Suppose we have several variables that we want to keep "bundled together" (e.g., they're all part of a larger, whole "thing"). Enter **structs**

- Struct stands for "structure"
- A struct is a container (like an array)---a group of zero or more values
- However, they're heterogeneous (**unlike arrays!**)---the contained values can be of different types
- Values are **named** rather than **indexed**. That is, a structure is basically just a group of variables
- Before you can create a structure itself, you have to define a **structure type**
- Structure type: A "blueprint" for a certain kind of structure. Specifies the variables contained within a structure of said type.

<u>Defining your very own structure type!</u>

Syntax:

```
struct <name of struct type> {
     <variable type> <variable name>;
     <variable type> <variable name>;
     ...
     <variable type> <variable name>;
}; // ← Mind the semicolon
```

Example:

```
struct baseball_player {
     string first_name;
     string last_name;
     int age;
     double batting_average;
}
```

- A structure type is a type of data! Once you've defined the `baseball_player` structure type, you can think of it like any other type (e.g., `int, double, string`, etc.).
- Now, we can create `baseball_player` structures. Declare them like any other variable: `baseball_player my_baseball_player;`
    - `baseball_player` is the name of the **type**
    - `my_baseball_player` is the name of the **variable**, whose type is `baseball_player.` Terminology: `my_baseball_player` is said to be an "instance" of the `baseball_player` struct type.

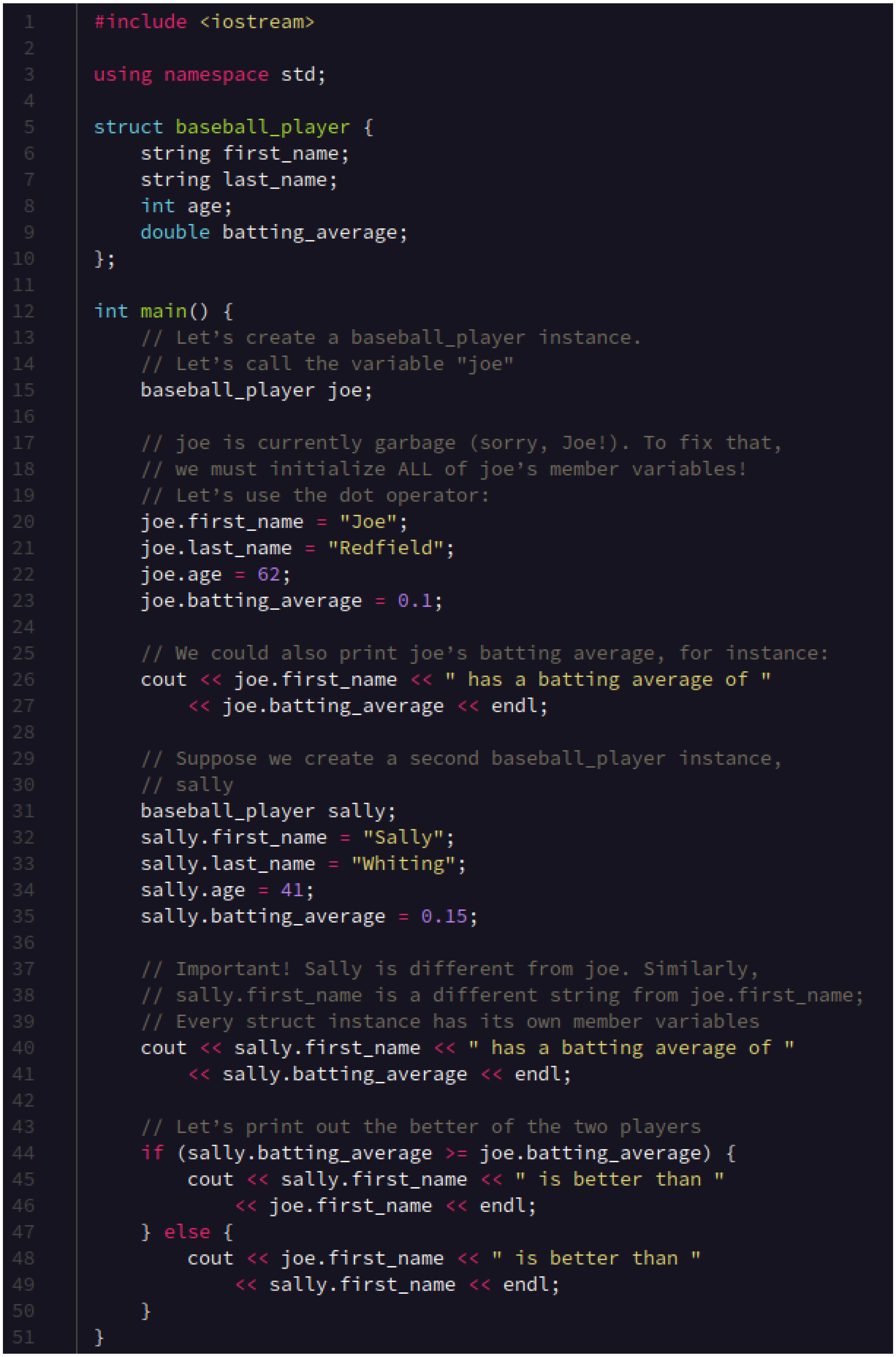
    - Important: the struct type definition must appear <u>before</u> you try to create a struct of that type
- Every `baseball_player` instance has its <u>own</u> `first_name, last_name, age,` and `batting_average.` To access `my_baseball_player's` variables (also called "members"), use the dot operator (AKA the "member access operator")

<u>Dot operator (member access operator)</u>:

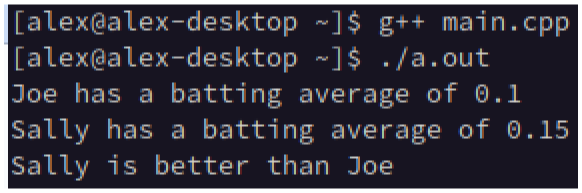
Syntax: <struct variable name>.<member variable name>

Note: This gives you both read and write access to the struct variable's internal member called <member variable name>.

