This lecture will teach you how to separate your program into multiple .cpp (and .h / .hpp) files, rather than writing all of your code in one giant file.

Build pipeline

First, recall that building your program happens in three steps:

1. Preprocessing (e.g., #include directives)
2. Compilation
3. Linking

- Preprocessing: directives beginning with # are processed / interpreted to modify the C++ source code in-place. Example:

  ```
  #include <iostream>
  ```

  This directive locates the iostream header file (installed in a system libraries folder on your computer) and copies / pastes its content in place of the include directive.
- Compilation: Each of your .cpp files is converted to "object code" (effectively, binary-encoded op codes that your CPU can interpret directly).
- Linking: If you had multiple .cpp files, all of their object code is linked together into one big executable file. Also, any libraries that you included in your program (e.g., the C++ standard library, like <iostream> and such) have their object code linked into your program as well.

That'll all be important soon. Next, let's talk about function declarations…

Breaking a .cpp file into multiple .cpp files

You can declare a function without immediately defining it. You do this by writing what's called a function prototype. A function prototype states that the function exists, but it will be defined later. To write a function prototype, you simply write out the function header (return type, name, parameters), followed by a semicolon instead of curly braces / a body.

In addition, you don't have to list the names of the parameters in a function prototype, but you do have to list their types. See below.

Function prototype examples:

```
// A function that adds two doubles and returns their sum. Notice
that
// we don’t list the name of the first double parameter. That’s okay.
// We don’t have to list the name of the second one either, but we
do.
// That’s also okay.
double add(double, double b);
```

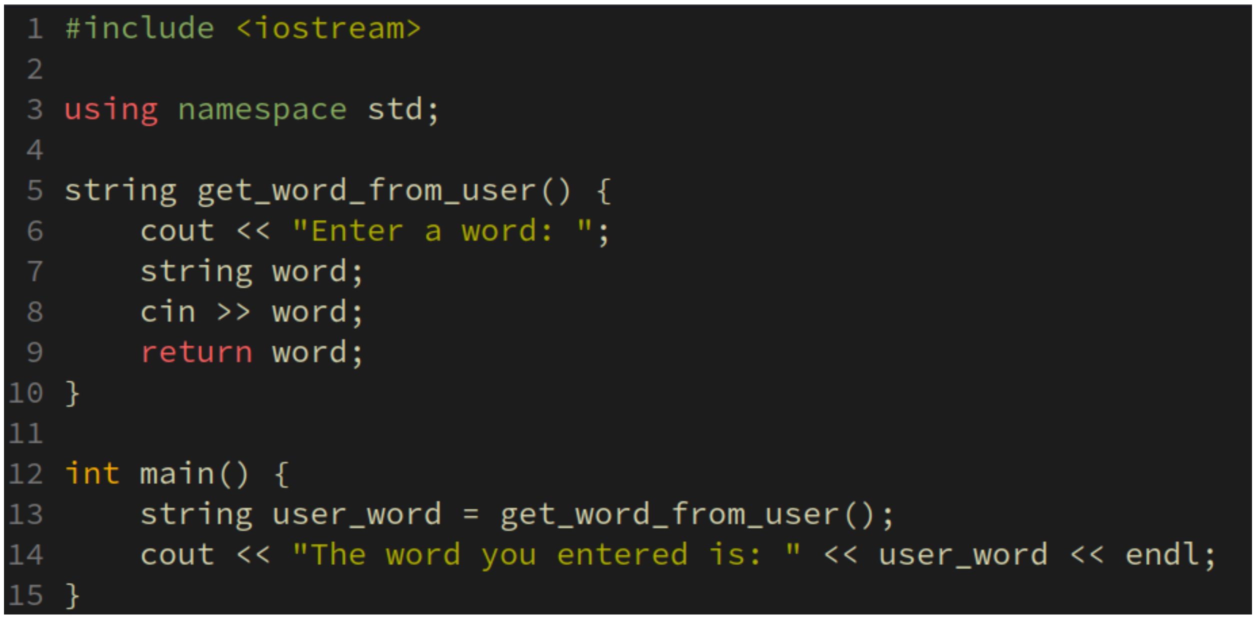
```
// A function that asks the user for a string via cin and returns it
string get_string_from_user();
```

What’s the point? Well…

- (**Criterion 1**) Each function that your entire program uses must be defined exactly once, in exactly one .cpp file
  - By definition, every program uses a main function… So main(), for example, must be defined exactly once, in exactly one .cpp file
  - But this is generally true for all functions that your program uses
- (**Criterion 2**) Every function that a .cpp file uses must be declared (prototyped) within that .cpp file before it can be used.

