** come from?
- How is **x** created?
- How is **x** destroyed?
- What does “**x** is an automatic variable” mean?

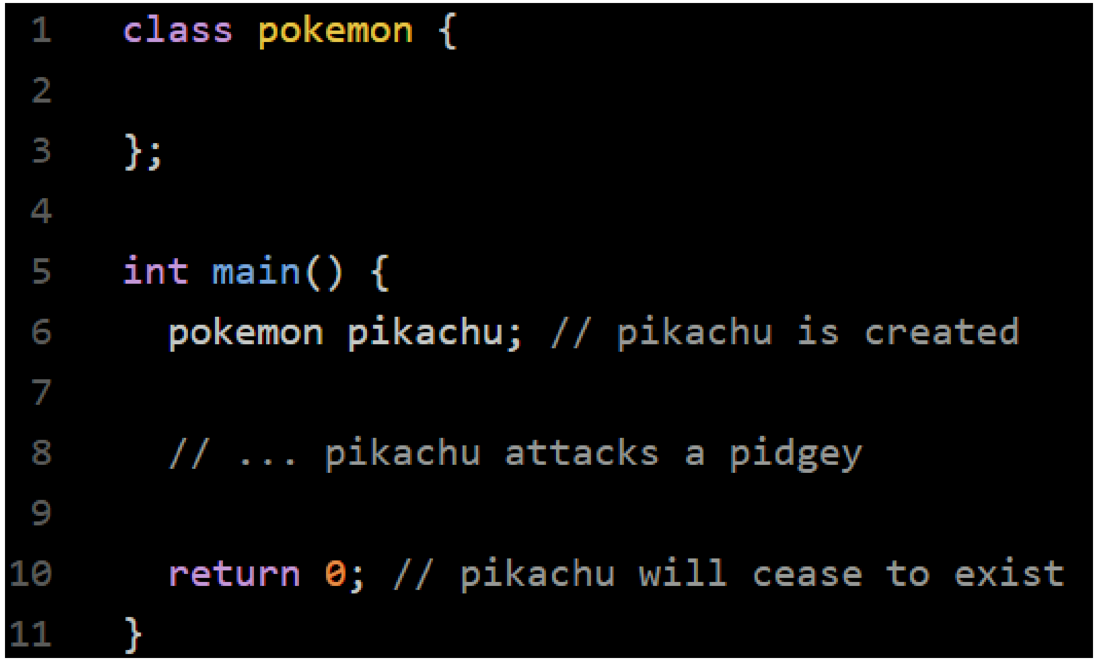
Automatic Variables: Pointers



```

int main() {
  int * p = new int(); // p is created

  // ... do something with p

  return 0; // p will cease to exist
}
```

- Where does **int *** come from?
- How is **p** created?
- How is **p** destroyed?
- Does this code leak memory?

Automatic Variables: Custom Data



```
class pokemon {

};

int main() {
  pokemon pikachu; // pikachu is created

  // ... pikachu attacks a pidgey

  return 0; // pikachu will cease to exist
}
```

- Where does **pokemon** come from?
- How is **Pikachu** created?
- How is **Pikachu** destroyed?
- Will this code compile?

Comparing Creation and Destruction

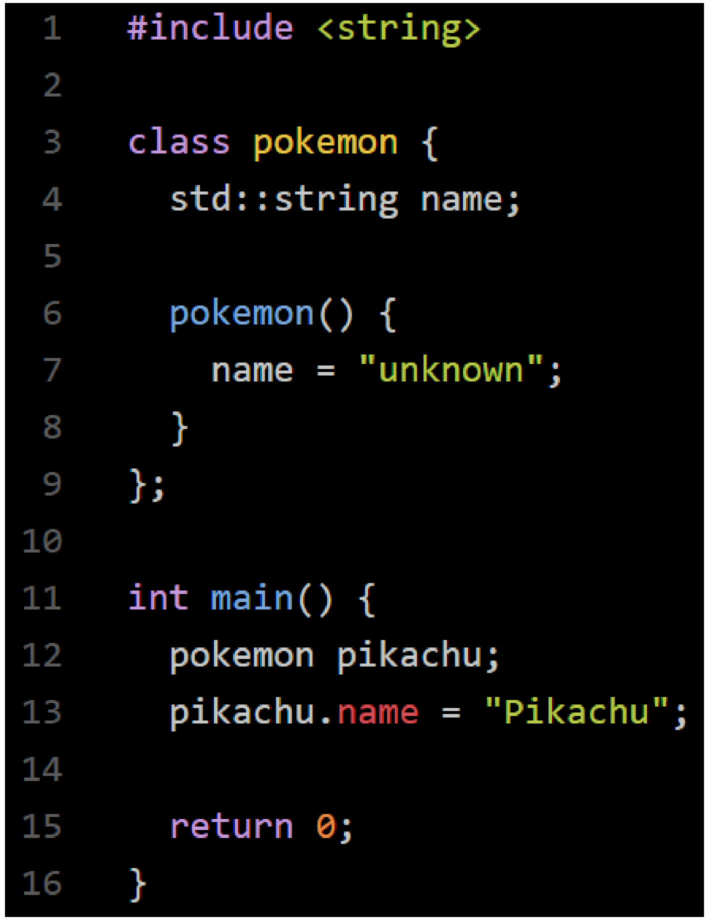
	How is it created?	**How is it destroyed?**	**Where does it come from?**
int x;	C++ Runtime	C++ Runtime	C++ Language
int * p = new int();	Trick Question • int * is created by the runtime • new int() is created **by the user**	Trick Question • int * is destroyed by the runtime • The int created by new *must* be deleted **by the user**	C++ Language
pokemon pikachu;	C++ Runtime calls the pokemon *constructor*	C++ Runtime calls the pokemon *destructor*	**User defined**

Comparing Creation and Destruction

	How is it created?	**How is it destroyed?**	**Where does it come from?**
int x;	C++ Runtime	C++ Runtime	C++ Language
int * p = new int();	Trick Question • int * is created by the runtime • new int() is created **by the user**	Trick Question • int * is destroyed by the runtime • The int created by new *must* be deleted **by the user**	C++ Language
pokemon pikachu;	C++ Runtime calls the pokemon *constructor*	C++ Runtime calls the pokemon *destructor*	**User defined**

Where does the Pokémon class's constructor and destructor come from?

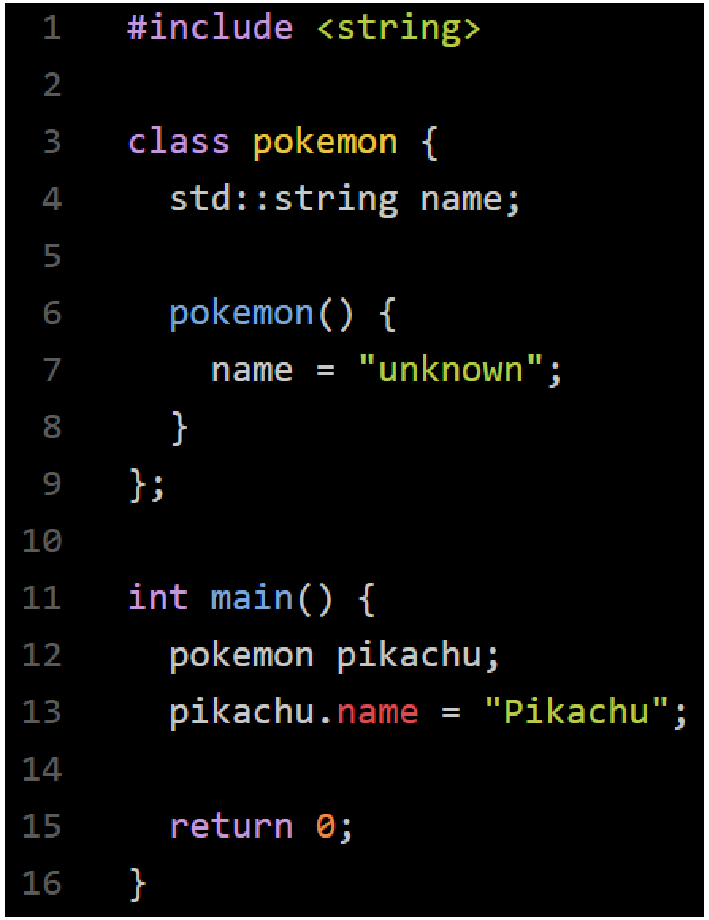
Customizing our Classes



```cpp
#include <string>

class pokemon {
  std::string name;

  pokemon() {
    name = "unknown";
  }
};

int main() {
  pokemon pikachu;
  pikachu.name = "Pikachu";

  return 0;
}
```

- Pokémon have a name
  - We can represent names with strings
- Should we have an “unknown” Pokémon?
- Will this code compile?

Customizing our Classes



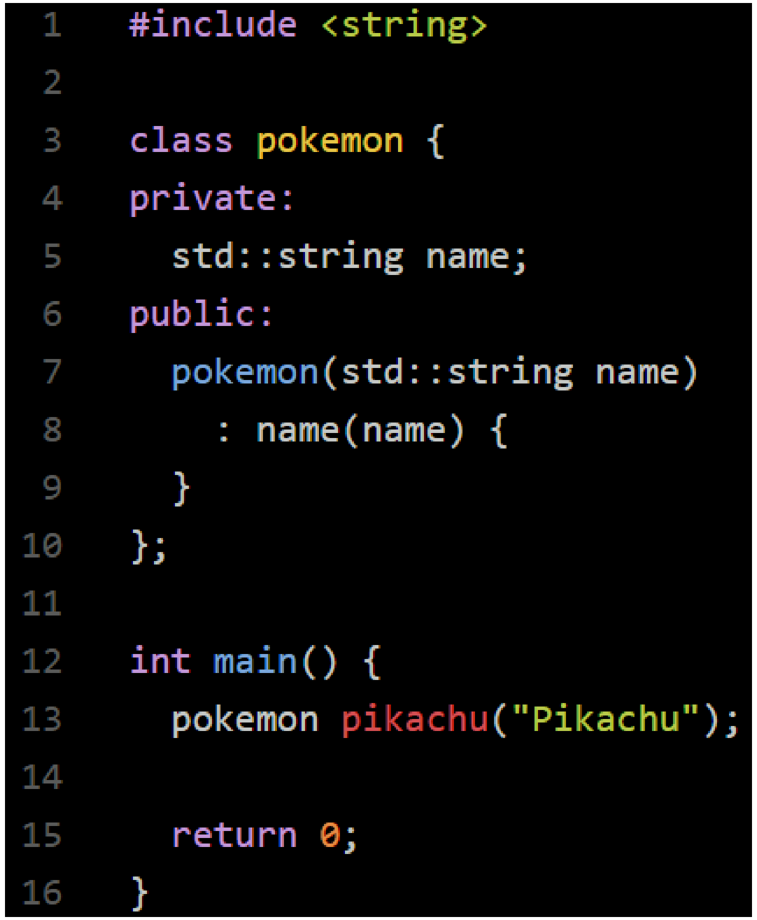
```
#include <string>

class pokemon {
  std::string name;

  pokemon() {
    name = "unknown";
  }
};

int main() {
  pokemon pikachu;
  pikachu.name = "Pikachu";

  return 0;
}
```

- Pokémon have a name
  - We can represent names with strings

- Should we have an “unknown” Pokémon?
  - It would be better not to have to handle an “unknown” case

- Will this code compile?
  - If *public/private* are not specified, a class is *private* by default
  - This means that **both** the name and the constructor are *private*
  - If the default constructor is *private*, the compiler will not provide a *public* one

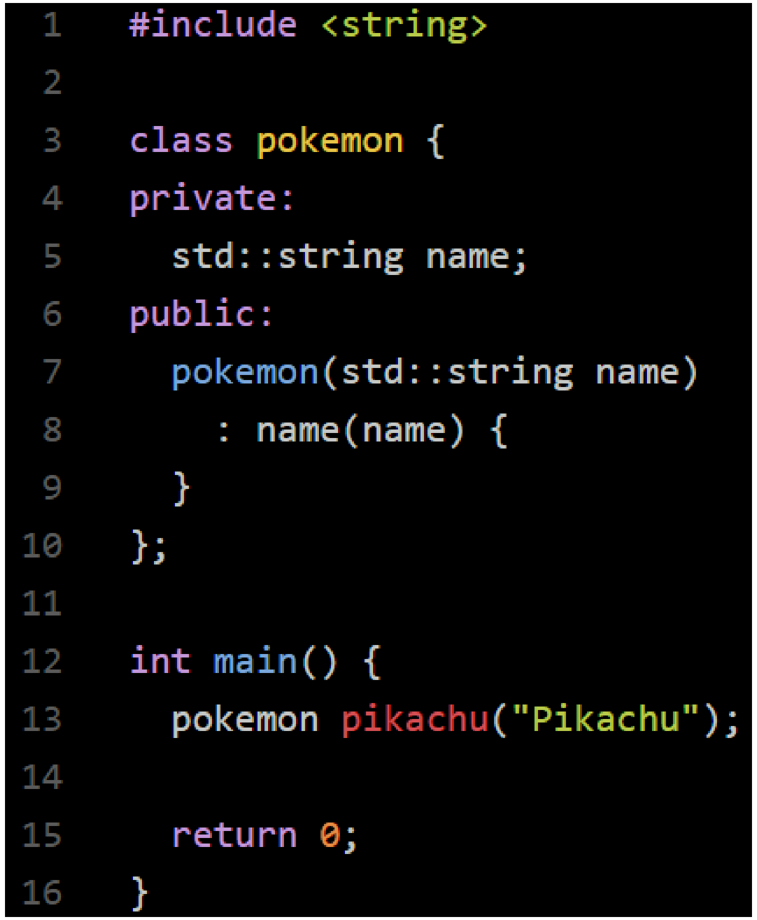
Properly Customizing our Class



```
#include <string>

class pokemon {
private:
  std::string name;
public:
  pokemon(std::string name)
    : name(name) {
  }
};

int main() {
  pokemon pikachu("Pikachu");

  return 0;
}
```

- Why should a Pokémon's **name** be private?
- How do we set the Pokémon's name?
- What is that weird *: name(name)* thing in the code?