<u>Simple struct example</u>:



```cpp
#include <iostream>

using namespace std;

struct baseball_player {
    string first_name;
    string last_name;
    int age;
    double batting_average;
};

int main() {
    // Let’s create a baseball_player instance.
    // Let’s call the variable "joe"
    baseball_player joe;

    // joe is currently garbage (sorry, Joe!). To fix that,
    // we must initialize ALL of joe’s member variables!
    // Let’s use the dot operator:
    joe.first_name = "Joe";
    joe.last_name = "Redfield";
    joe.age = 62;
    joe.batting_average = 0.1;

    // We could also print joe’s batting average, for instance:
    cout << joe.first_name << " has a batting average of "
        << joe.batting_average << endl;

    // Suppose we create a second baseball_player instance,
    // sally
    baseball_player sally;
    sally.first_name = "Sally";
    sally.last_name = "Whiting";
    sally.age = 41;
    sally.batting_average = 0.15;

    // Important! Sally is different from joe. Similarly,
    // sally.first_name is a different string from joe.first_name;
    // Every struct instance has its own member variables
    cout << sally.first_name << " has a batting average of "
        << sally.batting_average << endl;

    // Let’s print out the better of the two players
    if (sally.batting_average >= joe.batting_average) {
        cout << sally.first_name << " is better than "
            << joe.first_name << endl;
    } else {
        cout << joe.first_name << " is better than "
            << sally.first_name << endl;
    }
}
```



```
[alex@alex-desktop ~]$ g++ main.cpp
[alex@alex-desktop ~]$ ./a.out
Joe has a batting average of 0.1
Sally has a batting average of 0.15
Sally is better than Joe
```

- Terminology: we refer to the relationship between a struct instance and its members as a "has-a" relationship.
    - joe "has a" first_name, last_name, age, and batting_average
    - sally "has a" first_name, last_name, age, and batting_average
    - That is, each struct instance owns its member variables.
- Once you've created a struct instance (e.g., `joe` or `sally` in the above example), they work just like any other non-array variable
    - You can pass them to functions
        - By default, they're passed "by value". That is, the argument struct instance is copied into the parameter struct instance. Suppose the argument is called `joe` and the parameter is called `player`. In this case, `joe.first_name` is a separate string variable from `player.first_name`.
    - You can return them from functions
        - Again, they're returned by value by default (copied)
    - You can create an array of structs, on the stack or the heap
    - You can create a multidimensional array of structs, on the stack or the heap
    - You can create pointers to structs (e.g., `baseball_player* p = &joe;`)
        - You can, of course, dereference those pointers
    - You can create references to structs
    - You can assign a struct instance to another struct instance. This performs a copy. See below.
    - Etc…
- However, you cannot print an entire struct instance (e.g., `cout << joe << endl;`)... At least, not for now (we'll learn how to make this possible later). You can, of course, print a struct's primitive and string members (e.g., `cout << joe.first_name << endl;`)

Assigning one struct instance to another struct instance

Syntax (as you'd expect): <struct instance 1> = <struct instance 2>

Example: `sally = joe; // Where both are declared as in the previous example`

Note: What this really does:

```
1. sally.first_name = joe.first_name;
2. sally.last_name = joe.last_name;
```

3. `sally.age = joe.age;`
4. `sally.batting_average = joe.batting_average;`

So if we printed sally.first_name after running this assignment, it would print “Joe”. If we printed sally.last_name, it would print “Redfield”. And so on…

A struct instance can “have a” struct instance (or multiple struct instances) as a member. For example, if a `baseball_player joe` “has a” `baseball_bat b`, and `joe.b` “has an” `int length`, then you could print the length of `joe`’s `baseball_bat` like so:

```
cout << "Joe's baseball bat length: " << joe.b.length << endl;
```

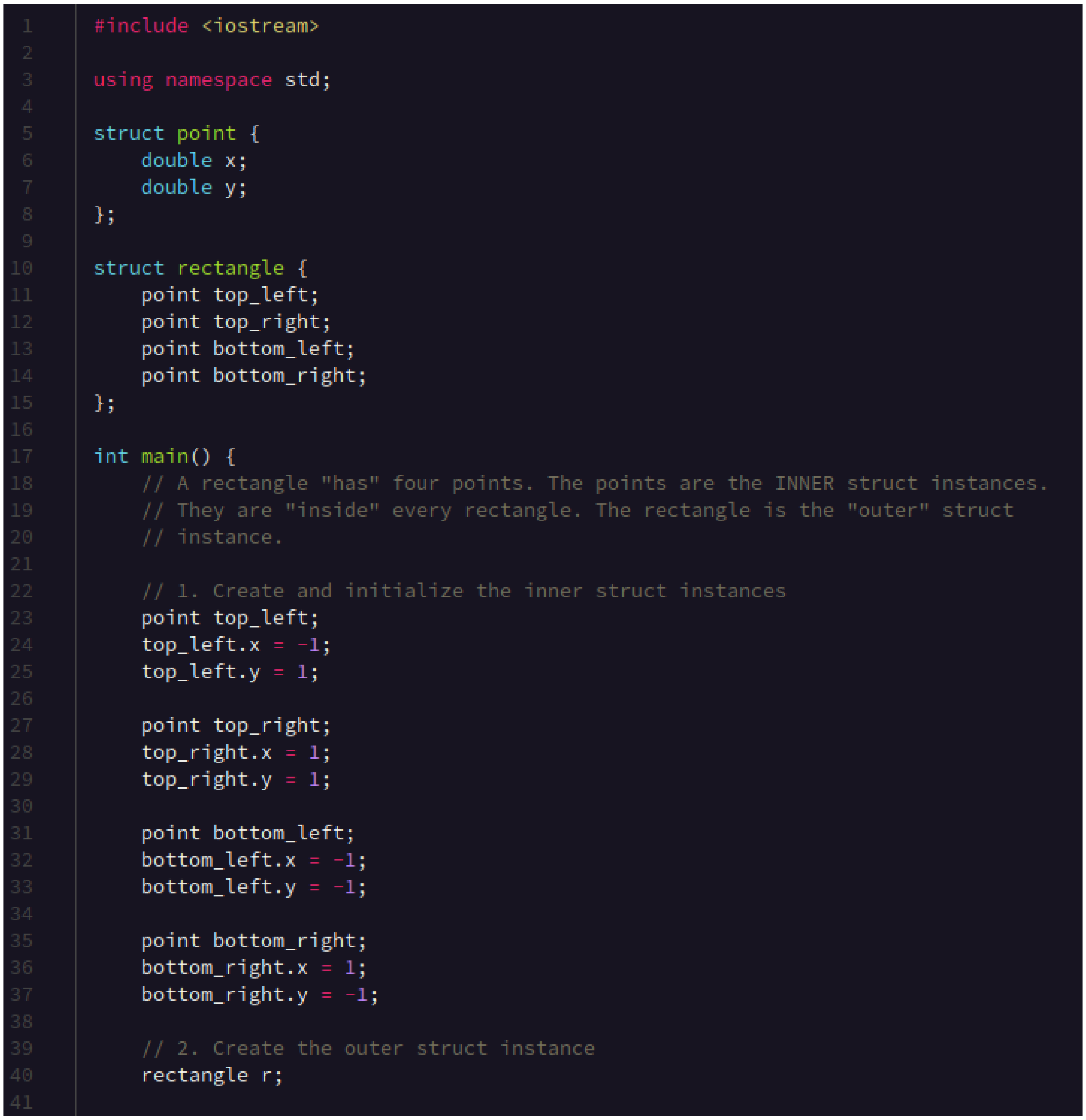
Notice the use of two dot operators—`joe.b` gives you access to `joe`’s `baseball_bat` instance called `b`, and the remaining `` `.length` `` gives you access to `b`’s `int` called `length`. See the option 1 example below regarding initialization—at line 54, we print the x coordinate of the top-left point of a rectangle (again, two dot operators).