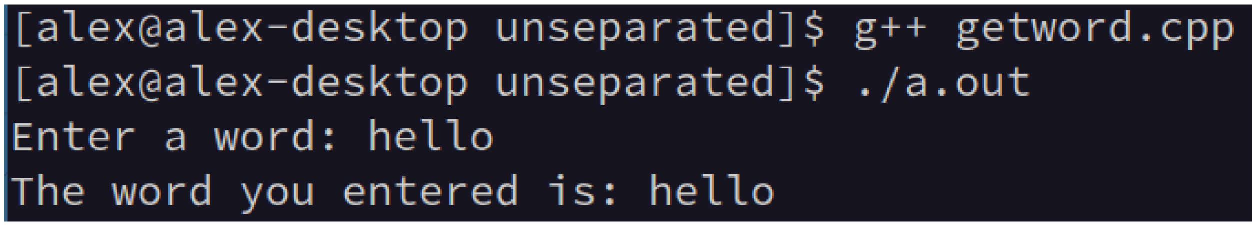
It’s crucial that your function prototypes match the function header in the corresponding definition exactly (return type, name, and parameter types must all match appropriately).

For instance, suppose we have a program that asks the user for a string and then prints it back out. Suppose the logic of asking the user for a string is modularized to its own function. Suppose all of the code is in `getword.cpp`:



```cpp
#include <iostream>

using namespace std;

string get_word_from_user() {
    cout << "Enter a word: ";
    string word;
    cin >> word;
    return word;
}

int main() {
    string user_word = get_word_from_user();
    cout << "The word you entered is: " << user_word << endl;
}
```

We could build and run it like so:

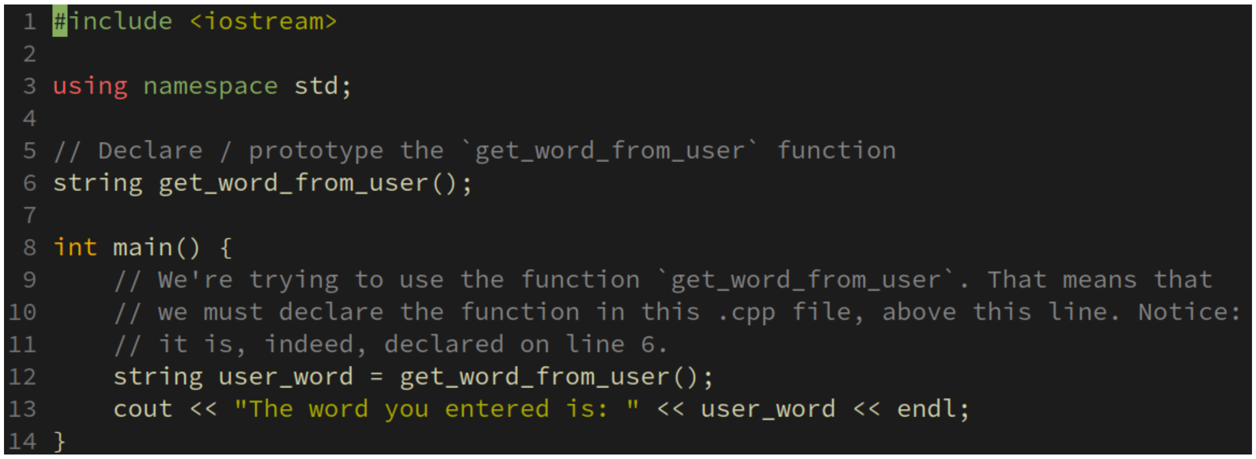


```
[alex@alex-desktop unseparated]$ g++ getword.cpp
[alex@alex-desktop unseparated]$ ./a.out
Enter a word: hello
The word you entered is: hello
```

Now, suppose we want to break this program down into two separate .cpp files—one that provides the `main()` function, and one that provides the `get_word_from_user()` function. Following criterion 1 and criterion 2 above, each of these functions must be defined exactly once in exactly one .cpp file, and every .cpp file that uses a function must declare it before using it. So, we're going to…