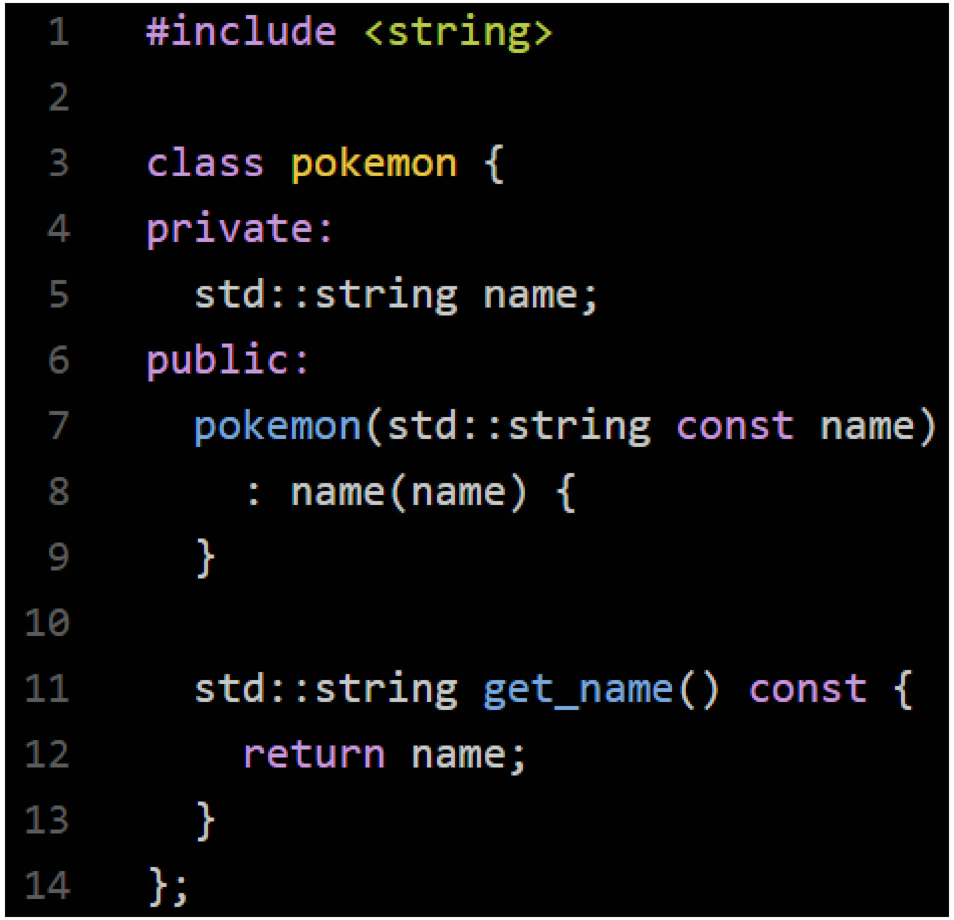
Properly Customizing our Class



```cpp
#include <string>

class pokemon {
private:
  std::string name;
public:
  pokemon(std::string name)
    : name(name) {
  }
};

int main() {
  pokemon pikachu("Pikachu");

  return 0;
}
```

- Why should a Pokémon's name be private?
  - Encapsulation: prevent other people from changing it
- How do we set the Pokémon's name?
  - By providing the name when we create the object
  - Now there is no “unknown” case for name – to create a Pokémon, a name **must** be provided
- What is that weird *: name(name)* thing in the code?
  - It’s called an **initialization list**
  - It is the preferred method to initializing member variables
  - **Sometimes** it is the only method to initialize a member variable

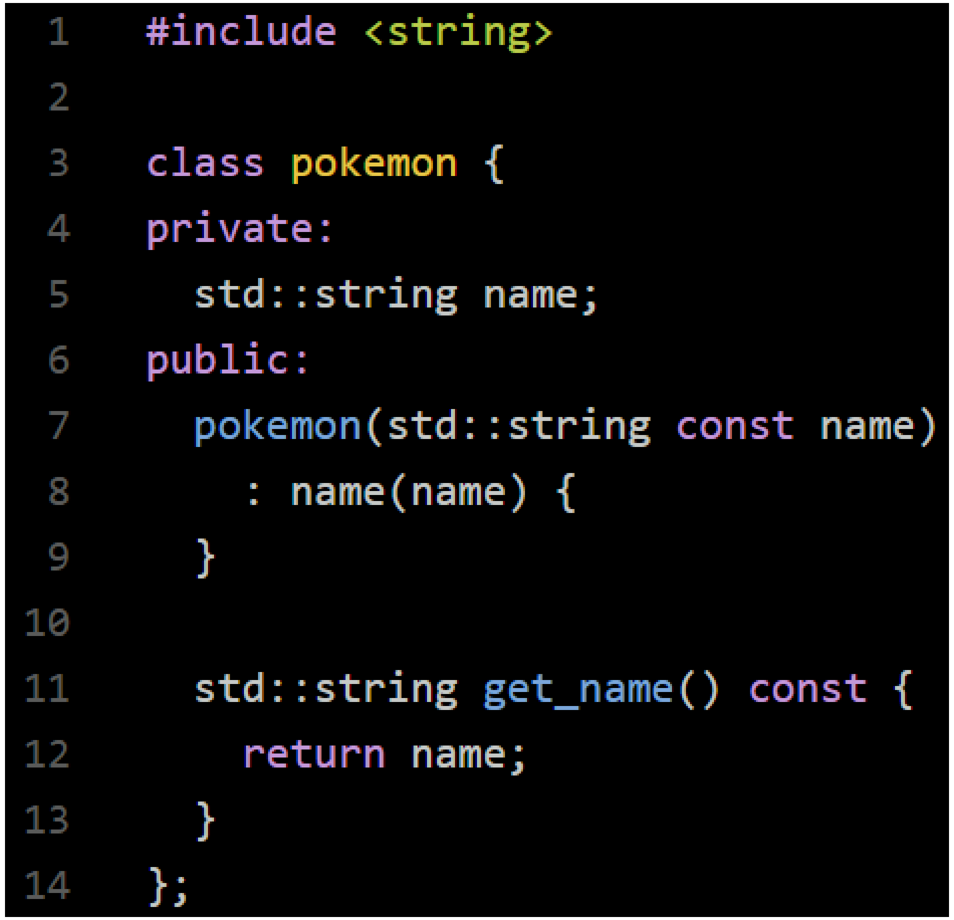
Accessing Private Member Variables



```
#include <string>

class pokemon {
private:
  std::string name;
public:
  pokemon(std::string const name)
    : name(name) {
  }

  std::string get_name() const {
    return name;
  }
};
```

- What does the **const** in the constructor parameters mean?
- What does the **const** after the **get_name()** function mean?

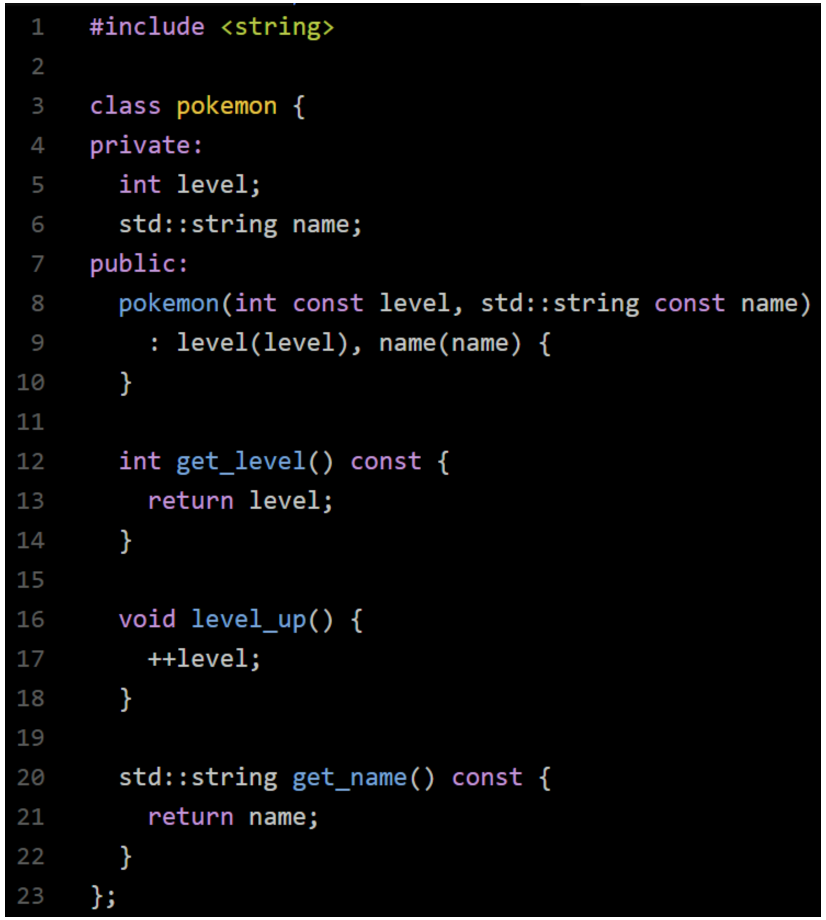
Accessing Private Member Variables



```
#include <string>

class pokemon {
private:
  std::string name;
public:
  pokemon(std::string const name)
    : name(name) {
  }

  std::string get_name() const {
    return name;
  }
};
```

- What does the **const** in the constructor parameters mean?
  - It is a promise that the name argument passed (e.g. “Pikachu”) will not be modified
  - If you do try to change it, your application will fail to compile
- What does the **const** after the get_name() function mean?
  - It means that this member function **will not change anything** in the object
- **Exercise**: update the class with a variable for the Pokémon's level

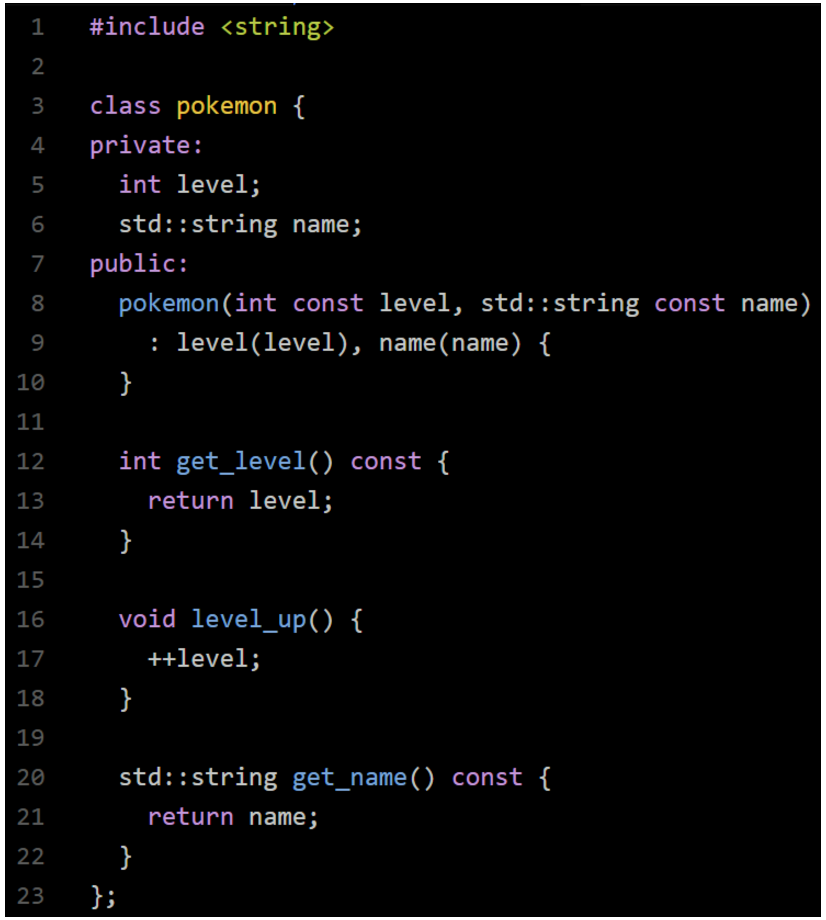
Adding Pokémon Levels



```cpp
#include <string>

class pokemon {
private:
  int level;
  std::string name;
public:
  pokemon(int const level, std::string const name)
    : level(level), name(name) {
  }

  int get_level() const {
    return level;
  }

  void level_up() {
    ++level;
  }

  std::string get_name() const {
    return name;
  }
};
```

- Why is **get_level()** marked **const**?
- Why is **level_up()** not marked **const**?
- Should level go before name in the initialization list?
- What should go in the destructor?