Initializing nested structs is not too hard, but it requires some thought. There are two common methods:

<u>Option 1:</u>

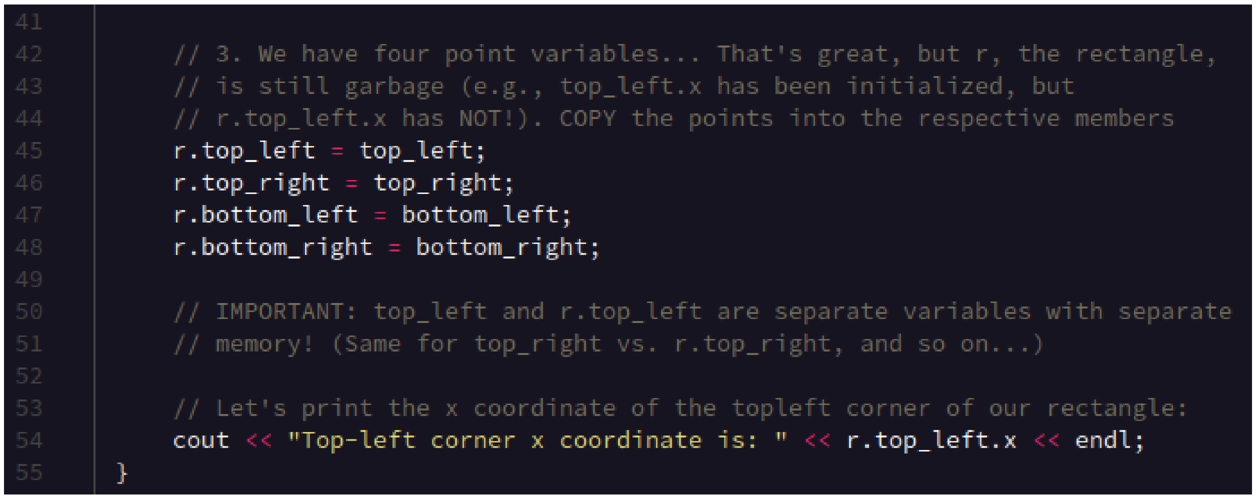
1. Create inner struct instance(s) and initialize them
2. Create the outer struct instance
3. <u>Copy</u> the constructed inner struct instances into the outer struct members

Example of option 1 (in this case, a rectangle “has” four points, each of which “have” x and y coordinates):



```cpp
#include <iostream>

using namespace std;

struct point {
    double x;
    double y;
};

struct rectangle {
    point top_left;
    point top_right;
    point bottom_left;
    point bottom_right;
};

int main() {
    // A rectangle "has" four points. The points are the INNER struct instances.
    // They are "inside" every rectangle. The rectangle is the "outer" struct
    // instance.

    // 1. Create and initialize the inner struct instances
    point top_left;
    top_left.x = -1;
    top_left.y = 1;

    point top_right;
    top_right.x = 1;
    top_right.y = 1;

    point bottom_left;
    bottom_left.x = -1;
    bottom_left.y = -1;

    point bottom_right;
    bottom_right.x = 1;
    bottom_right.y = -1;

    // 2. Create the outer struct instance
    rectangle r;

```

(scroll down for the rest of the code)

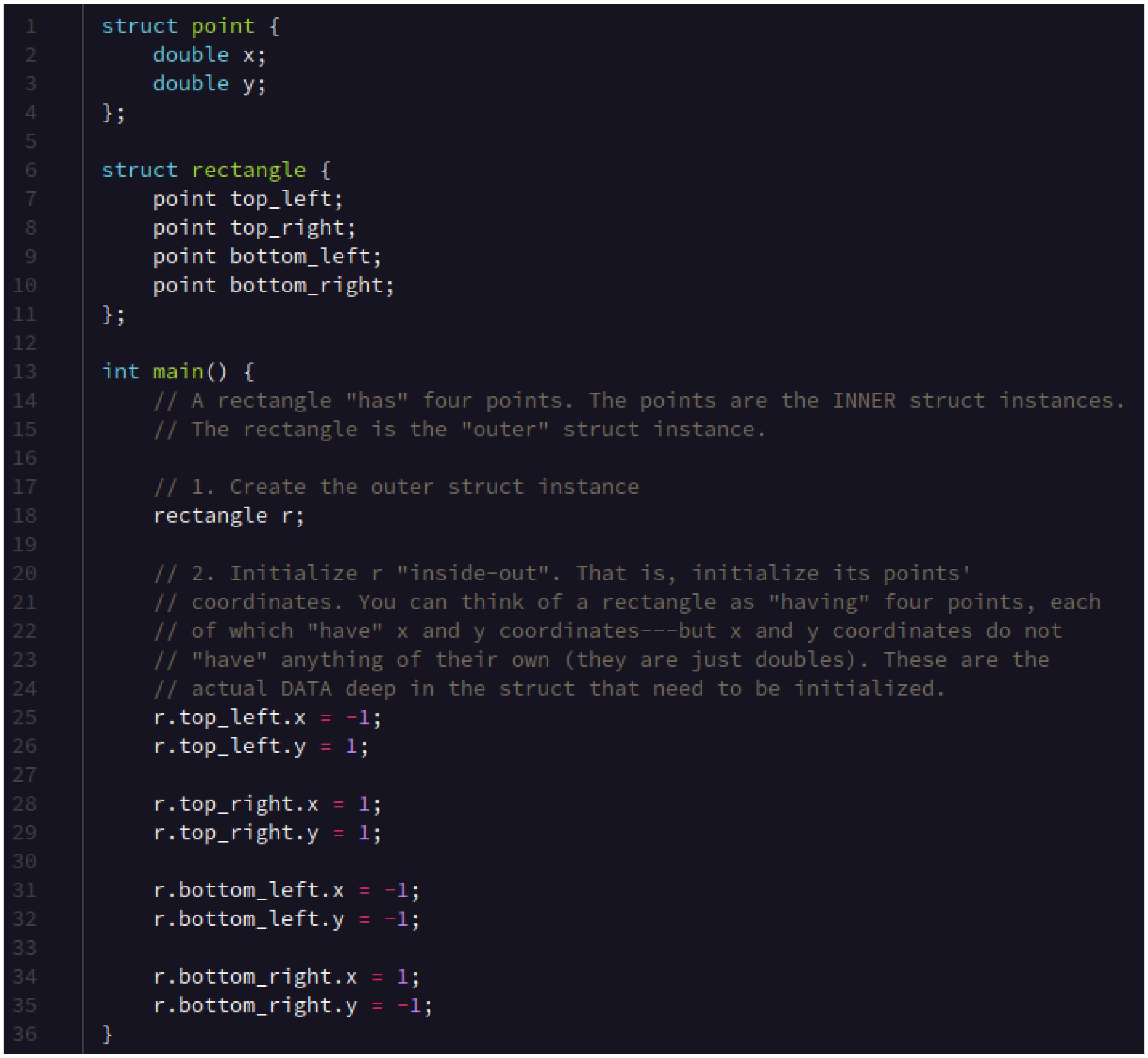


```
    // 3. We have four point variables... That's great, but r, the rectangle,
    // is still garbage (e.g., top_left.x has been initialized, but
    // r.top_left.x has NOT!). COPY the points into the respective members
    r.top_left = top_left;
    r.top_right = top_right;
    r.bottom_left = bottom_left;
    r.bottom_right = bottom_right;

    // IMPORTANT: top_left and r.top_left are separate variables with separate
    // memory! (Same for top_right vs. r.top_right, and so on...)

    // Let's print the x coordinate of the topleft corner of our rectangle:
    cout << "Top-left corner x coordinate is: " << r.top_left.x << endl;
}
```

