C to C++ Transition

CSC 100 Intro to Programming in C++ (Spring 2025)

Marcus Birkenkrahe (pledged)

May 3, 2025

Contents

1 Objectives

Object-Oriented Programming (OOP) with C++:

- ☐ Transform C `struct` into C++ `class`
- ☐ Understand that data can be `private` or `public`
- ☐ Access `private` `Player` data with `getHealth`
- ☐ Create a `Player` class with `takeDamage` and `heal`
- ☐ Define `class` constructors and destructors
- ☐ Apply the `this` pointer inside `class` methods
- ☐ Create `static` members to track state across instances
- ☐ Understand the object lifecycle

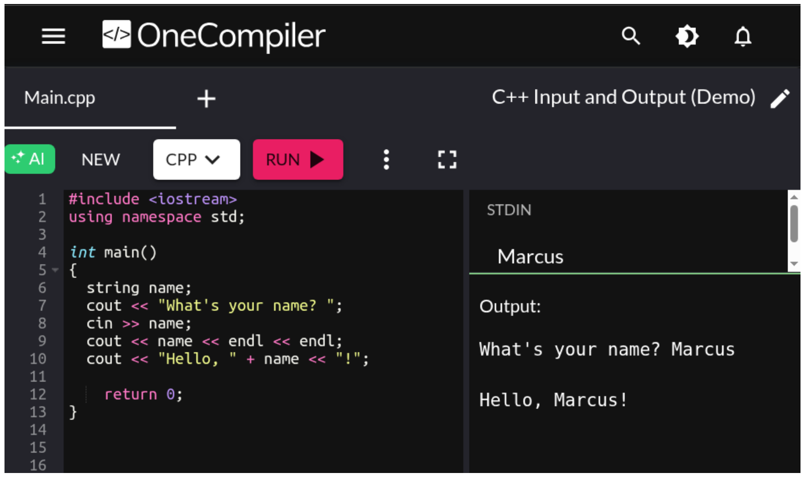
2 Codealong with C++ in OneCompiler

- OneCompiler is an IDE for multiple languages, and C/C++ are distinguished.
- Open `onecompiler.com/cpp` - this will bring up the template program for C++ rather than C - let's go through it line by line:

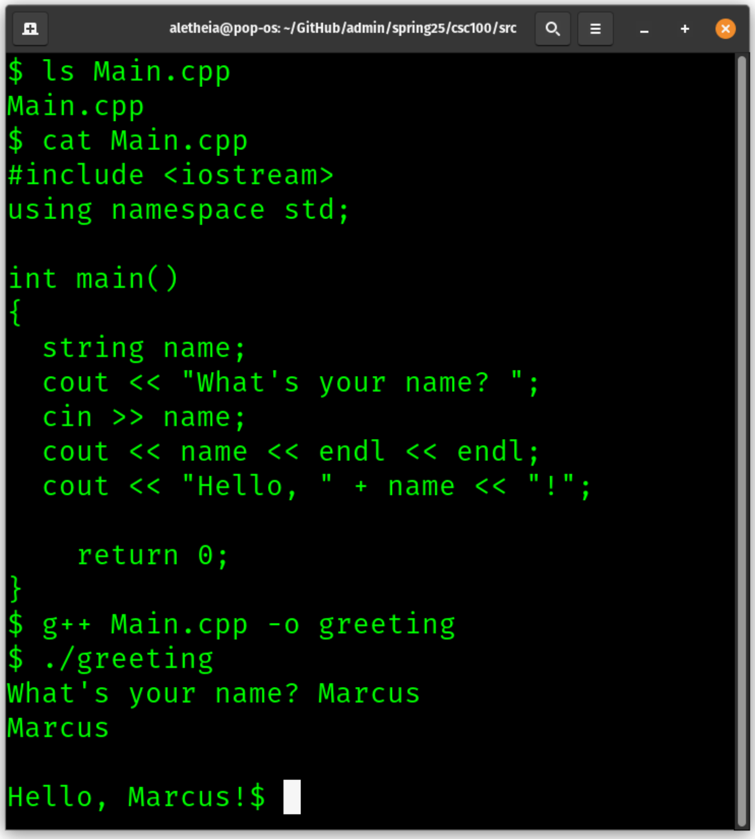
```
#include <iostream> // Input/output control
using namespace std; // Using stuff from the C++ standard library

int main(void) // the usual main program
{
  cout << "Hello, world!"; // direct the output to stdout
  return 0; // the usual END of main - 0 if successful
}
```

- It's just as (deceptively) simple to enter user input in C++. Replace the `cout` line with this and enter some input in the `STDIN` field.