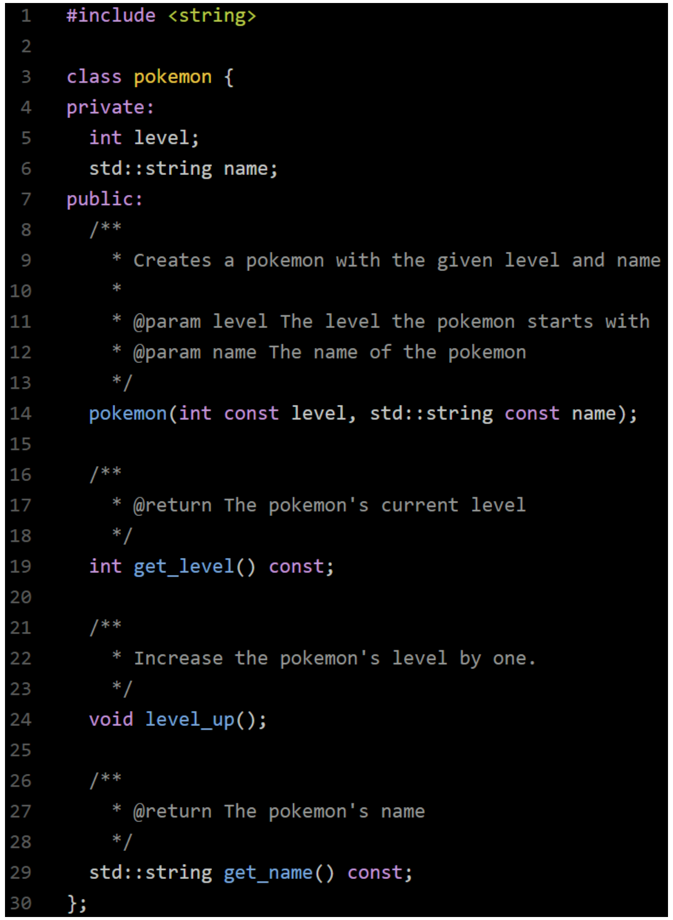
Adding Pokémon Levels



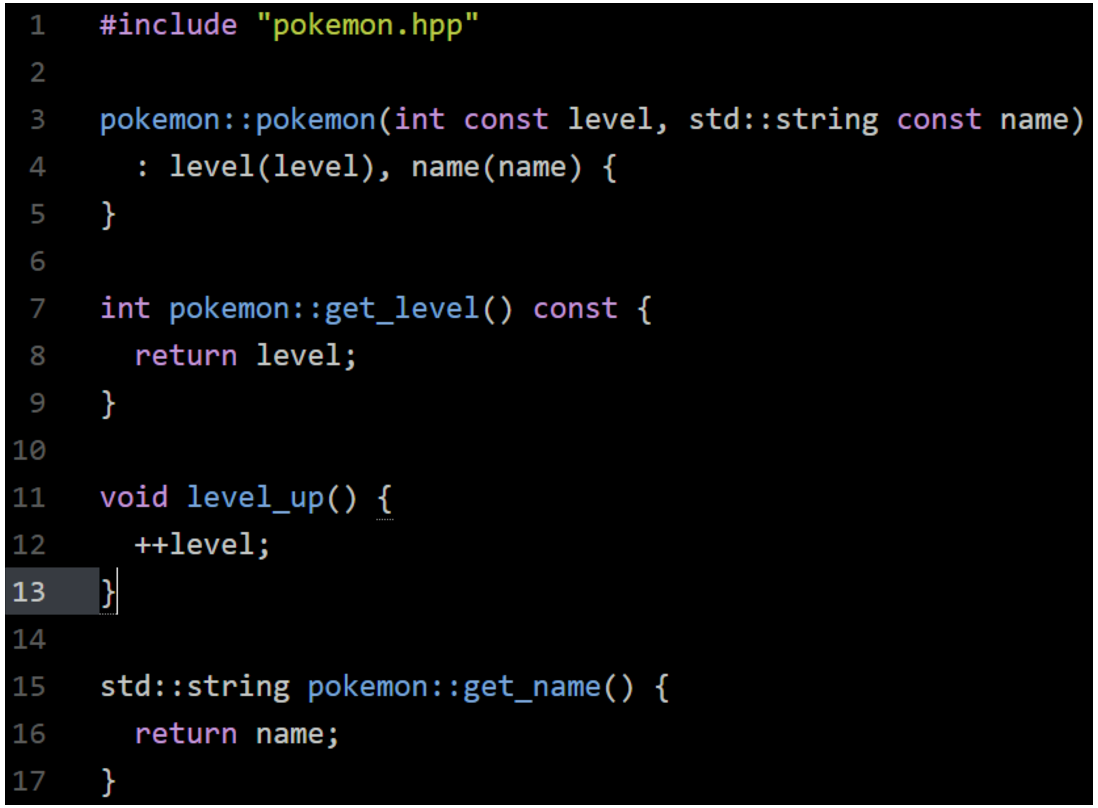
```cpp
#include <string>

class pokemon {
private:
  int level;
  std::string name;
public:
  pokemon(int const level, std::string const name)
    : level(level), name(name) {
  }

  int get_level() const {
    return level;
  }

  void level_up() {
    ++level;
  }

  std::string get_name() const {
    return name;
  }
};
```

- Why is **get_level()** marked **const**?
  - Because an accessor should not modify the Pokémon
- Why is **level_up()** not marked **const**?
  - Because we need to modify the level variable
- Should level go before name in the initialization list?
  - Yes, order matters
  - Initialization lists go in the order member variables are declared
- What should go in the destructor?
  - Nothing – we aren't managing any memory (e.g. pointers that use new)

Separating Specification from Implementation

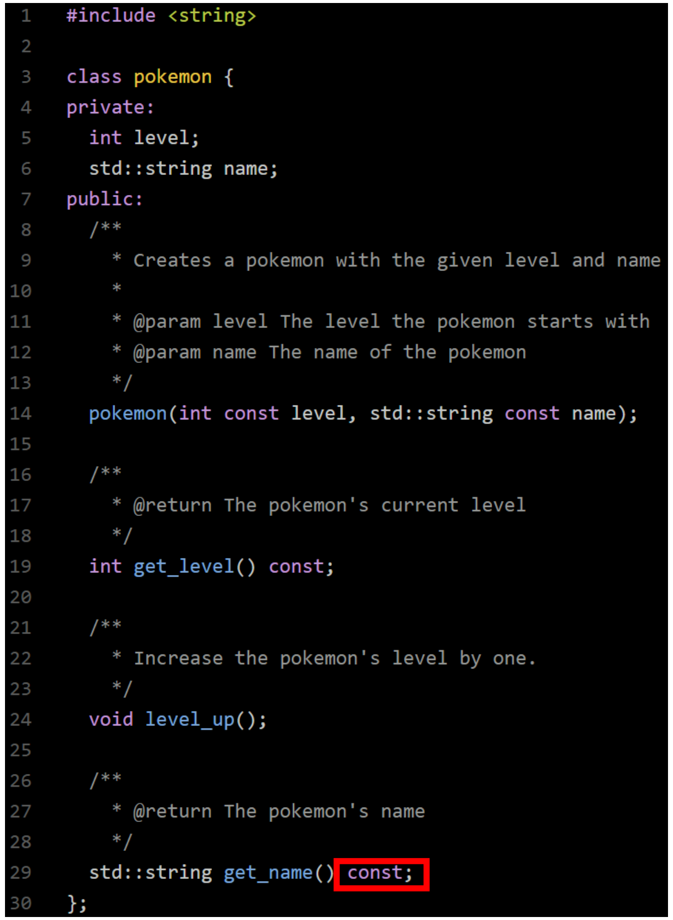


```cpp
#include <string>

class pokemon {
private:
  int level;
  std::string name;
public:
  /**
    * Creates a pokemon with the given level and name
    *
    * @param level The level the pokemon starts with
    * @param name The name of the pokemon
    */
  pokemon(int const level, std::string const name);

  /**
    * @return The pokemon's current level
    */
  int get_level() const;

  /**
    * Increase the pokemon's level by one.
    */
  void level_up();

  /**
    * @return The pokemon's name
    */
  std::string get_name() const;
};
```

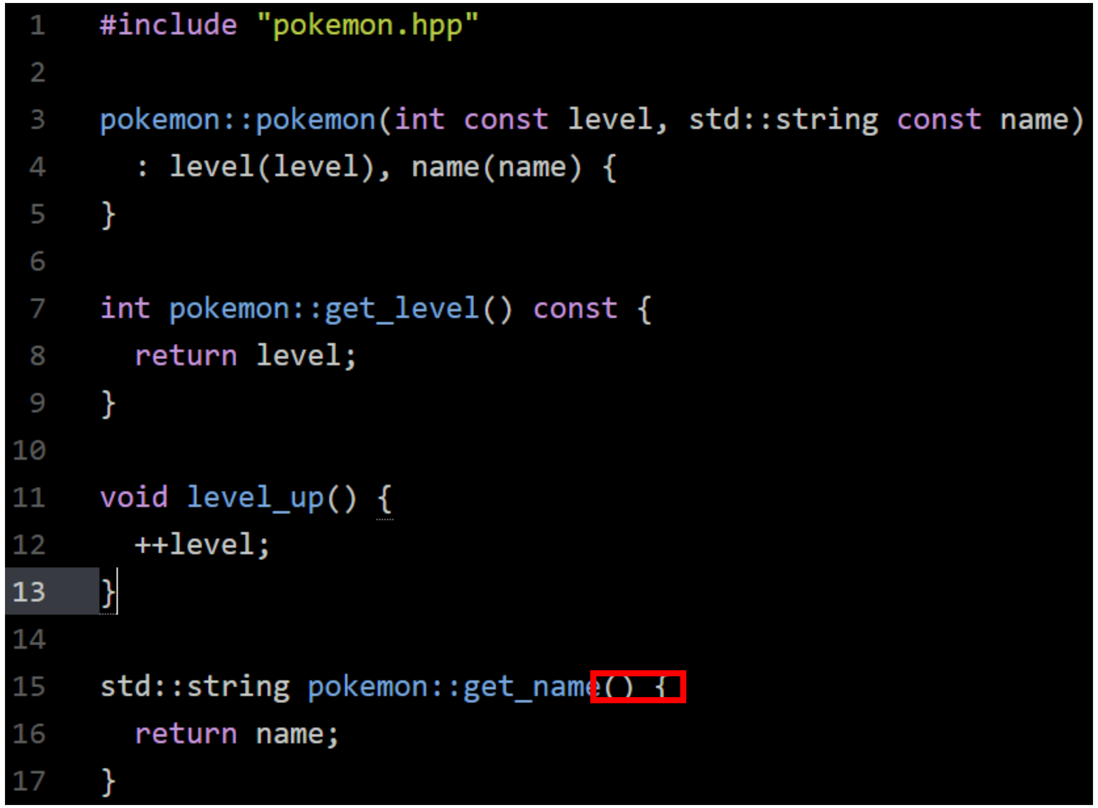


```cpp
#include "pokemon.hpp"

pokemon::pokemon(int const level, std::string const name)
  : level(level), name(name) {
}

int pokemon::get_level() const {
  return level;
}

void level_up() {
  ++level;
}

std::string pokemon::get_name() {
  return name;
}
```