Option 2:

1. Create the outer struct instance
2. Directly initialize all of its members "inside out", or "bottom-up" (see example below… this is hard to verbalize)

Example of option 2:



```cpp
struct point {
    double x;
    double y;
};

struct rectangle {
    point top_left;
    point top_right;
    point bottom_left;
    point bottom_right;
};

int main() {
    // A rectangle "has" four points. The points are the INNER struct instances.
    // The rectangle is the "outer" struct instance.

    // 1. Create the outer struct instance
    rectangle r;

    // 2. Initialize r "inside-out". That is, initialize its points'
    // coordinates. You can think of a rectangle as "having" four points, each
    // of which "have" x and y coordinates---but x and y coordinates do not
    // "have" anything of their own (they are just doubles). These are the
    // actual DATA deep in the struct that need to be initialized.
    r.top_left.x = -1;
    r.top_left.y = 1;

    r.top_right.x = 1;
    r.top_right.y = 1;

    r.bottom_left.x = -1;
    r.bottom_left.y = -1;

    r.bottom_right.x = 1;
    r.bottom_right.y = -1;
}
```

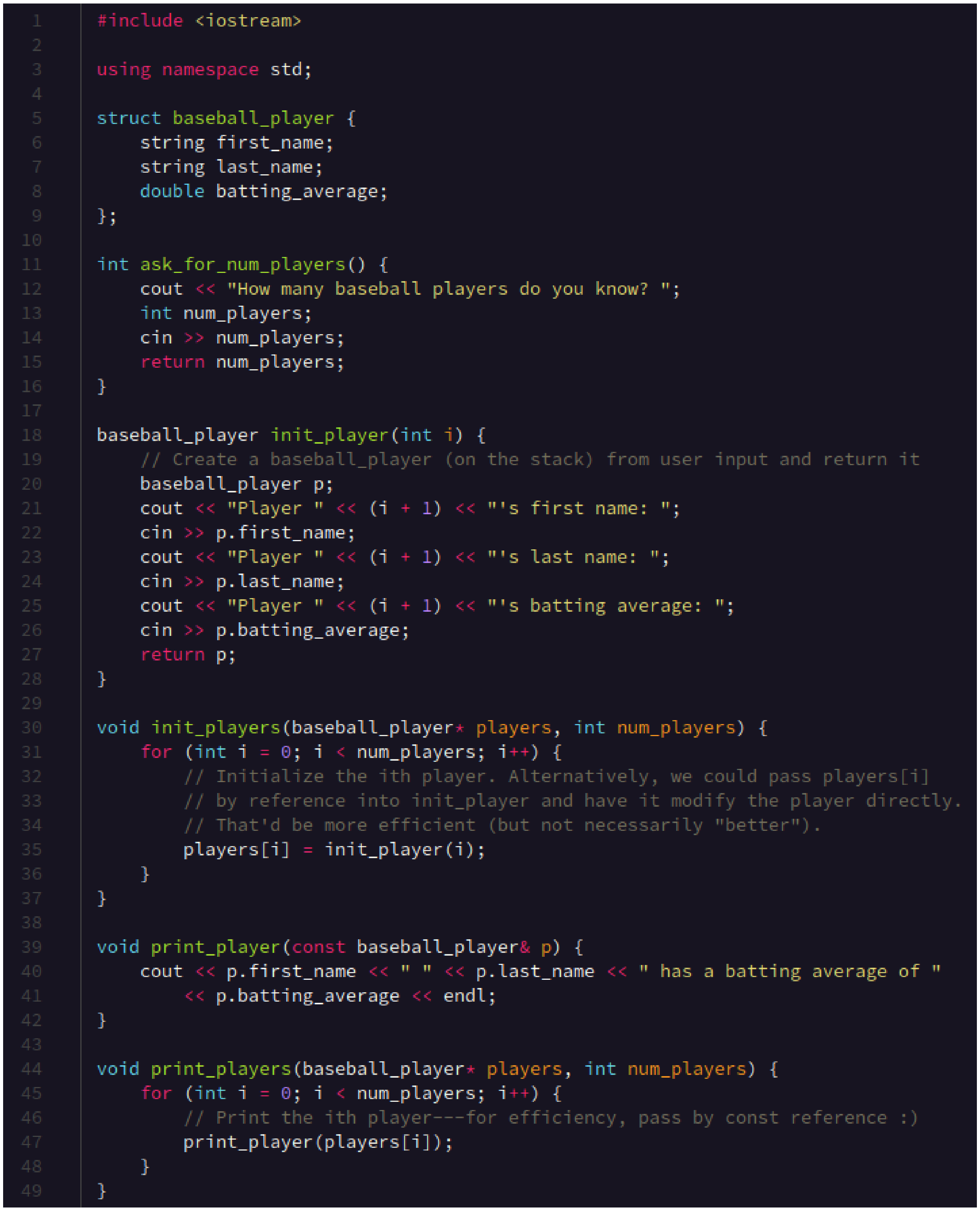
Either option works. However, with very deeply nested struct instances (structs within structs within structs within …), it's easier to forget to initialize some values with option 2. Option 1 is tried and true—it forces you to think about each nested struct instance one at a time (that's a good thing!)