- Notice that the program is called `Main.cpp` - .cpp is the default ending for C++ programs (compared to .c for C programs):
- You could download `Main.cpp`, compile and run it on the command-line (e.g. on the Google Cloud shell, `ide.cloud.google.com`):



- The Code:

```
string name; // declare string variable
cout << "What's your name? "; // ask for user input
cin >> name; // get user input from stdin
cout << name << endl << endl; // print user input
cout << "Hello, " + name << "!"; // print greeting
```

- **Explanations** (for lonely winter evenings):
  - `iostream` is the C++ header file for input and output
  - `std` is a `namespace`, a protected area that contains `cout` for output, `cin` for input, `string` and `vector` for string and vector identifiers, `endl` for new line, etc.

  - This is because there is a limited number of words and terms available, and different softwares can now use the same name but with a different `namespace` prefix.
  - The full version of `cout` is therefore `std::cout` etc.

- Input:

```
echo "Marcus" > input
cat input
```

```
Marcus
```

3 From `struct` in C to `class` in C++ - `Player`

- Our goal is to understand how to translate a C `struct` into a C++ `class` and why you'd want to do that.
- Here is a typical C-style `struct`:

```
struct Player {
  int x; // player's x position
  int y; // player's y position
  int health; // player's health points (> 0)
};
```

- There's a problem here: In C, all members of `Player` are `public` by default: There's no way to restrict access if someone wants to mess with your `Player`.
- Example: Negative `Player.health` points are meaningless but the compiler allows it - and the position `Player.x` can also be corrupted:

```
struct Player {
  int x; // player's x position
  int y; // player's y position
  int health; // player's health points (> 0)
};

struct Player John; // John is a Player now
```

```
John.health = -1000; // Invalid health!
John.x = 9999; // Out in the cold!

printf("Player's health (%d) and position (%d) are worrying.\n",
       John.health, John.x);
```

```
Player's health (-1000) and position (9999) are worrying.
```

4 Data hiding (aka encapsulation) in C++ - `Player.health`

- In C++, `Player` data can be hidden and controlled:

```
class Player {

private:      // private data
  int x, y, health;
};
```

- Let's try to mess with a player now:

```
class Player {

private:      // private data
  int x, y, health;
};

class Player Jane;
Jane.health = -1000;
Jane.x = 9999;
```

- In C++, if you don't specify data as `public`, they're `private`.

5 How to access `private` data - `getHealth()`

- Data that are `private` are accessed only indirectly through **methods**.
- You've already met one one those methods: `move_point` for the `Point` structure:

```
struct Point p;  // create a Point p
move_point(&p,dx,dy); // move p by dx in x-, and by dy in y-direction
```

- Methods are functions that belong to classes and act on their data. In C++, a method (or member function)
  1. is declared inside a `class`
  2. can access the class's `private` data
  3. is called using an object of the `class`
- Here's `Player` again but with a method that allows us to check the `Player`'s `health`:

```
class Player {
private:
  int health = 100;  // Player's private health
public:
  int getHealth() {
    return health;  // make Player's health public
  }
};
```

- Let's test it:

```
class Player {
private:
  int health = 100;  // Player's private health
public:
  int getHealth() {
    return health;  // make Player's health public
  }
};
 // Create a Player named Jane
class Player Jane;
// Get Jane's [private] health data
cout << "Player health = " << Jane.getHealth() << endl;
```

- C++ enforces data type and access control much more strongly than C.