Separating Specification from Implementation



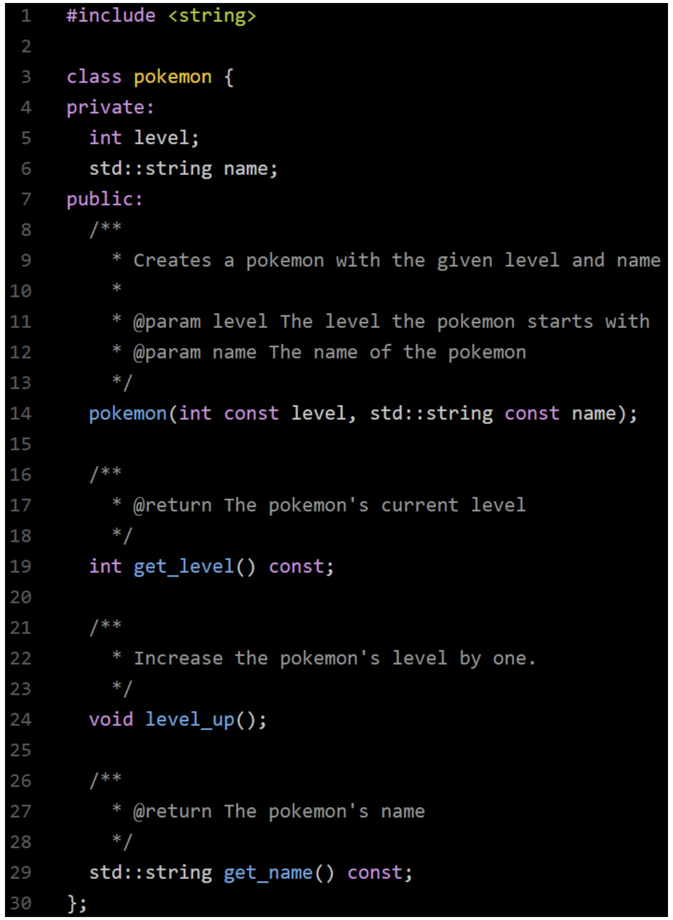
```
#include <string>

class pokemon {
private:
  int level;
  std::string name;
public:
  /**
    * Creates a pokemon with the given level and name
    *
    * @param level The level the pokemon starts with
    * @param name The name of the pokemon
    */
  pokemon(int const level, std::string const name);

  /**
    * @return The pokemon's current level
    */
  int get_level() const;

  /**
    * Increase the pokemon's level by one.
    */
  void level_up();

  /**
    * @return The pokemon's name
    */
  std::string get_name() const;
};
```



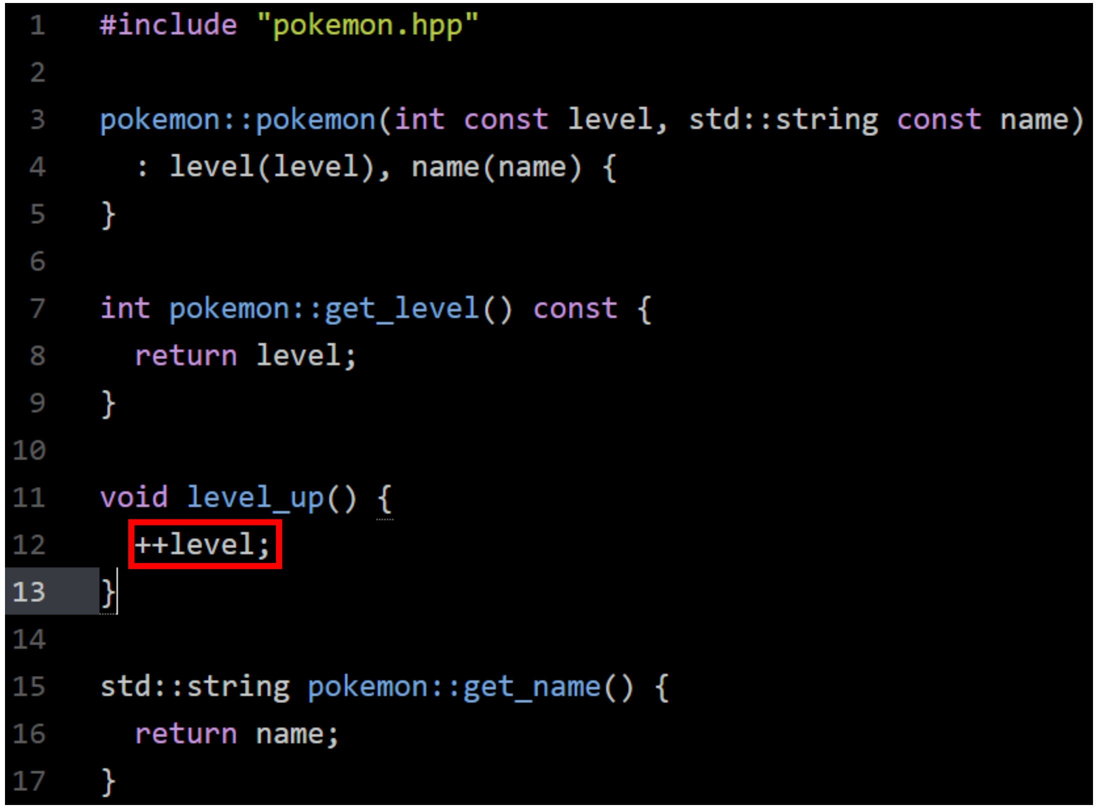
```
#include "pokemon.hpp"

pokemon::pokemon(int const level, std::string const name)
  : level(level), name(name) {
}

int pokemon::get_level() const {
  return level;
}

void level_up() {
  ++level;
}

std::string pokemon::get_name() {
  return name;
}
```

Compile Error #1: The prototype for get_name() does not match the definition!!!

Separating Specification from Implementation



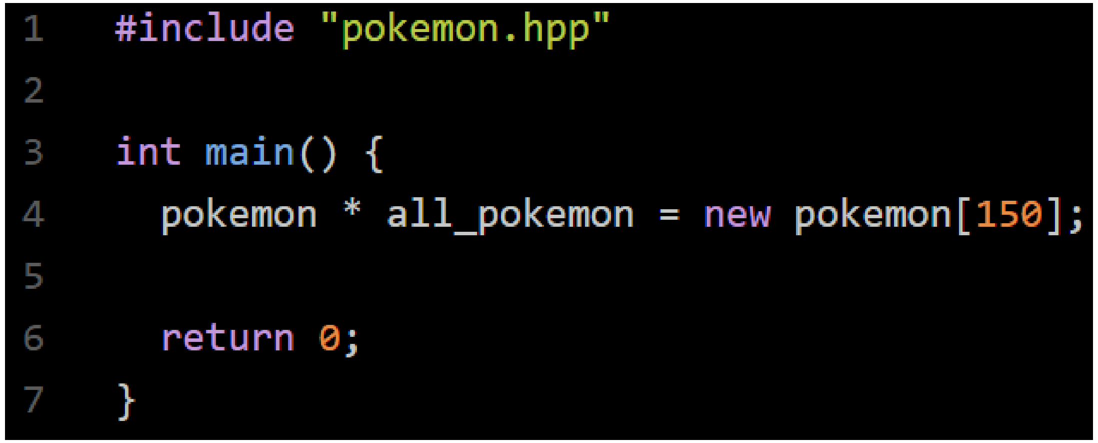
```
#include <string>

class pokemon {
private:
  int level;
  std::string name;
public:
  /**
    * Creates a pokemon with the given level and name
    *
    * @param level The level the pokemon starts with
    * @param name The name of the pokemon
    */
  pokemon(int const level, std::string const name);

  /**
    * @return The pokemon's current level
    */
  int get_level() const;

  /**
    * Increase the pokemon's level by one.
    */
  void level_up();

  /**
    * @return The pokemon's name
    */
  std::string get_name() const;
};
```



```
#include "pokemon.hpp"

pokemon::pokemon(int const level, std::string const name)
  : level(level), name(name) {
}

int pokemon::get_level() const {
  return level;
}

void level_up() {
  ++level;
}

std::string pokemon::get_name() {
  return name;
}
```

Compile Error #2: ‘level’ was not declared in this scope – the function must be prefixed with pokemon::

An Array of Pokémon



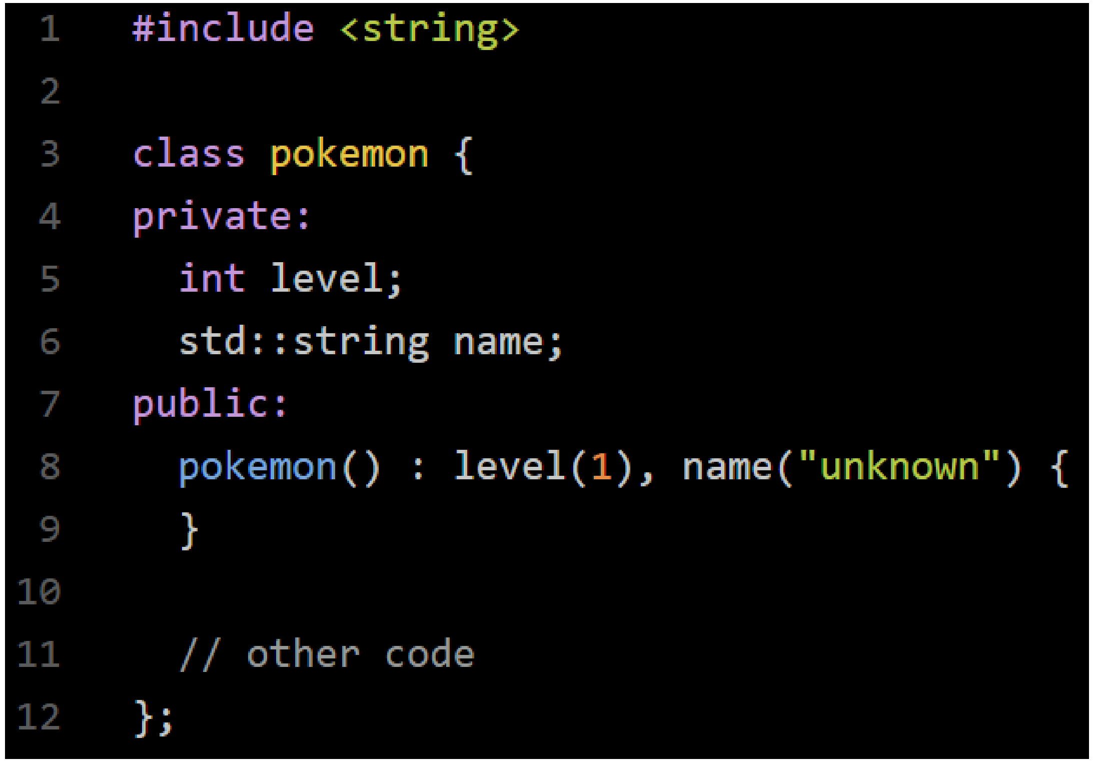
```
#include "pokemon.hpp"

int main() {
  pokemon * all_pokemon = new pokemon[150];

  return 0;
}
```

- This will not compile
  - **new** is looking for a constructor that takes no arguments, e.g. pokemon()
  - But our constructor takes two arguments, e.g. pokemon(int, std::string)

- How can we solve this problem?