- As mentioned, you can pass structs into functions as arguments / parameters
  - e.g., `void print_player_info(baseball_player p);`
  - Very common pattern: if the function doesn't need to modify the struct instance argument, pass it by constant reference. If it's a constant reference, then the function cannot modify the argument (much like pass by value). This prevents mistakes. However, references are basically pointers, which are only 8 bytes each. Copying a pointer (or reference) from argument -> parameter is cheap, but copying a large

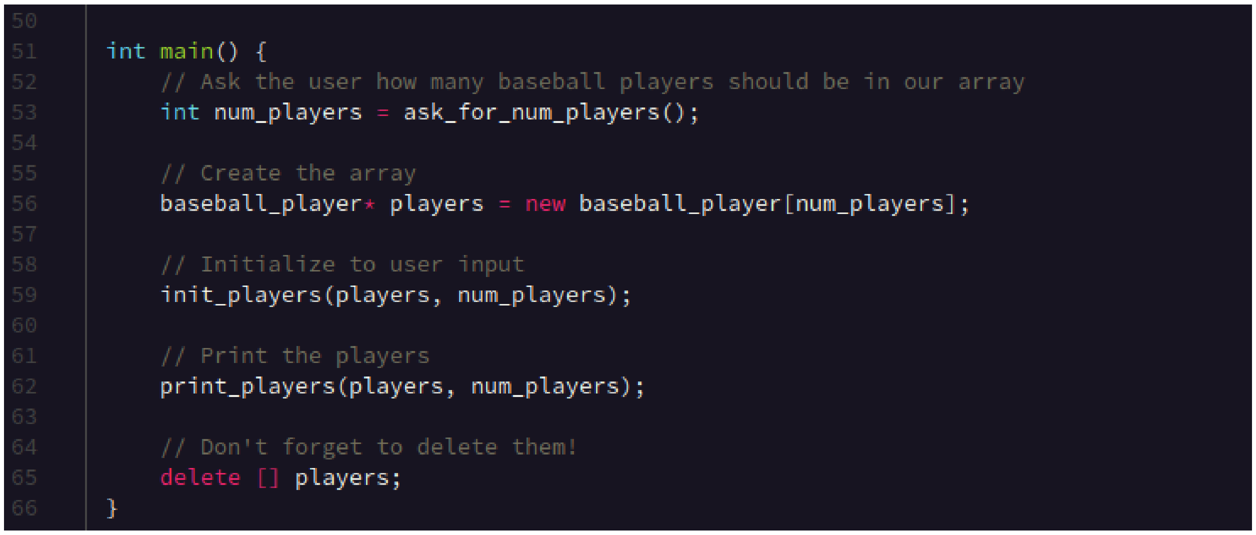
struct instance might be expensive:

e.g., `void print_player_info(baseball_player p);`

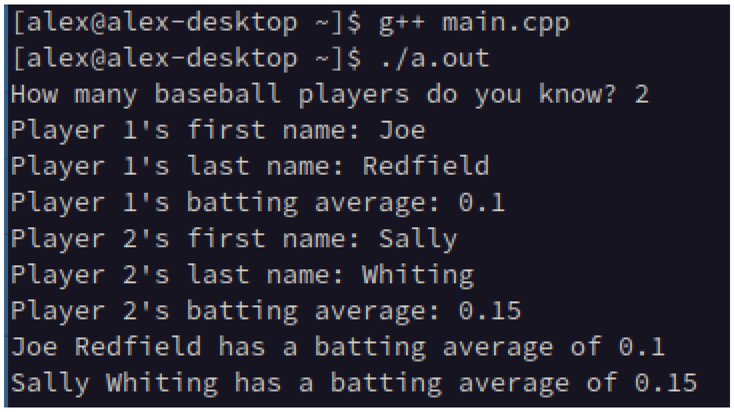
Here's a fully fledged example involving a 1D dynamic array of structs:



```cpp
#include <iostream>

using namespace std;

struct baseball_player {
    string first_name;
    string last_name;
    double batting_average;
};

int ask_for_num_players() {
    cout << "How many baseball players do you know? ";
    int num_players;
    cin >> num_players;
    return num_players;
}

baseball_player init_player(int i) {
    // Create a baseball_player (on the stack) from user input and return it
    baseball_player p;
    cout << "Player " << (i + 1) << "'s first name: ";
    cin >> p.first_name;
    cout << "Player " << (i + 1) << "'s last name: ";
    cin >> p.last_name;
    cout << "Player " << (i + 1) << "'s batting average: ";
    cin >> p.batting_average;
    return p;
}

void init_players(baseball_player* players, int num_players) {
    for (int i = 0; i < num_players; i++) {
        // Initialize the ith player. Alternatively, we could pass players[i]
        // by reference into init_player and have it modify the player directly.
        // That'd be more efficient (but not necessarily "better").
        players[i] = init_player(i);
    }
}

void print_player(const baseball_player& p) {
    cout << p.first_name << " " << p.last_name << " has a batting average of "
        << p.batting_average << endl;
}

void print_players(baseball_player* players, int num_players) {
    for (int i = 0; i < num_players; i++) {
        // Print the ith player---for efficiency, pass by const reference :)
        print_player(players[i]);
    }
}
```