6 How to alter `private` data - `takeDamage`

- Now we know how to get to the `private` data - to alter them, we need a new method. In the example, we're adding the `takeDamage` method, and we're retaining the `getHealth` method (we need it to check).
- Example: Create a `Player` that can take damage

```
class Player { // a Player class

private: // private data
  int health = 100;  // Player's health is hidden

public:  // public member function

  int getHealth() {
    return health;  // make Player's health public
  }

  void takeDamage(int amount) {
    health -= amount; // reduce Player's health by amount
  }
};
```

- In the `main` program, we're adding a `Player` who can take damage:

```
class Player { // a Player class

private: // private data
  int health = 100;  // Player's health is hidden

public:  // public member function

  int getHealth() {
    return health;  // make Player's health public
  }

  void takeDamage(int amount) {
    health -= amount; // reduce Player's health by amount
  }
};
```

```
class Player John; // John's a Player

// What's his health like?
cout << "Before the fight: Player's health = " << John.getHealth() << endl;

// In a fight, John takes damage
John.takeDamage(50);

// What's his health like?
cout << "After the fight: Player's health = " << John.getHealth() << endl;
```

7 Challenge: Heal the `Player` with `heal`

- Use the code developed so far, and add a `heal` method that increases a `Player`'s `health`:
  1. Create `Player` class with `private` member `health`, and `public` methods `getHealth`, `takeDamage`, and `heal`.
  2. Create `main` program, create a `Player`, print his `health`, let him `takeDamage` (50), print his `health`, `heal` him (80), print `health`.
- Sample output:

```
Player's health = 100
Player's health after battle = 50
Player's health after healing = 130
```

- Here is the starter code:

```
// include input / output
// use standard names

/* class definition */
// Create a Player class


// private data

// Player's health (initially 100)
```

```
// public data

// Return Player's health
// int getHealth(void)

// Reduce Player's health by amount
// void takeDamage(int)

// Heal Player by amount
// void heal(int)

// END CLASS

/* main program */

// BEGIN MAIN
// Create a Player [name]

// PRINT Player's health + new line

// Player takes damage (50)

// PRINT Player's health after battle + new line

// Player heals (80)

// PRINT Player's health after healing + new line

// END MAIN
```

- Solution:

```
#include <iostream> // include input / output
using namespace std; // use standard names

// Create a Player class
class Player {

private: // private data
```

```
  // Player's health (initially 100)
  int health = 100;

public: // public data

  // Return Player's health (int)
  int getHealth() {
    return health;
  }

  // Reduce Player's health by amount (int)
  void takeDamage(int amount) {
    health -= amount;
  }
  // Heal Player by amount (int)
  void heal(int amount) {
    health += amount;
  }
};

/* main program */
int main(void)
{
  // Create a Player
  Player John;
  // PRINT Player's health
  cout << "Player's health = " << John.getHealth() << endl;
  // Player takes damage (50)
  John.takeDamage(50);
  // PRINT Player's health after battle
  cout << "Player's health after battle = " << John.getHealth() << endl;
  // Player heals (80)
  John.heal(80);
  // PRINT Player's health after healing
  cout << "Player's health after healing = " << John.getHealth();

  return 0;
}
```

8 Bonus challenge: Cap `Player health` at 100

- Modify the previous program to cap the `health` at 100. That is, `if health` is above 100, reset it to 100.
- Using the same values as before (take 50 damage, heal 80), the sample output is now:

```
Player's health = 100
Player's health after battle = 50
Player's health after healing = 100
```

- Solution:

```
#include <iostream> // include input/output
using namespace std; // use standard names

// Create a Player class
class Player {
private: // private data
  // Player's health (initially 100)
  int health = 100;

public: // public data

  // Return Player's health (int)
  int getHealth() { return health; }
  // Reduce Player's health by amount (int)
  void takeDamage(int amount) { health -= amount; }
  // Heal Player by amount (int)
  void heal(int amount) {
    health += amount;
    if (health > 100) health = 100;
  }
};

/* main program */
int main(void) {
  // Create a Player
  Player John;
  // PRINT Player's health
```

```
cout << "Player's health = " << John.getHealth() << endl;
// Player takes damage (50)
John.takeDamage(50);
// PRINT Player's health after battle
cout << "Player's health after battle = " << John.getHealth() << endl;
// Player heals (80)
John.heal(80);
// PRINT Player's health after healing
cout << "Player's health after healing = " << John.getHealth();

return 0; }
```