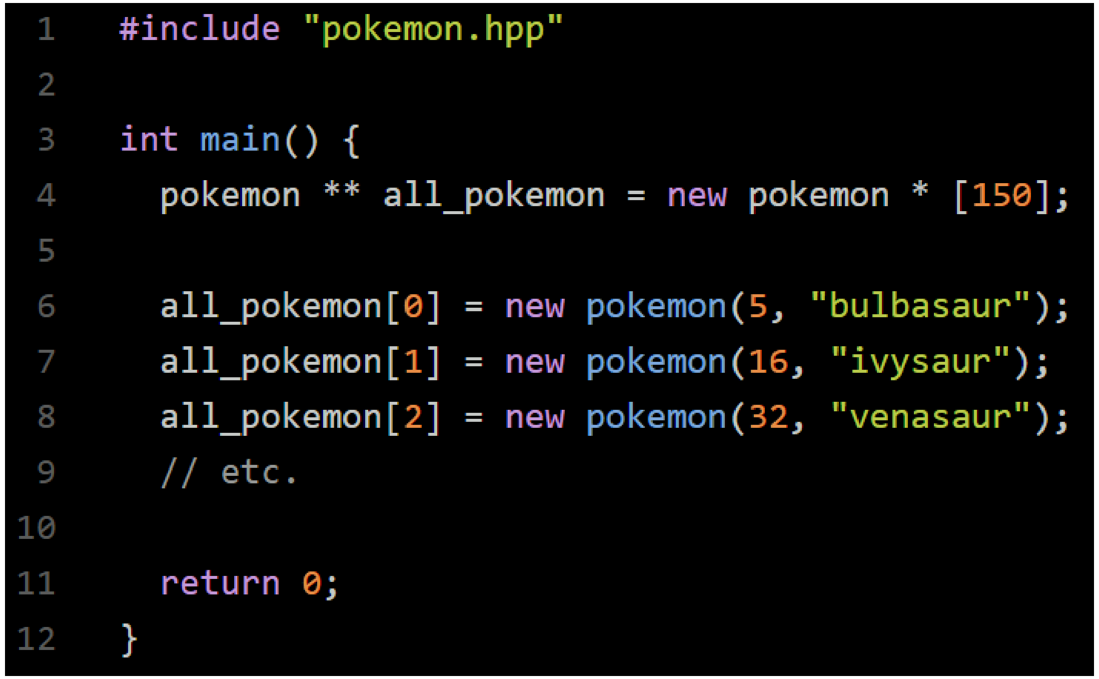
Solving the Pokémon Array Problem

Option #1 – Create another constructor



```cpp
#include <string>

class pokemon {
private:
  int level;
  std::string name;
public:
  pokemon() : level(1), name("unknown") {
  }

  // other code
};
```

Option #2 – Use a double pointer



```cpp
#include "pokemon.hpp"

int main() {
  pokemon ** all_pokemon = new pokemon * [150];

  all_pokemon[0] = new pokemon(5, "bulbasaur");
  all_pokemon[1] = new pokemon(16, "ivysaur");
  all_pokemon[2] = new pokemon(32, "venasaur");
  // etc.

  return 0;
}
```

Option #2 Actually Solves the Problem

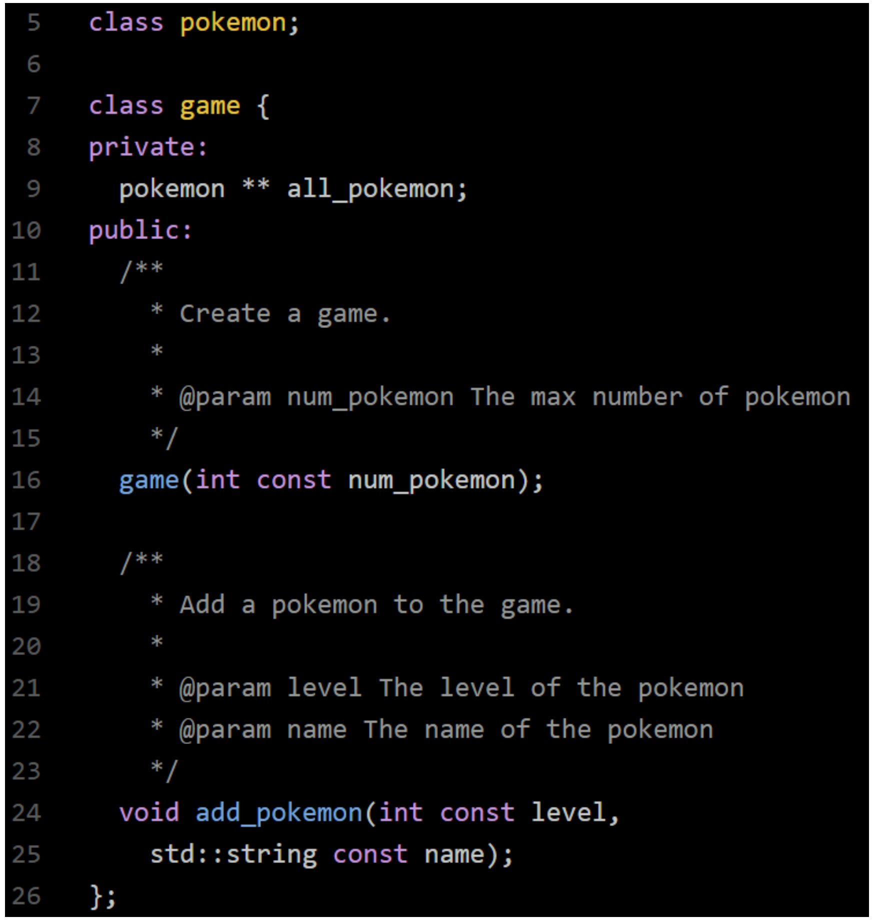
Option #1 – Create another constructor

- By adding a default constructor your code will compile…
- But now you need ways to change all the member variables!
  - Ruins encapsulation
  - Needless extra code

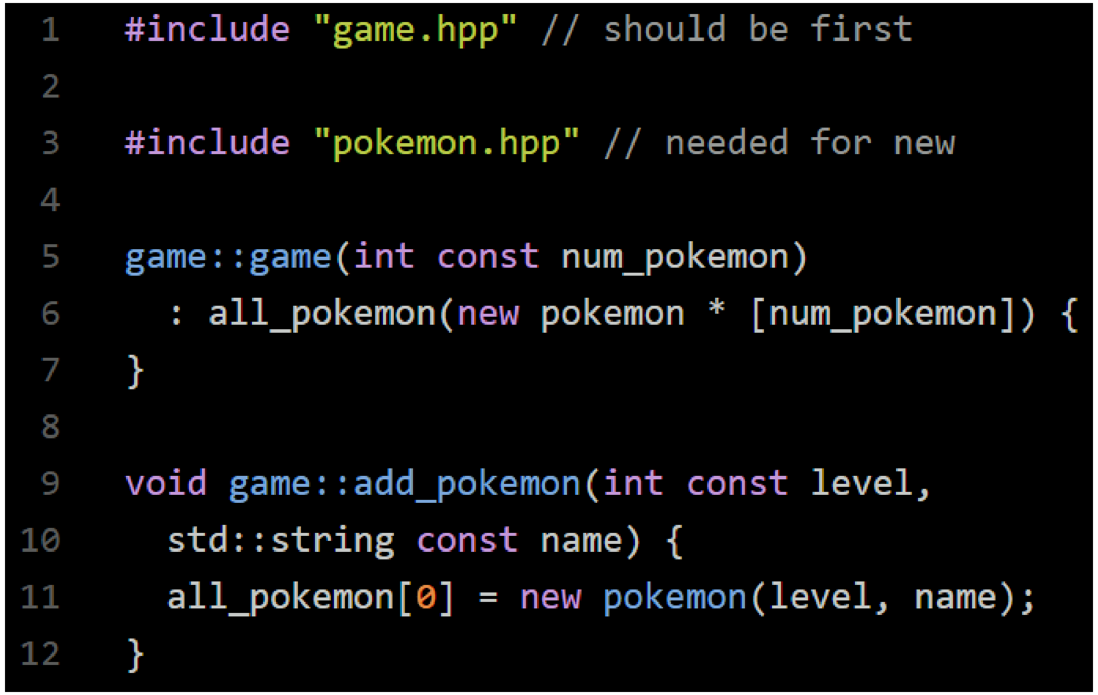
Option #2 – Use a double pointer

- Delays initialization of a Pokémon
- Allows you to use a constructor that takes arguments when you call **new** for each Pokémon

Creating a Pokémon Game

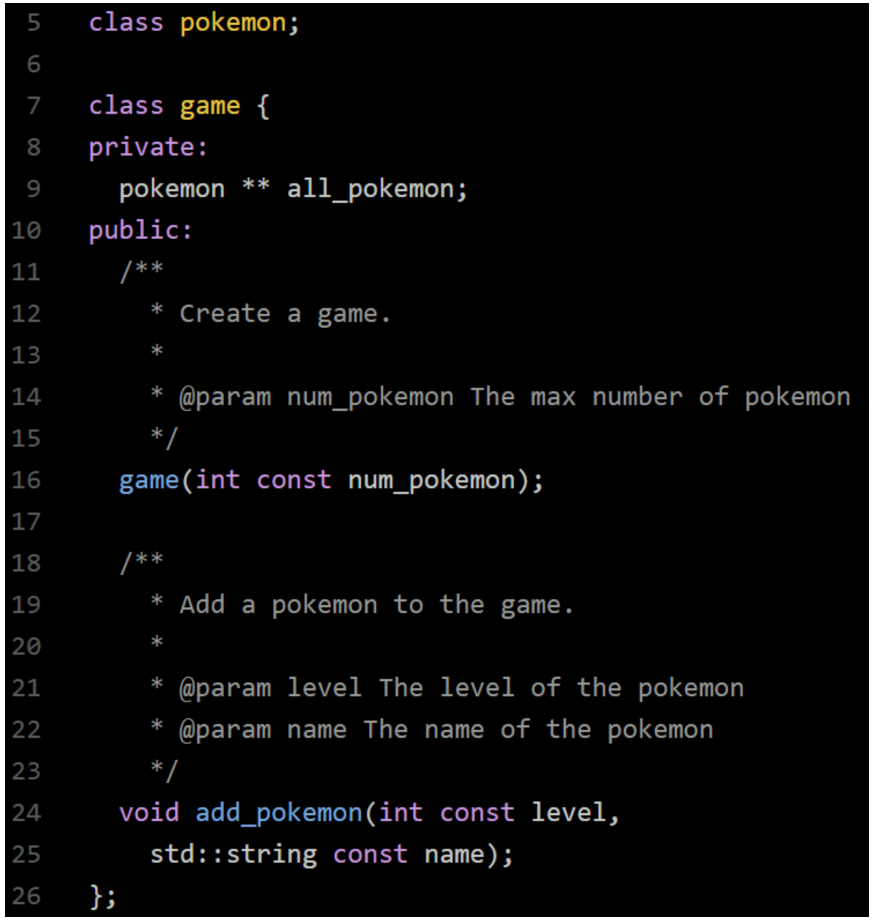


```cpp
class pokemon;

class game {
private:
  pokemon ** all_pokemon;
public:
  /**
    * Create a game.
    *
    * @param num_pokemon The max number of pokemon
    */
  game(int const num_pokemon);

  /**
    * Add a pokemon to the game.
    *
    * @param level The level of the pokemon
    * @param name The name of the pokemon
    */
  void add_pokemon(int const level,
    std::string const name);
};
```

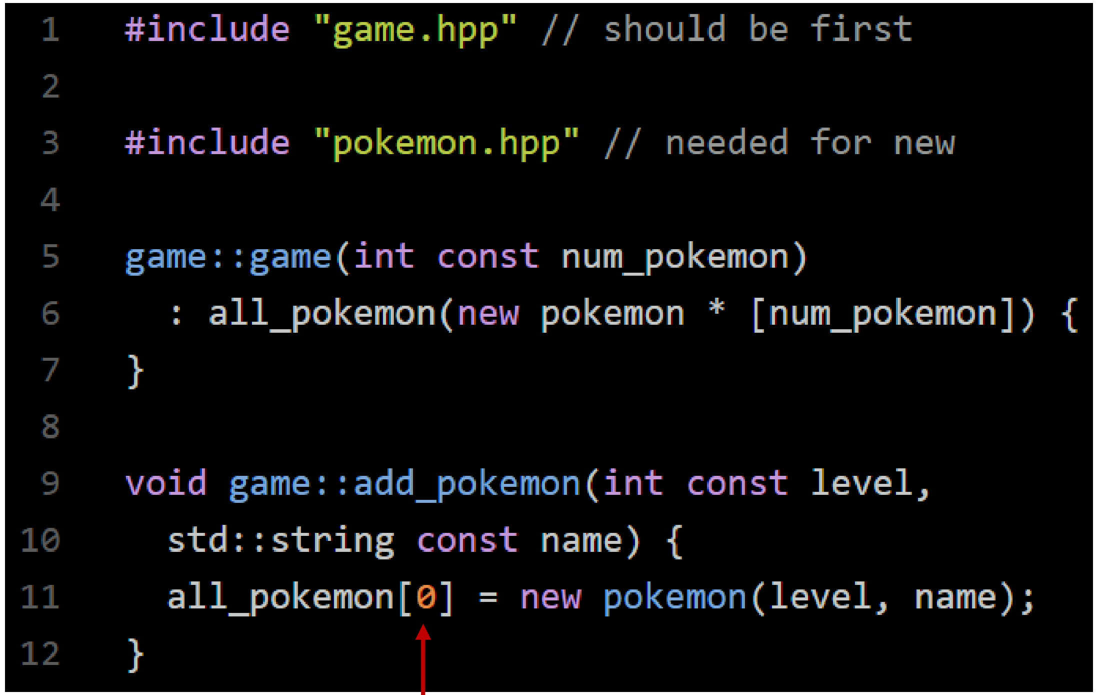


```cpp
#include "game.hpp" // should be first

#include "pokemon.hpp" // needed for new

game::game(int const num_pokemon)
  : all_pokemon(new pokemon * [num_pokemon]) {
}

void game::add_pokemon(int const level,
  std::string const name) {
  all_pokemon[0] = new pokemon(level, name);
}
```