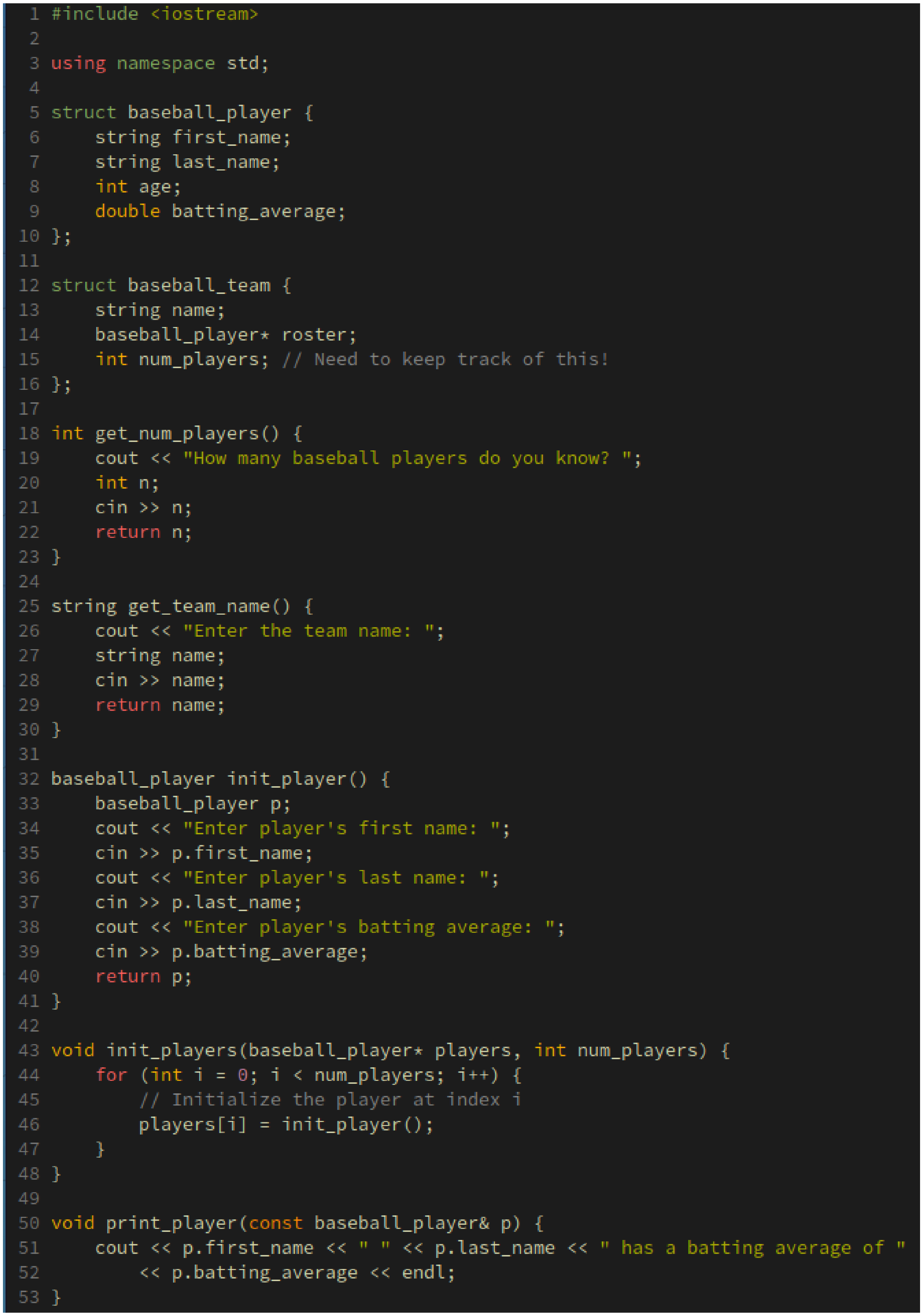
```cpp

int main() {
    // Ask the user how many baseball players should be in our array
    int num_players = ask_for_num_players();

    // Create the array
    baseball_player* players = new baseball_player[num_players];

    // Initialize to user input
    init_players(players, num_players);

    // Print the players
    print_players(players, num_players);

    // Don't forget to delete them!
    delete [] players;
}
```



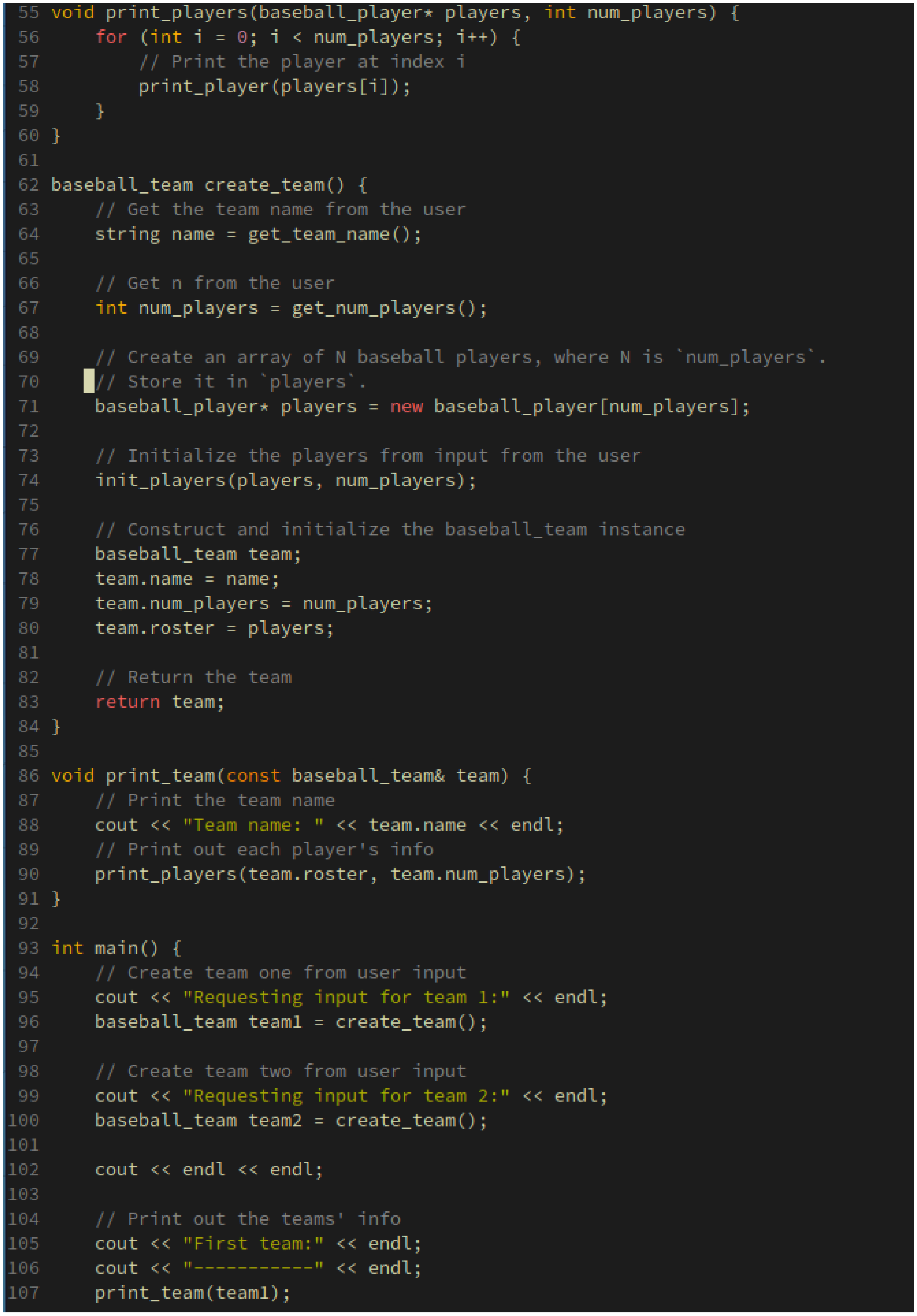
```
[alex@alex-desktop ~]$ g++ main.cpp
[alex@alex-desktop ~]$ ./a.out
How many baseball players do you know? 2
Player 1's first name: Joe
Player 1's last name: Redfield
Player 1's batting average: 0.1
Player 2's first name: Sally
Player 2's last name: Whiting
Player 2's batting average: 0.15
Joe Redfield has a batting average of 0.1
Sally Whiting has a batting average of 0.15
```

Of course, not only can you have an array of structs, but structs can also "have" arrays themselves (both on the stack and the heap). Note: if a struct "has an" array on the stack, then the array's size is specified as a compile-time constant in the declaration of the array member variable within the struct type definition. As such, all instances of that struct type will have an array of the same size!