9 Initializing before and cleaning up after `class`

- A C `struct` requires manual setup and teardown.
- A C++ `class` automates initialization and cleanup with special methods called **constructor** and **destructor**.
- Doing this makes code safer, more readable, and more reusable (these are the three big reasons for OOP to scale[1] code development).
- Example: A **constructor** `Player(int)` in the `Player class` will set `health` when the player is created, or "goes into scope".
- Example: A **destructor** `Player()` in the `Player class` will delete all memory allocated to a `Player` when finishes, or "goes out of scope".

10 Constructor: Set `Player health` when a player is created

- This is the `Player` class so far (without the `health` cap):

```
class Player {
private:
```

[1] "Scale" is what distinguishes learning from production: When you're learning you craft small pieces of code hoping it will run at all for you alone, on your single computer. When you're creating production code, you write (or monitor, or maintain, more likely), large collections of code that runs on millions, or billions of computers, and that must run safely and smoothly. That is "scale".

```
  int health = 100;
public:
  int getHealth() { return health; };
  void takeDamage(int amount) { health -= amount; }
  void heal(int amount) { health += amount; }
};
```

- Modify this class:
  1. Add a constructor to allow `health` to be set when the `Player` is created.
  2. Test the constructor in a `main` program by creating a `Player` named `John` with initial `health` of 150.
  3. Display `John`'s `health`.
- Starter code: onecompiler.com/cpp/43gpntxa2

```
// Include Input/Output functions
// Use standard namespace

// Create Player class

// BEGIN class
// private data:
// Player's health
// public data
// Initialize Player with health h
// Return Player's health
// END class

// main program
// BEGIN main
// Create a Player named John with a health of 150

// Display John's health

// END main
```

- Solution:

```
#include <iostream>
using namespace std;

class Player {
private:
  int health; // Player's health is private
public:
  Player(int h) { // constructor
    health = h;
  }
  int getHealth() { return health; }
};

int main(void)
{
  // create a Player named John with a health of 150
  Player John(150);
  // display John's health
  cout << "John's health is " << John.getHealth();
  return 0;
}
```

11 Initializer Lists Syntax

- A better style for initializing member variables in constructors is using a list:

```
class Player {
private:
  int health;
  int level;
public:
  // Default constructor
  Player(int h) : health(h) { // initializer list
    cout << "Player created with health: " << health << endl;
  }
};

int main() {
  Player John(100);
```

```
}
```

- In the example, the constructor informs us when the `Player` was created.

- Any number of members can be initialized, e.g. this list would create a `Player` with default `health=100`, `level=1`, and `build=3`:

  ```
  Player() : health(100), level(1), build(3) { }
  ```

- Why is this better?

  1. Member variables are initialized directly at construction time (rather than default-constructed first and then assigned a value)
  2. You must use initializer lists for `const` members or references because they cannot be assigned to after construction:

     ```
     class Person {
       const int ssn; // Person's SSN does not change
     public:
       Person(int i) : ssn(i) { } // must use initializer list
     };
     ```

12 Destructor: Clean up `Player` at the end of the game

- The destructor method runs when the `Player` object goes "out of scope", when the `Player` has died and the `Player` data are no longer needed.

- This is useful for keeping a log, or for memory cleanup.

- Here is the `Player` class with a default destructor method `~Player`. This destructor is called automatically.

  ```
  class Player {
  private:
    int health;
    int level;
  public:
    // Default constructor
  ```

```
  Player(int h) : health(h) {
    cout << "Player created with health: " << health << endl;
  }
  // Default destructor
  ~Player() {
    cout << "Player destroyed. " << endl;
  }
};
```

- Let's test it:

  ```
  class Player {
  private:
    int health;
    int level;
  public:
    // Default constructor
    Player(int h) : health(h) {
      cout << "Player created with health: " << health << endl;
    }
    // Default destructor
    ~Player() {
      cout << "Player destroyed. " << endl;
    }
  };

  int main() {
    Player John(100);
   }
  ```

- This looks as if the `Player John` was created and instantly destroyed. But that's not true: `John` lived for the duration of the `main` program.