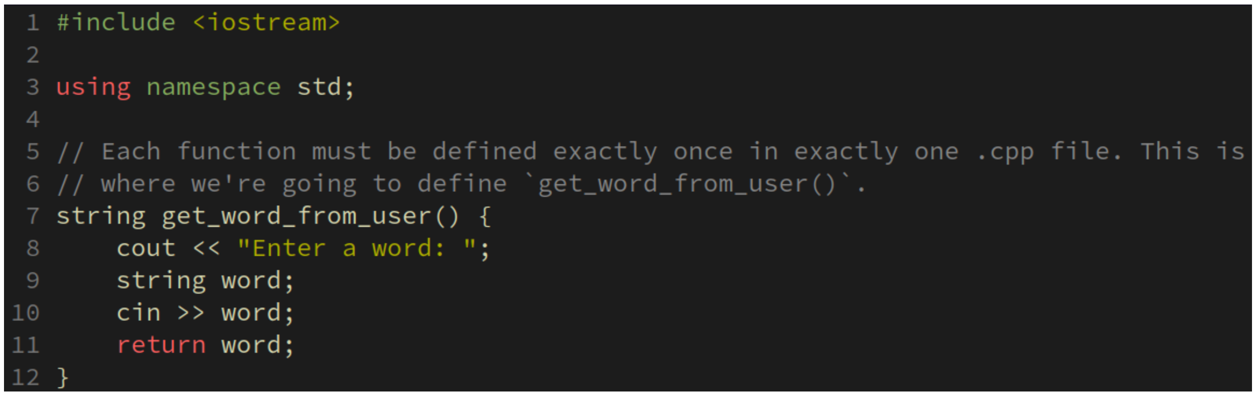
1. Create a main.cpp file
   a. This file will define the main() function.
   b. Notice that the `main()` function in `main.cpp` uses (calls) the `get_word_from_user()` function. Therefore, the `get_word_from_user()` function must be declared (prototyped) within `main.cpp`, before the `main()` function.
2. Create a separate `getword.cpp` file. This file will define the `get_word_from_user()` function.

Here's `main.cpp`:



```cpp
 1 #include <iostream>
 2 
 3 using namespace std;
 4 
 5 // Declare / prototype the `get_word_from_user` function
 6 string get_word_from_user();
 7 
 8 int main() {
 9     // We're trying to use the function `get_word_from_user`. That means that
10     // we must declare the function in this .cpp file, above this line. Notice:
11     // it is, indeed, declared on line 6.
12     string user_word = get_word_from_user();
13     cout << "The word you entered is: " << user_word << endl;
14 }
```

And here's `getword.cpp`:



```cpp
 1 #include <iostream>
 2 
 3 using namespace std;
 4 
 5 // Each function must be defined exactly once in exactly one .cpp file. This is
 6 // where we're going to define `get_word_from_user()`.
 7 string get_word_from_user() {
 8     cout << "Enter a word: ";
 9     string word;
10     cin >> word;
11     return word;
12 }
```

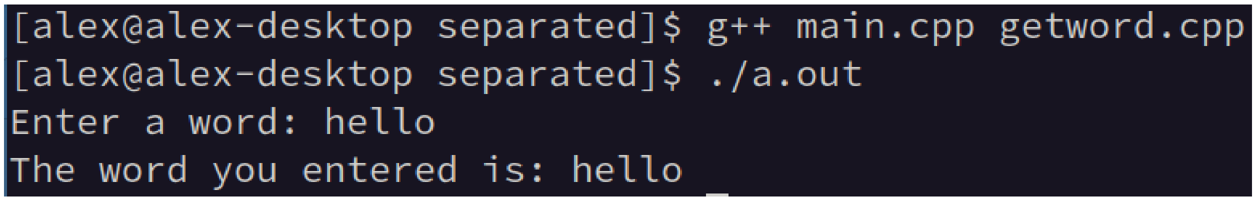
In this case, `getword.cpp` does not have to include `getword.h`. But when we introduce structs in a moment, the story will change.