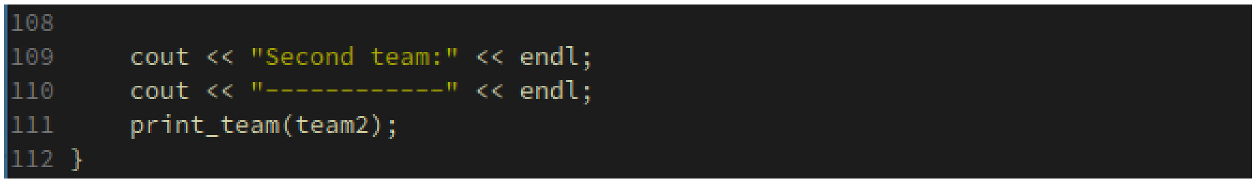
Here's an expansion of the previous example to include a baseball team struct type, each instance of which "has a" dynamic array of players (warning: long example)



```cpp
#include <iostream>

using namespace std;

struct baseball_player {
    string first_name;
    string last_name;
    int age;
    double batting_average;
};

struct baseball_team {
    string name;
    baseball_player* roster;
    int num_players; // Need to keep track of this!
};

int get_num_players() {
    cout << "How many baseball players do you know? ";
    int n;
    cin >> n;
    return n;
}

string get_team_name() {
    cout << "Enter the team name: ";
    string name;
    cin >> name;
    return name;
}

baseball_player init_player() {
    baseball_player p;
    cout << "Enter player's first name: ";
    cin >> p.first_name;
    cout << "Enter player's last name: ";
    cin >> p.last_name;
    cout << "Enter player's batting average: ";
    cin >> p.batting_average;
    return p;
}

void init_players(baseball_player* players, int num_players) {
    for (int i = 0; i < num_players; i++) {
        // Initialize the player at index i
        players[i] = init_player();
    }
}

void print_player(const baseball_player& p) {
    cout << p.first_name << " " << p.last_name << " has a batting average of "
        << p.batting_average << endl;
}
```

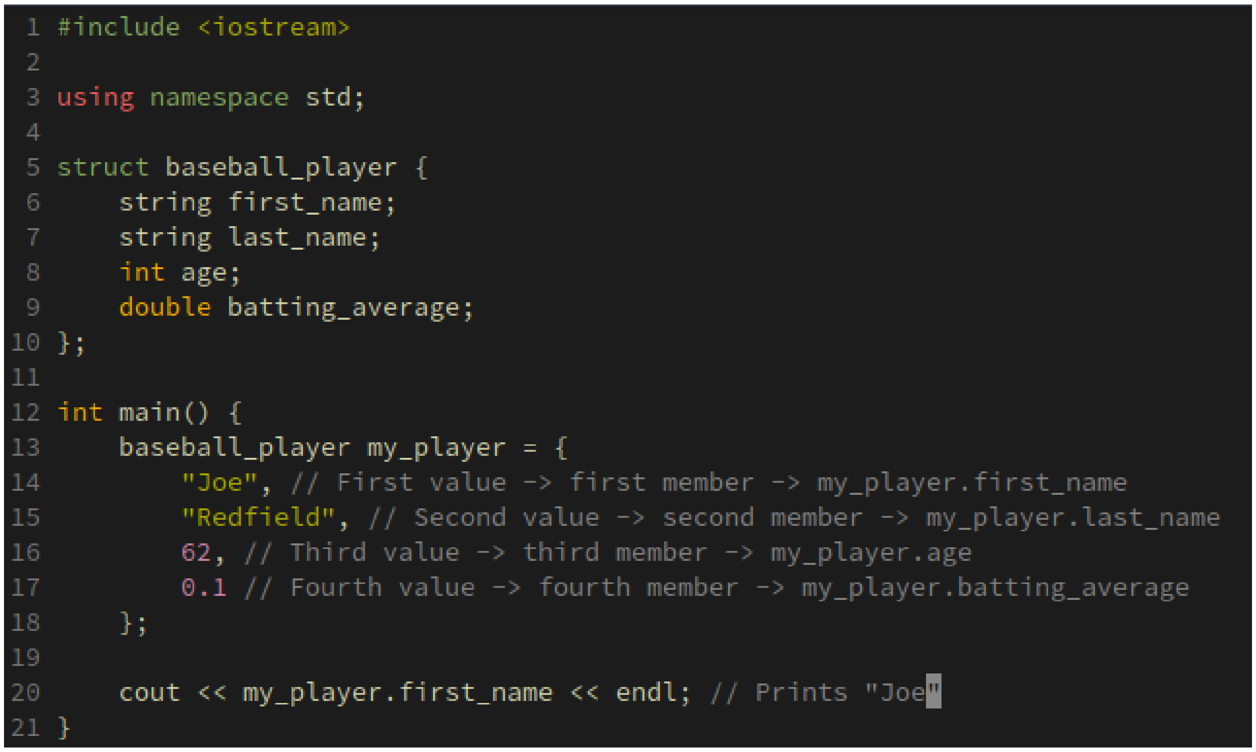


```cpp
void print_players(baseball_player* players, int num_players) {
    for (int i = 0; i < num_players; i++) {
        // Print the player at index i
        print_player(players[i]);
    }
}

baseball_team create_team() {
    // Get the team name from the user
    string name = get_team_name();

    // Get n from the user
    int num_players = get_num_players();

    // Create an array of N baseball players, where N is `num_players`.
    // Store it in `players`.
    baseball_player* players = new baseball_player[num_players];

    // Initialize the players from input from the user
    init_players(players, num_players);

    // Construct and initialize the baseball_team instance
    baseball_team team;
    team.name = name;
    team.num_players = num_players;
    team.roster = players;

    // Return the team
    return team;
}

void print_team(const baseball_team& team) {
    // Print the team name
    cout << "Team name: " << team.name << endl;
    // Print out each player's info
    print_players(team.roster, team.num_players);
}

int main() {
    // Create team one from user input
    cout << "Requesting input for team 1:" << endl;
    baseball_team team1 = create_team();

    // Create team two from user input
    cout << "Requesting input for team 2:" << endl;
    baseball_team team2 = create_team();

    cout << endl << endl;

    // Print out the teams' info
    cout << "First team:" << endl;
    cout << "-----------" << endl;
    print_team(team1);
```



```cpp

    cout << "Second team:" << endl;
    cout << "------------" << endl;
    print_team(team2);
}
```