Creating a Pokémon Game



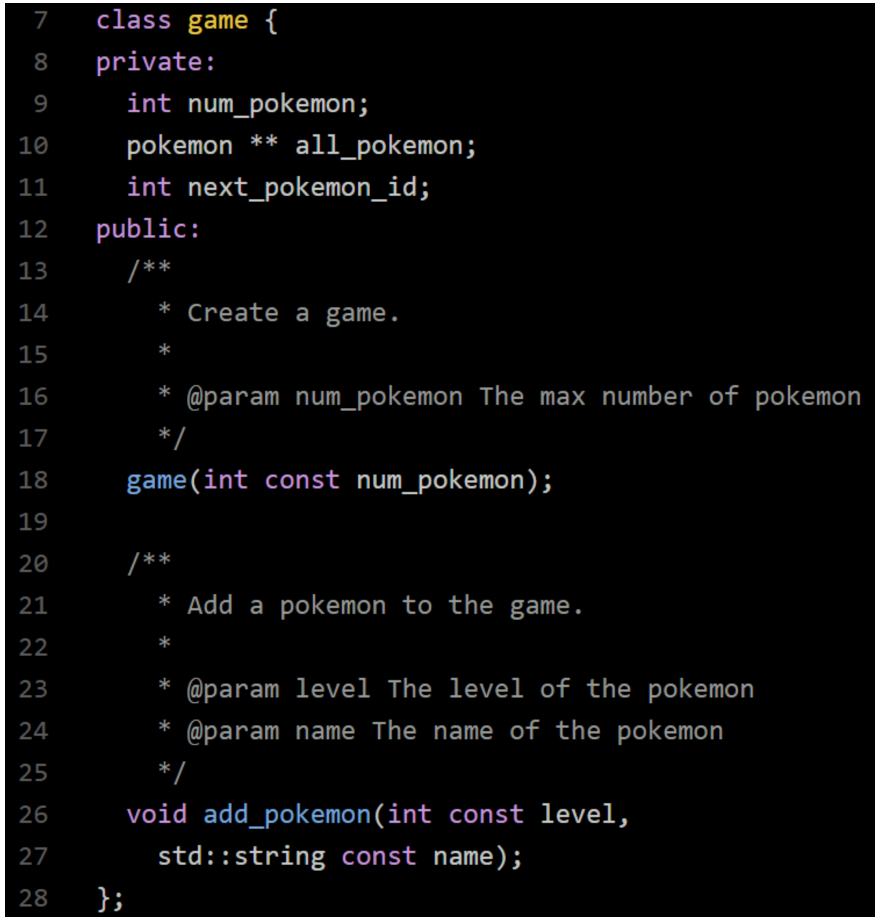
```cpp
class pokemon;

class game {
private:
  pokemon ** all_pokemon;
public:
  /**
    * Create a game.
    *
    * @param num_pokemon The max number of pokemon
    */
  game(int const num_pokemon);

  /**
    * Add a pokemon to the game.
    *
    * @param level The level of the pokemon
    * @param name The name of the pokemon
    */
  void add_pokemon(int const level,
    std::string const name);
};
```



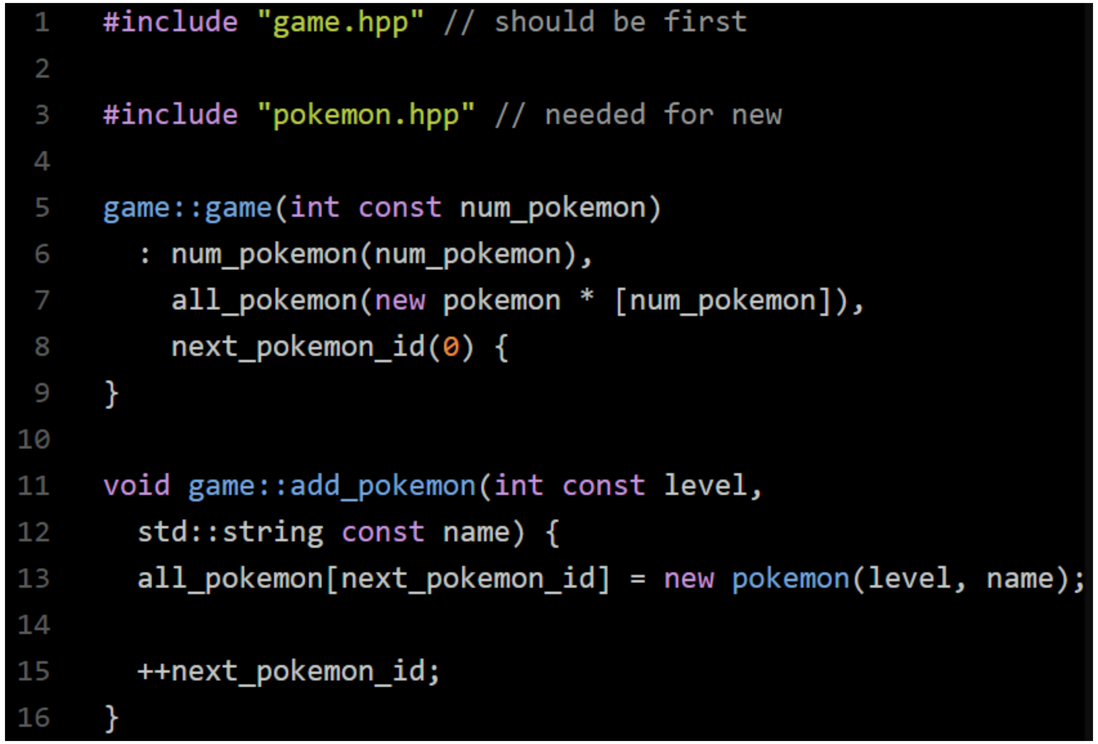
```cpp
#include "game.hpp" // should be first

#include "pokemon.hpp" // needed for new

game::game(int const num_pokemon)
  : all_pokemon(new pokemon * [num_pokemon]) {
}

void game::add_pokemon(int const level,
  std::string const name) {
  all_pokemon[0] = new pokemon(level, name);
}
```

Exercise: Fix it!

Solution



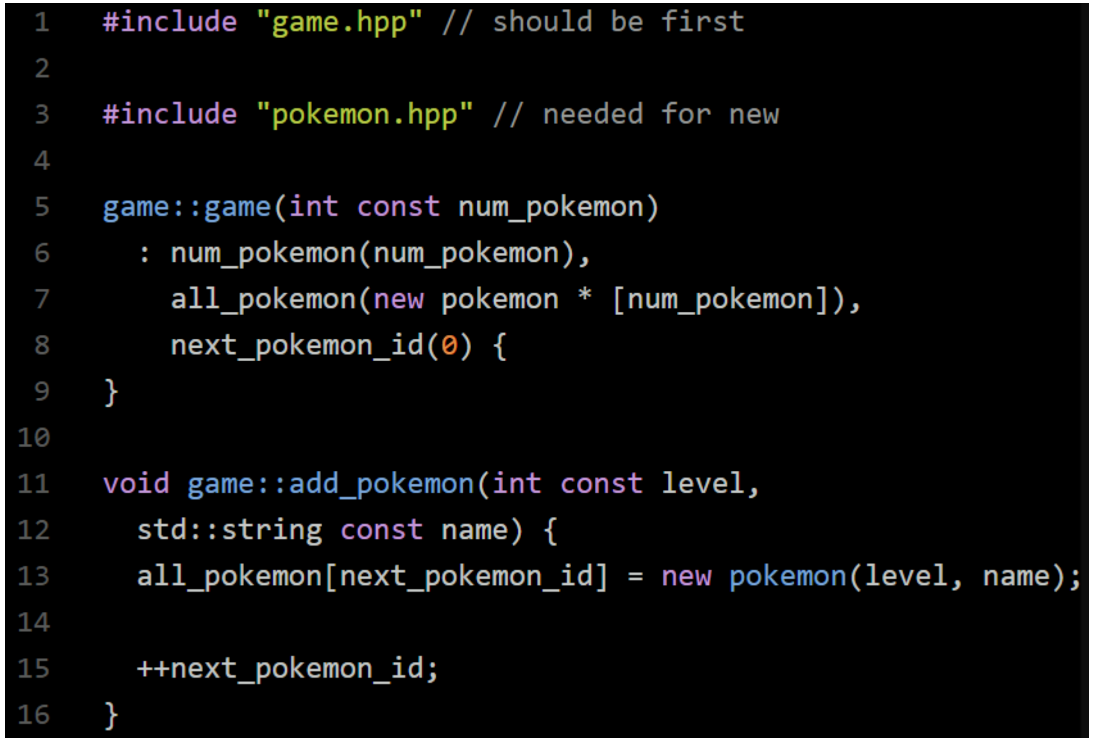
```cpp
class game {
private:
  int num_pokemon;
  pokemon ** all_pokemon;
  int next_pokemon_id;
public:
  /**
   * Create a game.
   *
   * @param num_pokemon The max number of pokemon
   */
  game(int const num_pokemon);

  /**
   * Add a pokemon to the game.
   *
   * @param level The level of the pokemon
   * @param name The name of the pokemon
   */
  void add_pokemon(int const level,
    std::string const name);
};
```



```cpp
#include "game.hpp" // should be first

#include "pokemon.hpp" // needed for new

game::game(int const num_pokemon)
  : num_pokemon(num_pokemon),
    all_pokemon(new pokemon * [num_pokemon]),
    next_pokemon_id(0) {
}

void game::add_pokemon(int const level,
  std::string const name) {
  all_pokemon[next_pokemon_id] = new pokemon(level, name);

  ++next_pokemon_id;
}
```

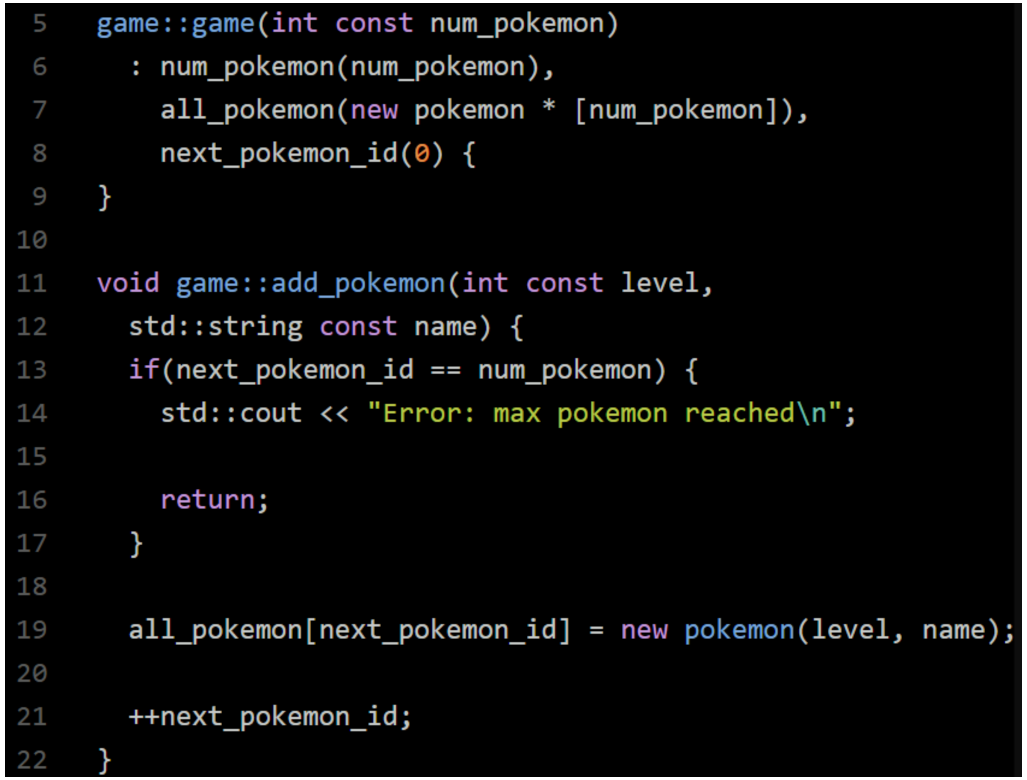
Actual Solution

Can Segfault



```
#include "game.hpp" // should be first

#include "pokemon.hpp" // needed for new

game::game(int const num_pokemon)
  : num_pokemon(num_pokemon),
    all_pokemon(new pokemon * [num_pokemon]),
    next_pokemon_id(0) {
}

void game::add_pokemon(int const level,
  std::string const name) {
  all_pokemon[next_pokemon_id] = new pokemon(level, name);

  ++next_pokemon_id;
}
```

Check the Array Bounds



```
game::game(int const num_pokemon)
  : num_pokemon(num_pokemon),
    all_pokemon(new pokemon * [num_pokemon]),
    next_pokemon_id(0) {
}

void game::add_pokemon(int const level,
  std::string const name) {
  if(next_pokemon_id == num_pokemon) {
    std::cout << "Error: max pokemon reached\n";

    return;
  }

  all_pokemon[next_pokemon_id] = new pokemon(level, name);

  ++next_pokemon_id;
}
```

What does an array need?