Question: How do we build and run our program now that we have two separate .cpp files? Well, we have to <u>preprocess</u> and <u>compile</u> both of them into object code, and then <u>link</u> their object code together. To do this all in one command, simply list all of your .cpp files when running g++. For example:

```
g++ main.cpp getword.cpp -o <name_of_executable>
```

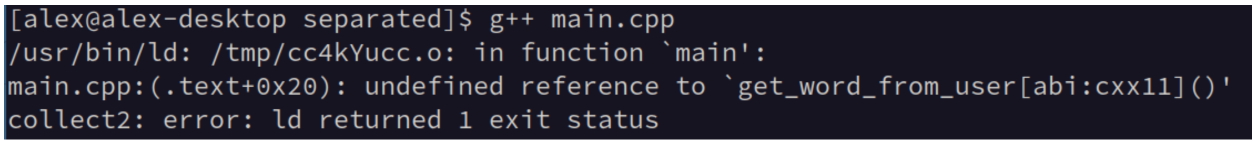
As always, you can leave out the `-o <name of executable>`, and it will automatically name your executable "a.exe" or "a.out", depending on your platform.

Here's what it looks like in the terminal:



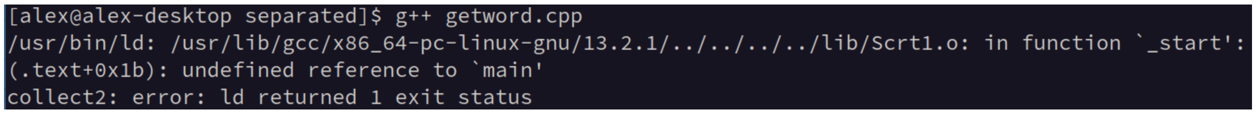
```
[alex@alex-desktop separated]$ g++ main.cpp getword.cpp
[alex@alex-desktop separated]$ ./a.out
Enter a word: hello
The word you entered is: hello
```

Important: If you only list `main.cpp` in your g++ command, you'll get an error saying that the get_word_from_user() function was referenced (used) but not defined. This is because it's defined in `getword.cpp`, and you forgot to link in the definition. For example:



```
[alex@alex-desktop separated]$ g++ main.cpp
/usr/bin/ld: /tmp/cc4kYucc.o: in function `main':
main.cpp:(.text+0x20): undefined reference to `get_word_from_user[abi:cxx11]()'
collect2: error: ld returned 1 exit status
```

Similarly, if you only list `getword.cpp` in your g++ command, you'll get an error saying that the main() function was referenced (used) but not defined. In this case, main() is implicitly referenced because it's the entrypoint of the program—every program must define main() exactly once in exactly one .cpp file. For example:



```
[alex@alex-desktop separated]$ g++ getword.cpp
/usr/bin/ld: /usr/lib/gcc/x86_64-pc-linux-gnu/13.2.1/../../../../lib/Scrt1.o: in function `_start':
(.text+0x1b): undefined reference to `main'
collect2: error: ld returned 1 exit status
```

Header files

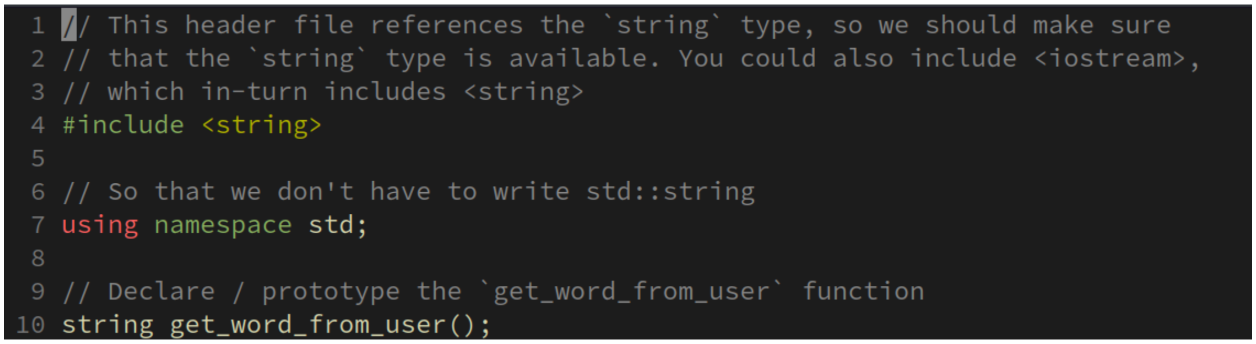
So we've learned how to break our code up a little bit, but there's a big problem: suppose we have 20 .cpp files and 10 functions. Suppose each of those 20 .cpp files needs to use / call every one of those 10 functions. Then, in each of our 20 .cpp files, we need to declare / prototype all 10 of those functions! That's a lot of copying / pasting…