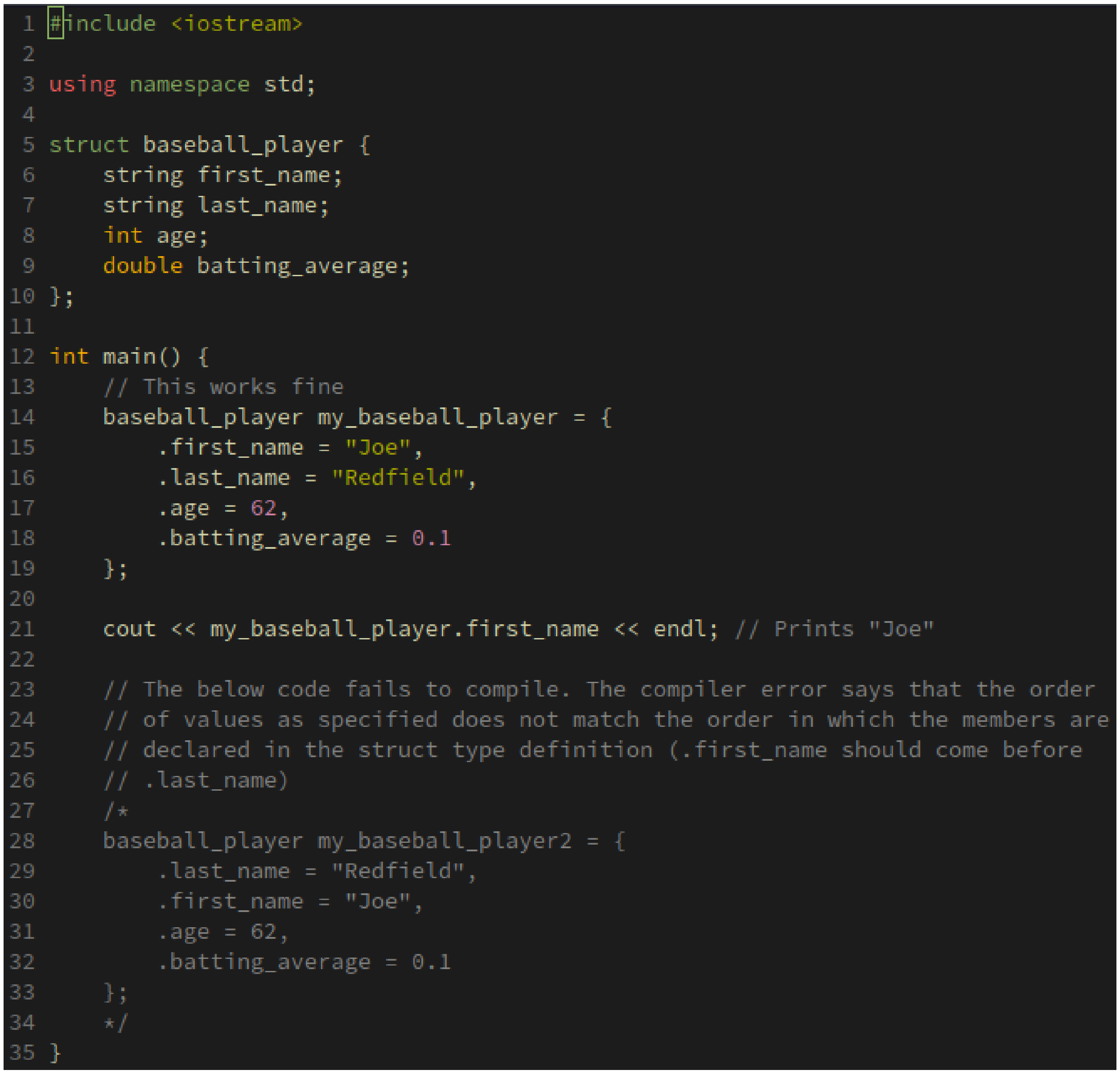
There’s a special syntax for initializing a struct instance’s members all at once upon declaration, called “list-style initialization”. You write out the values for the members in curly braces. The values get mapped to the member variables in declaration-order within the struct type definition. Here’s an example:



```cpp
 1 #include <iostream>
 2
 3 using namespace std;
 4
 5 struct baseball_player {
 6     string first_name;
 7     string last_name;
 8     int age;
 9     double batting_average;
10 };
11
12 int main() {
13     baseball_player my_player = {
14         "Joe", // First value -> first member -> my_player.first_name
15         "Redfield", // Second value -> second member -> my_player.last_name
16         62, // Third value -> third member -> my_player.age
17         0.1 // Fourth value -> fourth member -> my_player.batting_average
18     };
19
20     cout << my_player.first_name << endl; // Prints "Joe"
21 }
```

This is fine… **<u>But</u>**, consider what would happen if someone decided to swap lines 6 and 7 above. What would then be printed on line 20? Answer: “Redfield”. That is, this kind of list-style initialization is sensitive to the declaration order of member variables in the struct type definition. Maybe that’s not such a great thing…

If you want to protect yourself from such a mistake, you can use an alternative list-style initialization with dot syntax where, for each list value, you specify the name of the member that it should get stored in. The list values must still be specified in declaration order. However, in this case, if someone swapped lines 6 and 7, you'd get a compiler error telling you that the values are out of order. Here's what it looks like:



```
 1 #include <iostream>
 2 
 3 using namespace std;
 4 
 5 struct baseball_player {
 6     string first_name;
 7     string last_name;
 8     int age;
 9     double batting_average;
10 };
11 
12 int main() {
13     // This works fine
14     baseball_player my_baseball_player = {
15         .first_name = "Joe",
16         .last_name = "Redfield",
17         .age = 62,
18         .batting_average = 0.1
19     };
20 
21     cout << my_baseball_player.first_name << endl; // Prints "Joe"
22 
23     // The below code fails to compile. The compiler error says that the order
24     // of values as specified does not match the order in which the members are
25     // declared in the struct type definition (.first_name should come before
26     // .last_name)
27     /*
28     baseball_player my_baseball_player2 = {
29         .last_name = "Redfield",
30         .first_name = "Joe",
31         .age = 62,
32         .batting_average = 0.1
33     };
34     */
35 }
```