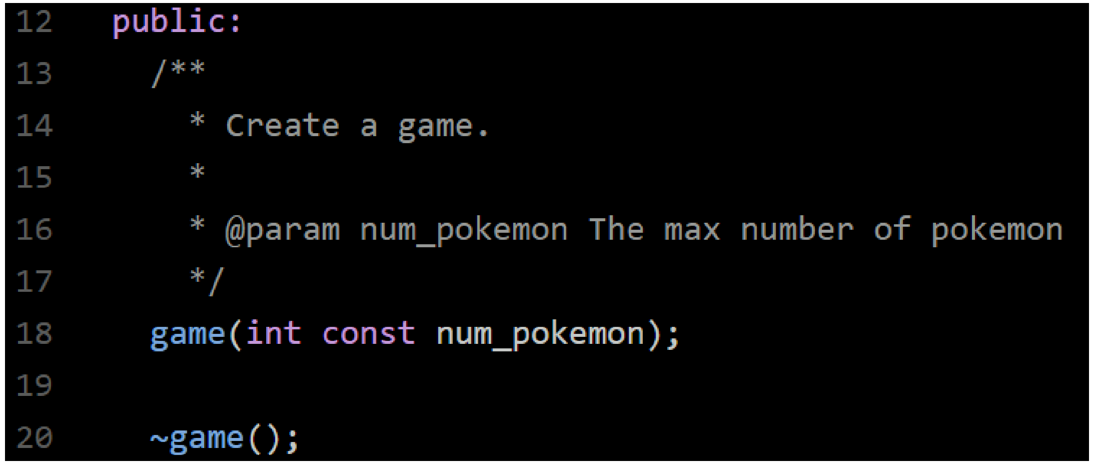
- A variable to track the maximum number of elements
  - E.g. num_pokemon

- A variable to track the index of an empty spot in the array
  - E.g. next_pokemon_id

- Bounds checking to make sure we don’t get a segmentation fault
  - E.g. when num_pokemon == next_pokemon_id

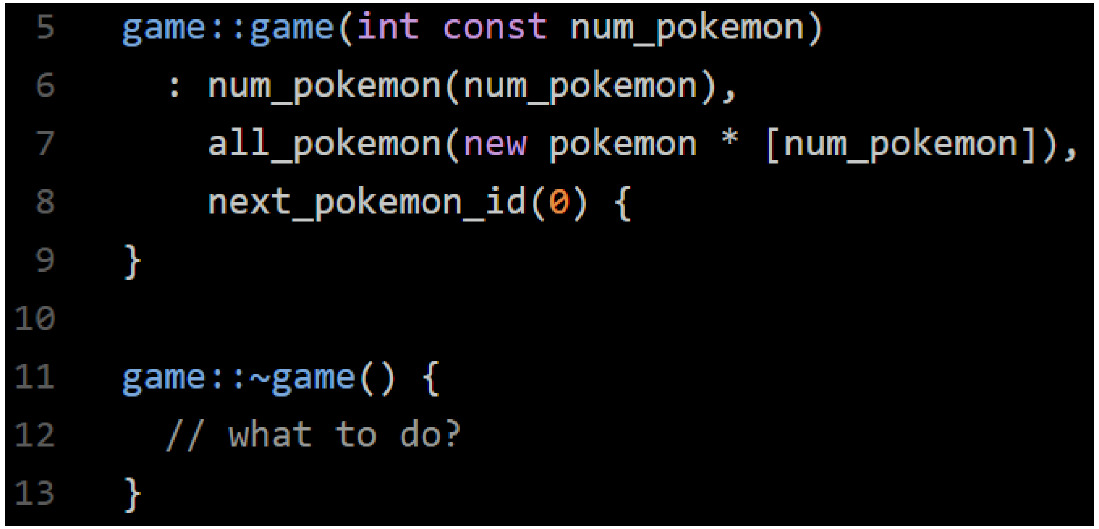
Exercise: Write the destructor for class game

Header



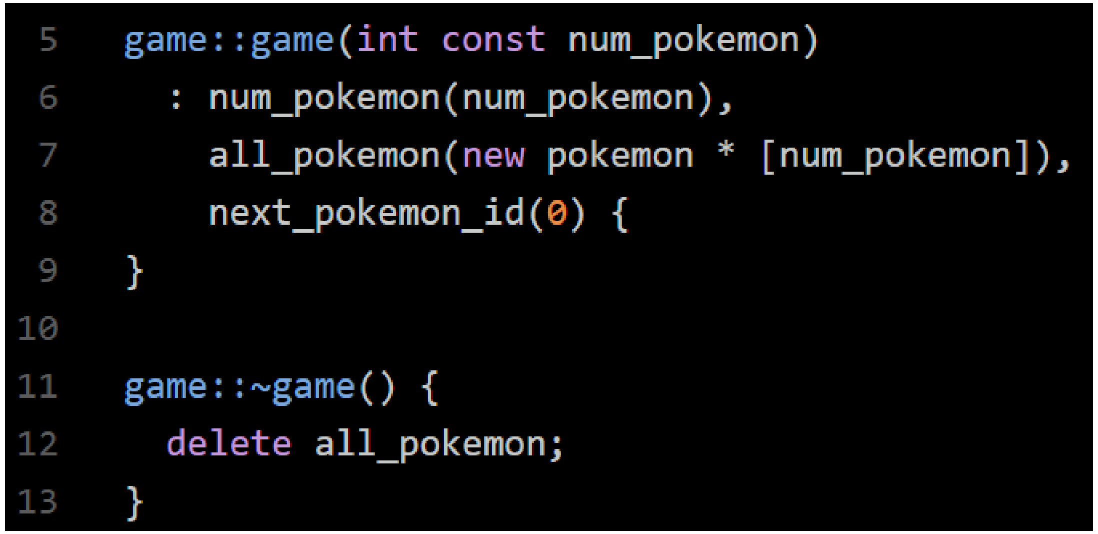
```
public:
  /**
   * Create a game.
   *
   * @param num_pokemon The max number of pokemon
   */
  game(int const num_pokemon);

  ~game();
```

Implementation



```
game::game(int const num_pokemon)
  : num_pokemon(num_pokemon),
    all_pokemon(new pokemon * [num_pokemon]),
    next_pokemon_id(0) {
}

game::~game() {
  // what to do?
}
```

Solution #1 – Delete the all_pokemon array



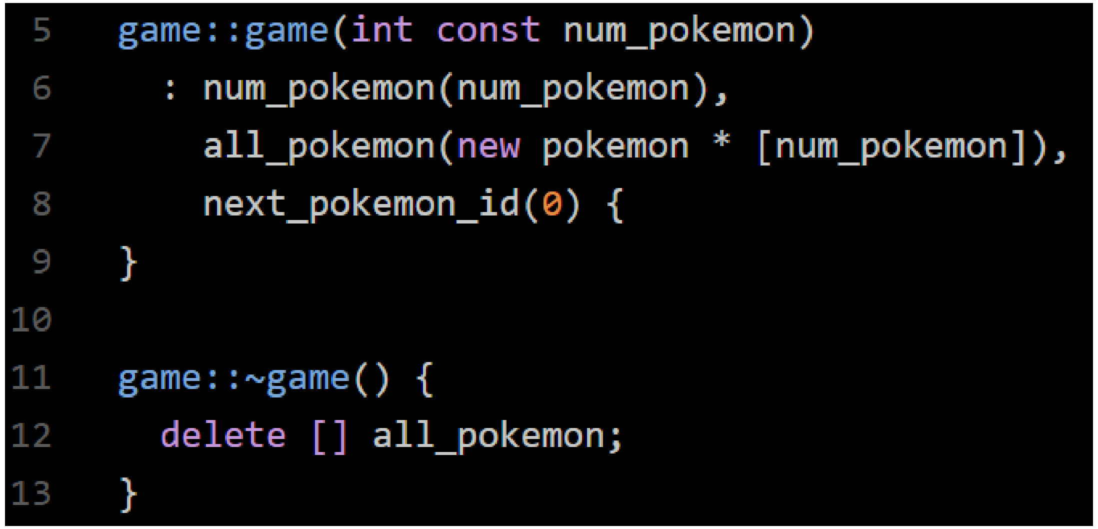
```
 5  game::game(int const num_pokemon)
 6    : num_pokemon(num_pokemon),
 7      all_pokemon(new pokemon * [num_pokemon]),
 8      next_pokemon_id(0) {
 9  }
10
11  game::~game() {
12    delete all_pokemon;
13  }
```

- This will cause a segmentation fault

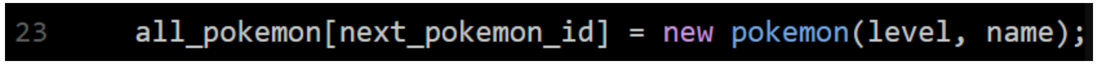
- There is a difference between **new** and **new[]**
  - **new** is not for arrays
  - **delete** is not for arrays
  - **new[]** is for arrays
  - **delete[]** is for arrays

- We are using **new[] on line 7**

Solution #2 – Actually delete it this time



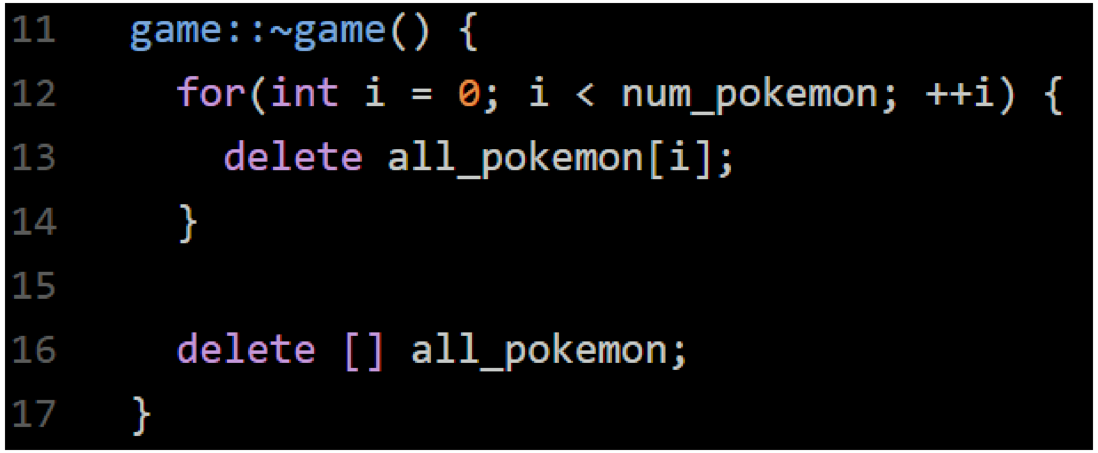
```cpp
game::game(int const num_pokemon)
  : num_pokemon(num_pokemon),
    all_pokemon(new pokemon * [num_pokemon]),
    next_pokemon_id(0) {
}

game::~game() {
  delete [] all_pokemon;
}
```



```cpp
    all_pokemon[next_pokemon_id] = new pokemon(level, name);
```

- No segmentation fault
  - Yay!
- But we may have a memory leak
  - Doh!
- If we added Pokémon the **new** operator was used
  - For every **new** there must be a **delete**