Luckily, there's a tool that we can use to resolve this problem: the preprocessor. Specifically, the #include directive locates a file and copies / pastes its contents directly into your source code. So far, we've only used #include for system libraries (e.g., `<iostream>`)… But what if we could use it to include our own files? Then, we could…

1. Declare / prototype all 10 of our functions in a single file
2. #include that file in each of our 20 .cpp files

Indeed, we can do this. A file that contains prototypes and is meant to be included by other files (i.e., the single file in step 1 above) is called a header file, and we usually suffix them with .h (or .hpp) instead of .cpp. To use the #include directive to include your own header file (copy / paste its contents directly into your source code), you just have to use double quotes instead of angle brackets, and you have to specify the whole header file path (e.g., `#include "my_header.h"`, if `my_header.h` is present in the working directory).

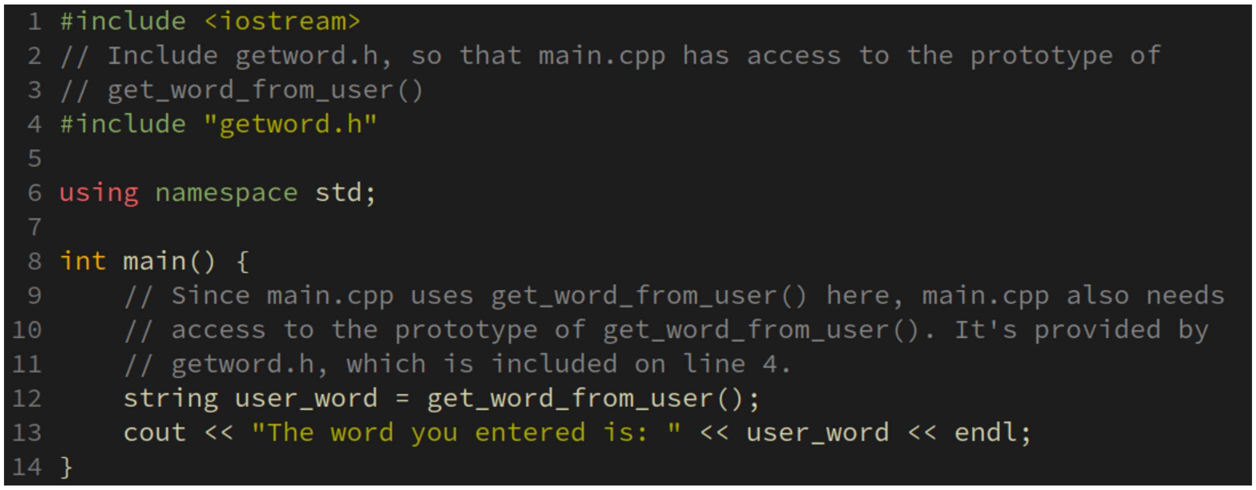
Let's modify our previous example and move the function prototype into its own header file, and then #include that header file within main.cpp. First, here's our header file, `getword.h`:



```cpp
// This header file references the `string` type, so we should make sure
// that the `string` type is available. You could also include <iostream>,
// which in-turn includes <string>
#include <string>

// So that we don't have to write std::string
using namespace std;

// Declare / prototype the `get_word_from_user` function
string get_word_from_user();
```

Notice that our header file includes `<string>` and dumps the standard namespace. That's important since the header file makes use of strings. (yes, header files can include other header files, and so on…)

Now, here's our `main.cpp`. Notice that we got rid of the `string get_word_from_user()` prototype and simply included the header file instead:



```cpp
 1 #include <iostream>
 2 // Include getword.h, so that main.cpp has access to the prototype of
 3 // get_word_from_user()
 4 #include "getword.h"
 5 
 6 using namespace std;
 7 
 8 int main() {
 9     // Since main.cpp uses get_word_from_user() here, main.cpp also needs
10     // access to the prototype of get_word_from_user(). It's provided by
11     // getword.h, which is included on line 4.
12     string user_word = get_word_from_user();
13     cout << "The word you entered is: " << user_word << endl;
14 }
```

Note: `getword.cpp` is unchanged—scroll up to the previous example to see what it looks like.