- This is more obvious if we give `John` a fighting chance:

  ```
  class Player {
  private:
    int health;
    int level;
  public:
  ```

```
  // Default constructor
  Player(int h) : health(h) {
    cout << "Player created with health: " << health << endl;
  }
  // Default destructor
  ~Player() {
    cout << "Player destroyed. " << endl;
  }
};

int main() {
  Player John(100);
  cout << "Player is fighting monsters...\n" << endl;
  sleep(2);
  cout << "Player gains 50 XP...\n" << endl;
  sleep(2);
  cout << "Main function is about to end.\n" << endl;
 }
```

- I'm going to run this program (`fight.cpp`) on the command-line so that you can see what's going on:

```
g++ fight.cpp -o fight
./fight
```

- Output:

```
$ ./fight
Player created with health: 100
Player is fighting monsters...

Player gains 50 XP...

Main function is about to end.

Player destroyed.
$
```

- Unlike in "garbage-collecting" programming languages like Java or Python, in C++ you know exactly when objects are destroyed - that's a feature!

13 Challenge: Create two `Player` characters

- Create two `Player` characters, name them `Alice` and `Bob`, start them off with `health` 100 and 200, respectively. Let them `takeDamage` (50), show the resulting `health`, and finish.
- Tip: You need `Player::health`, and the `class` methods
  1. `Player(int)` to create a `Player`
  2. `~Player()` to destroy a `Player`
  3. `getHealth()` to `return health`
  4. `takeDamage(int)` to reduce `health` after a fight.
- Sample output:

```
Player created, health = 100
Player created, health = 200
Game in progress...

Game is about to end...

Alice's health: 50
Bob's health: 150
Game over.

Player destroyed
Player destroyed
```

- Solution:

```
#include <iostream>
using namespace std;
class Player {
private:
  int health;
public:
  Player(int h) : health(h) {
    cout << "Player created, health = " << health << endl;}
  ~Player() { cout << "Player destroyed" << endl; }
  int getHealth() { return health;}
```

```
  void takeDamage(int amount) { health -= amount; }
};
int main(void)
{
  Player Alice(100), Bob(200);
  cout << "Game in progress...\n" << endl;
  Alice.takeDamage(50);
  Bob.takeDamage(50);
  cout << "Game is about to end...\n" << endl;
  cout << "Alice's health: " << Alice.getHealth() << endl;
  cout << "Bob's health: " << Bob.getHealth() << endl;
  cout << "Game over.\n" << endl;
  return 0;
}
```

14 The `this` Pointer

- When you define a method like `takeDamage` inside a `class` like `Player`, you are writing instructions for what every object of that class (`John`, `Alice`, `Bob`) should be able to do.
- But how does the method know which object it is working on?
- The special keyword `this` is a pointer to the current object, the one calling the method, e.g. `Alice` in `Alice.takeDamage(50)`.
- Think of `this` as the way an object says "me". When a `Player` says `this->health`, it means "my health."
- Example: Here, the `class Player` has a member variable `health` (full name `Player::health`), and the constructor has a parameter also named `health`. `this` is used to keep them apart:

```
class Player {
private:
  int health;
public:
  Player(int health) {
    this->health = health; // LHS: ptr to member, RHS: parameter
  }
  int getHealth() { return this->health; }
```

```
};

#include <iostream>
using namespace std;
int main() {
  Player Jack(100);
  cout << "Player's health: " << Jack.getHealth() << endl;
}
```

- Why is this useful?