Solution #3 – Loop through all the Pokémon

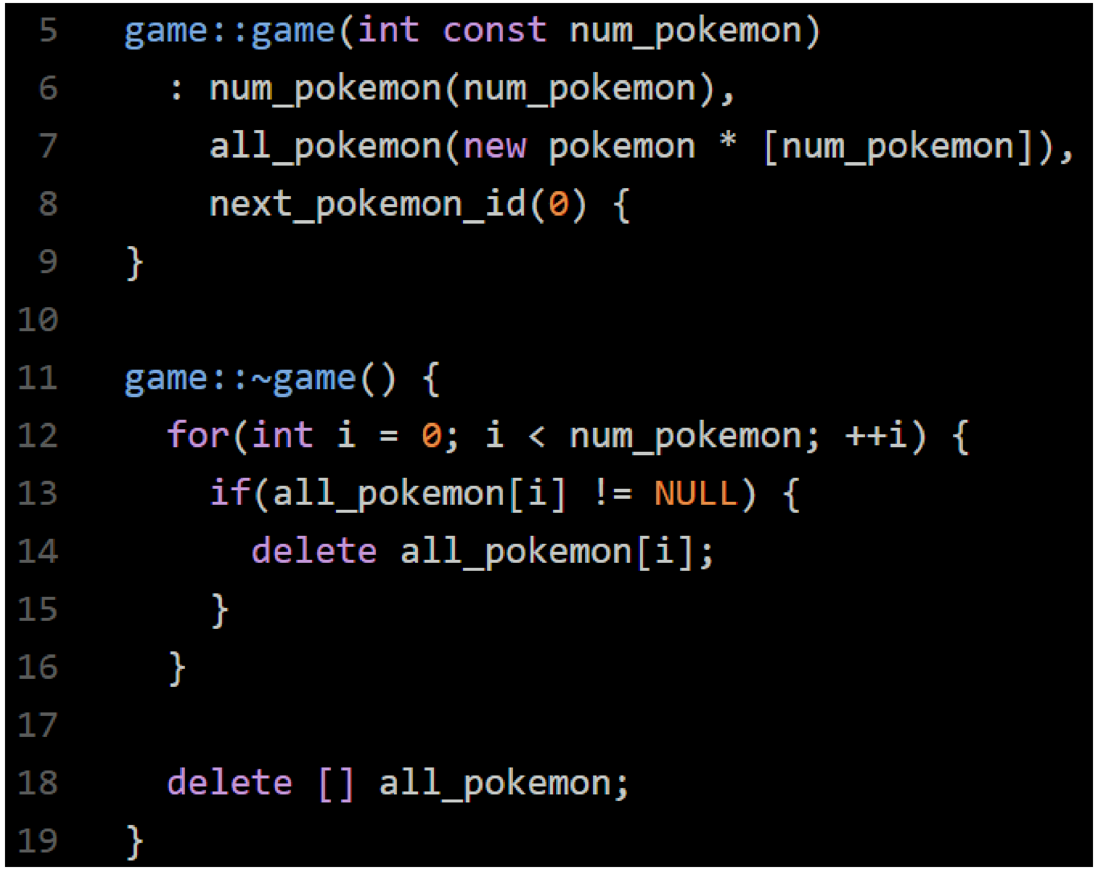


```
game::~game() {
  for(int i = 0; i < num_pokemon; ++i) {
    delete all_pokemon[i];
  }

  delete [] all_pokemon;
}
```

- This will cause a segmentation fault

- It loops until num_pokemon, but we may not have added that many Pokémon

Solution #4 – Make sure the pointer exists

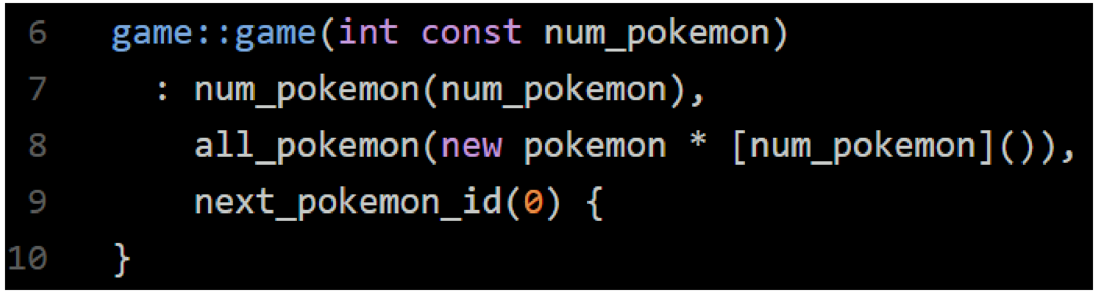


```
game::game(int const num_pokemon)
  : num_pokemon(num_pokemon),
    all_pokemon(new pokemon * [num_pokemon]),
    next_pokemon_id(0) {
}

game::~game() {
  for(int i = 0; i < num_pokemon; ++i) {
    if(all_pokemon[i] != NULL) {
      delete all_pokemon[i];
    }
  }

  delete [] all_pokemon;
}
```

- This **might** cause a segmentation fault

- What is the value of all our pointers in the all_pokemon array initialized to?
  - It’s not necessarily **NULL**!

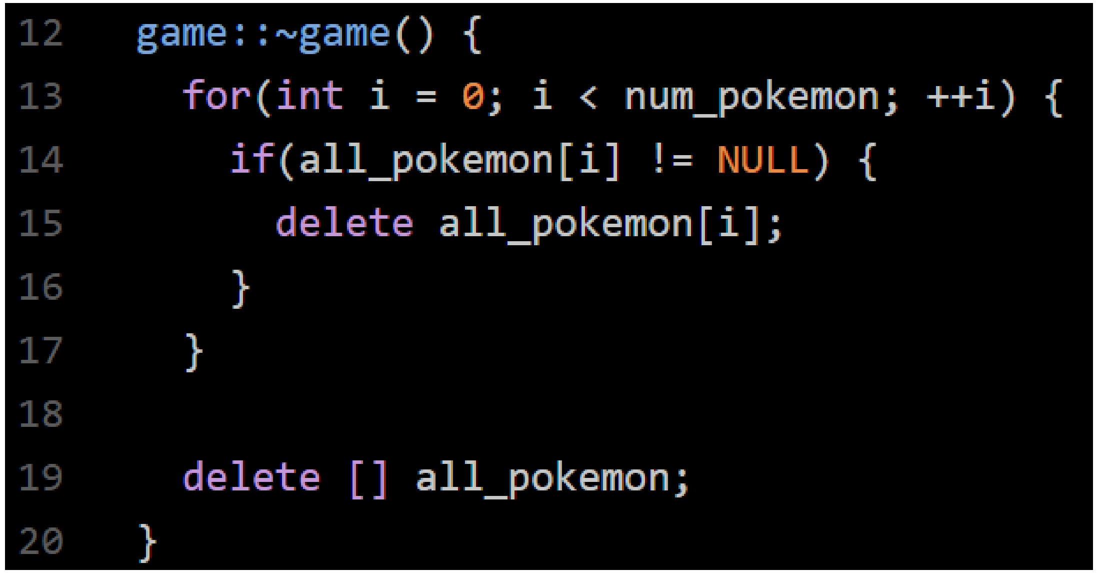
Solution #5 – Make sure pointers are NULL



```
game::game(int const num_pokemon)
  : num_pokemon(num_pokemon),
    all_pokemon(new pokemon * [num_pokemon]()),
    next_pokemon_id(0) {
}
```

- Notice the brackets after [num_pokemon]()
    - This will initialize all your pointers to zero (which is the same as **NULL**)
    - It **must** be empty brackets
- No segmentation fault
    - Yay
- Do we really need to loop for num_pokemon?

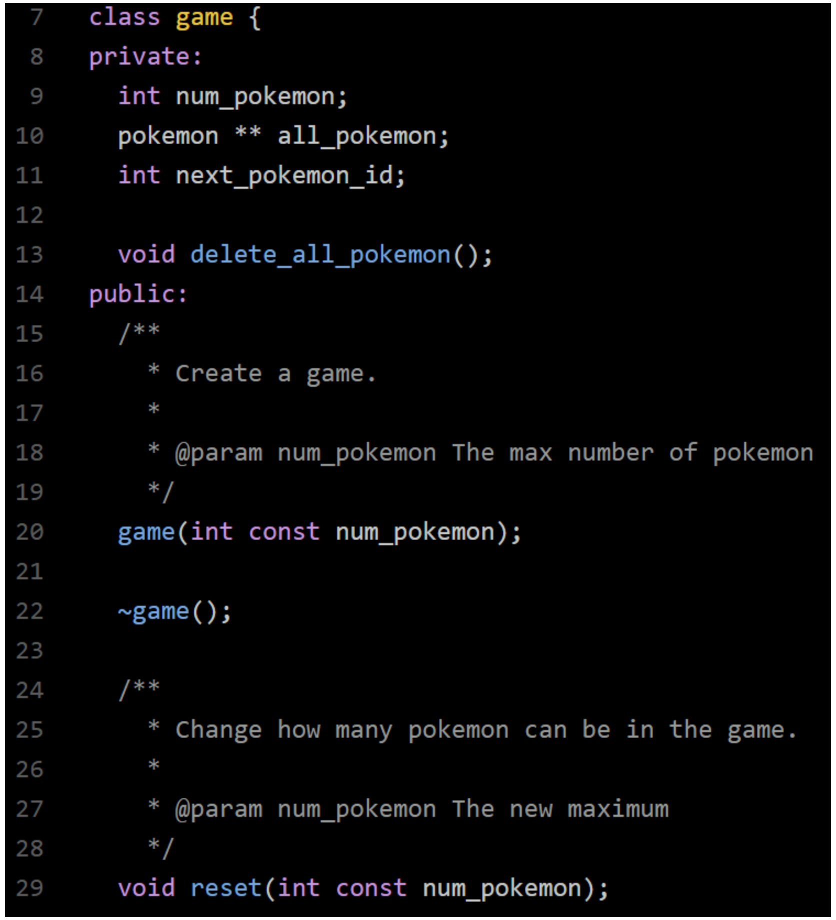
Solution #6 – Loop through some Pokémon



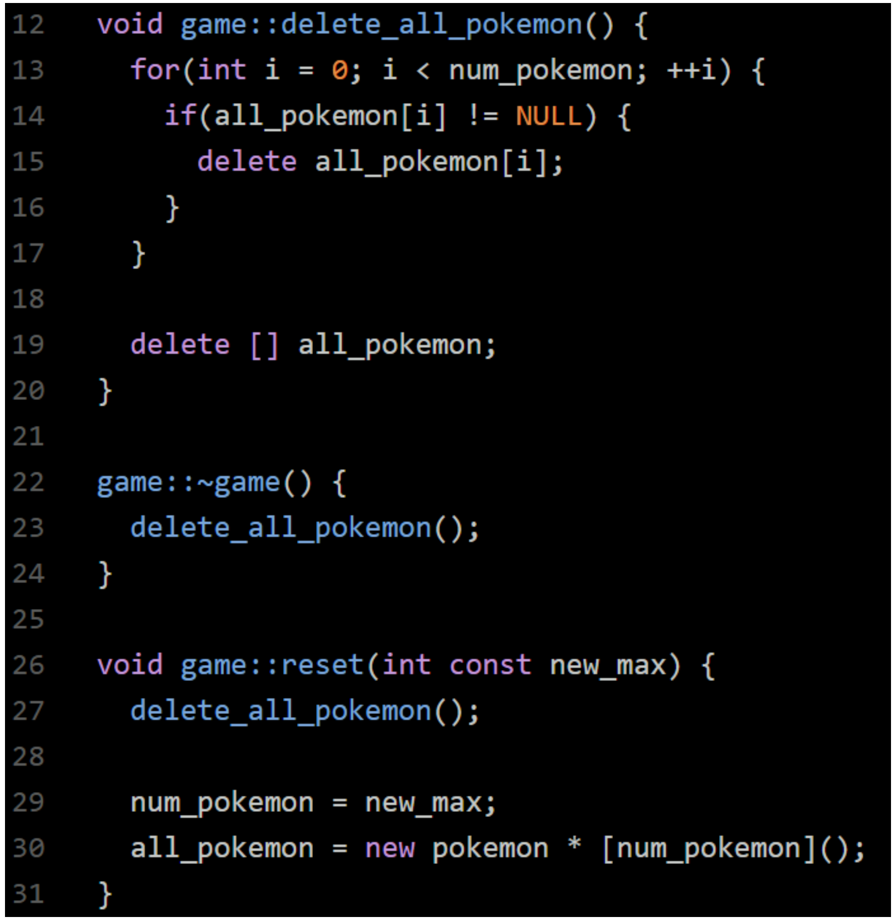
```
game::~game() {
  for(int i = 0; i < num_pokemon; ++i) {
    if(all_pokemon[i] != NULL) {
      delete all_pokemon[i];
    }
  }

  delete [] all_pokemon;
}
```

- Notice the change in the for loop
- Do we still need to check for **NULL**?
  - Probably not, but we can
  - Doing so is “defensive” programming

Changing the Number of Pokémon



```
class game {
private:
  int num_pokemon;
  pokemon ** all_pokemon;
  int next_pokemon_id;

  void delete_all_pokemon();
public:
  /**
    * Create a game.
    *
    * @param num_pokemon The max number of pokemon
    */
  game(int const num_pokemon);

  ~game();

  /**
    * Change how many pokemon can be in the game.
    *
    * @param num_pokemon The new maximum
    */
  void reset(int const num_pokemon);
```



```
void game::delete_all_pokemon() {
  for(int i = 0; i < num_pokemon; ++i) {
    if(all_pokemon[i] != NULL) {
      delete all_pokemon[i];
    }
  }

  delete [] all_pokemon;
}

game::~game() {
  delete_all_pokemon();
}

void game::reset(int const new_max) {
  delete_all_pokemon();

  num_pokemon = new_max;
  all_pokemon = new pokemon * [num_pokemon]();
}
```