  1. To resolve naming conflicts when constructor parameters or method arguments have the same name.

     ```
     Player(int health) {
       this->health = health;  // assign parameter to member
     }
     ```

  2. To return the object itself when chaining methods together:

     ```
     Player& setHealth(int h) {
       this->health = h;
       return *this;
       // allows chaining like: player.setHealth(100).takeDamage(10);
     }
     ```

  3. To reinforce object identity.

15 `static` members are shared across all instances

- An **instance** is another word for an object of a class. So `Player Jack(100);` creates a `Player` named `Jack` - Jack is an instance.
- Some methods are defined as `static` because they don't operate on `class` members. For example a method `Player::getCount` that counts the number of `Player` objects in the game.
- Example code:

```
class Player {
private:
  int health;
  static int count; // static member variable
```

```
public:
  // constructor
  Player(int health) {
    this->health = health;
    count++; }
  // destructor with counter variable
  ~Player() { count--; }
  // get Player count
  static int getCount() { return count; }
};
```

– Test this:

```
#include <iostream>
using namespace std;

class Player {
private:
  int health;
  static int count; // static member variable
public:
  // constructor
  Player(int health) {
    this->health = health;
    count++; }
  // destructor with counter variable
  ~Player() { count--; }
  // get Player count
  static int getCount() { return count; }
};
int Player::count = 0; // initialize Player::count

int main() {
  { // BEGIN scope
    Player Jack(100); // create Player
    cout << "Player count: " << Player::getCount() << endl;

    Player Alice(100); // create Player
    cout << "Player count: " << Player::getCount() << endl;
  } // END scope
  cout << "Player count: " << Player::getCount() << endl;
```

```
  return 0;
}
```

  - Remember that the default destructor is only invoked when the objects to "out of scope". This happens at the end of `main` but then we can no longer use `Player::getCount`. Therefore I added a scope by nesting the `Player` code inside `{ }` which is "scope".
  - Note that `this` does not operate on `static` methods of a `class` because these methods do not belong to any object - they can be called without creating an object.

16 Challenge: `Enemy class`

- Create an `Enemy` class with:
  1. A constructor
  2. A destructor
  3. Private `strength` member
  4. `attack()` method returning damage (e.g. `strength / 2`)
  5. Static member tracking total enemies
- Sample Output

```
Enemy created with strength 80
Enemy created with strength 100
Enemies active: 2
Attack damage: 40
Enemy destroyed
Enemy destroyed
Enemies active: 0
```

- Solution:

```
#include <iostream>
using namespace std;

class Enemy {
private:
  int strength;
```

```
  static int active;
public:
  Enemy(int strength) { // constructor
    this->strength=strength;
    cout << "Enemy created with strength " << this->strength << endl;
    active++;
  } //
  ~Enemy() { // destructor
    cout << "Enemy destroyed." << endl;
    active--;
  }
  int attack() {
    return this->strength/2;
  }
  static int getActive() {
    return active;
  }
};

int Enemy::active = 0;
int main(void)
{
  { // BEGIN battle
    Enemy Sauron(80);
    Enemy Saruman(100);
    cout << "Enemies active: " << Enemy::getActive() << endl;
    cout << "Attack damage: " << Sauron.attack() << endl;
  } // END battle
  cout << "Enemies active: " << Enemy::getActive() << endl;
  return 0;
}
```

17 Summary: Object Lifecycle in C++

- **Construction**: Objects are automatically initialized using a **constructor** when declared.
  - Example: `Player John(100);` runs the `Player(int)` constructor.

  - Constructors can take parameters or use initializer lists for clean, direct setup.

- **Usage / Lifetime**: Objects remain alive and usable for the duration of their **scope**.
  - Methods like `takeDamage()` or `getHealth()` operate on the object during this phase.
  - The `this` pointer refers to the current object and is used to access members clearly.

- **Destruction**: When an object goes **out of scope**, its **destructor** is automatically called.
  - Used to log destruction or clean up memory/resources.
  - Order of destruction is the reverse of construction — last-in, first-out (LIFO).

- **Static Members & Lifecycle Awareness**:
  - Use static counters (e.g., `Player::count`) to track how many objects are alive.
  - Static methods (like `getCount()`) can observe lifecycle changes from outside any object.

- **Scopes Reveal Lifecycle Timing**:
  - Wrapping object creation in a nested block (= { ... } =) shows **when** the destructor runs.
  - Helpful for visualizing stack-based memory and RAII (Resource Acquisition Is Initialization).

- **Why It Matters**:
  - Predictable object lifetimes help avoid memory leaks and bugs.
  - C++ gives fine-grained control — unlike garbage-collected languages.
  - Mastering lifecycle is foundational for managing resources, especially in larger programs.