Common convention: for every XXX.h file you have that declares / prototypes one or more functions, you should have a corresponding XXX.cpp file that defines all of those functions. In this case, we have `getword.h`, which declares `get_word_from_user()`, and `getword.cpp`, which defines `get_word_from_user()`.

Now, how do we build and run our program? Answer: exactly as before:

```
g++ main.cpp getword.cpp -o <name_of_executable>
```

Important: Do not list header files when building your program via g++! You should only list your .cpp files. If you list header files, then it will generate files referred to as "precompiled headers", which have a .gch extension. If precompiled header files are present, your corresponding header files themselves will be ignored. This can cause problems. If you accidentally compile a header file into a precompiled header file, you can safely delete the precompiled header file to resolve the issue (but be careful not to delete your header file itself!)

Comments in header files

For functions that are prototyped in a header file, you should write their function header comments immediately above their prototypes in the header file. You do not need to copy / paste those function header comments in the implementation (.cpp) file next to the function's definition, but the implementation file should still contain in-line comments explaining code where appropriate.

For functions that are defined in an implementation file but not prototyped in any header files (e.g., "internal" functions that are only accessible by other functions within the implementation file), you should write their function header comments immediately above their definitions in the implementation file. Again, they should also contain in-line comments explaining code where appropriate.

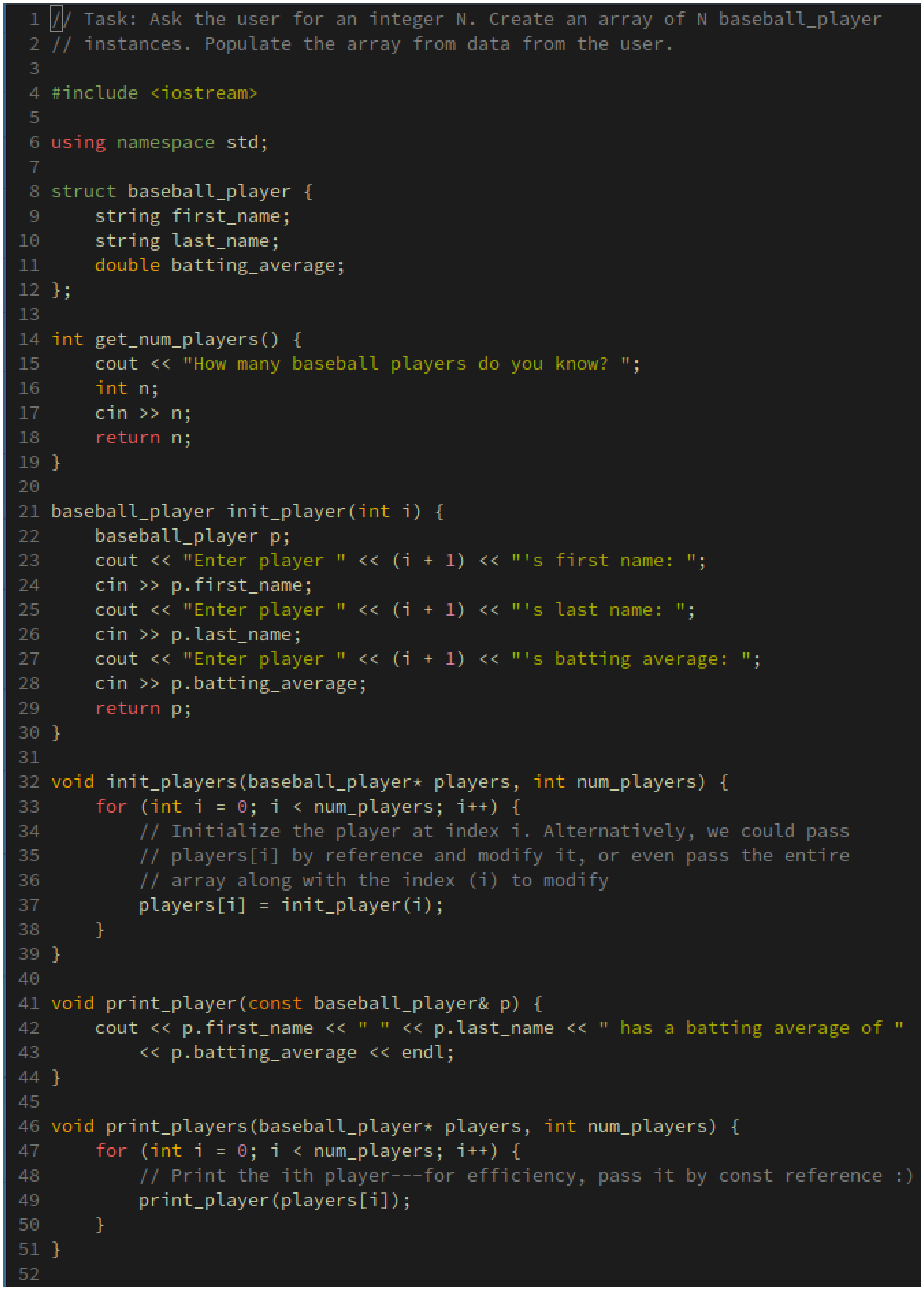
Structs in header files

Now, let's introduce one more bullet point criterion related to structs:

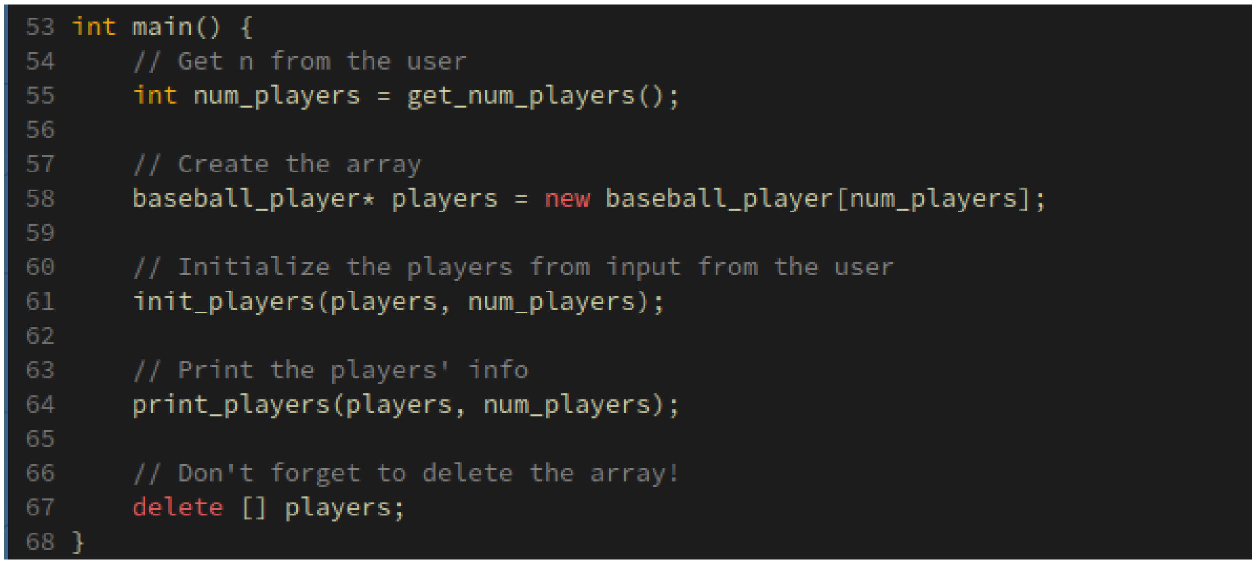
- (**Criterion 3**) Every struct type that a .cpp file uses must be defined within that .cpp file before it can be used.

Notice that criterion 3 is just like criterion 2, but instead of declaring / prototyping functions, we need to define structure types. Hence, the solution to this criterion is the same—define your structure type in a header file, and then include that header file in every .cpp file that needs to use the structure type.

Recall that in the structs lecture we worked with a demo that constructed a 1D dynamic array of `baseball_player` instances. Here's what it looked like all in one file (two screenshots):



```cpp
// Task: Ask the user for an integer N. Create an array of N baseball_player
// instances. Populate the array from data from the user.

#include <iostream>

using namespace std;

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

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

baseball_player init_player(int i) {
    baseball_player p;
    cout << "Enter player " << (i + 1) << "'s first name: ";
    cin >> p.first_name;
    cout << "Enter player " << (i + 1) << "'s last name: ";
    cin >> p.last_name;
    cout << "Enter 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 player at index i. Alternatively, we could pass
        // players[i] by reference and modify it, or even pass the entire
        // array along with the index (i) to modify
        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 it by const reference :)
        print_player(players[i]);
    }
}

```

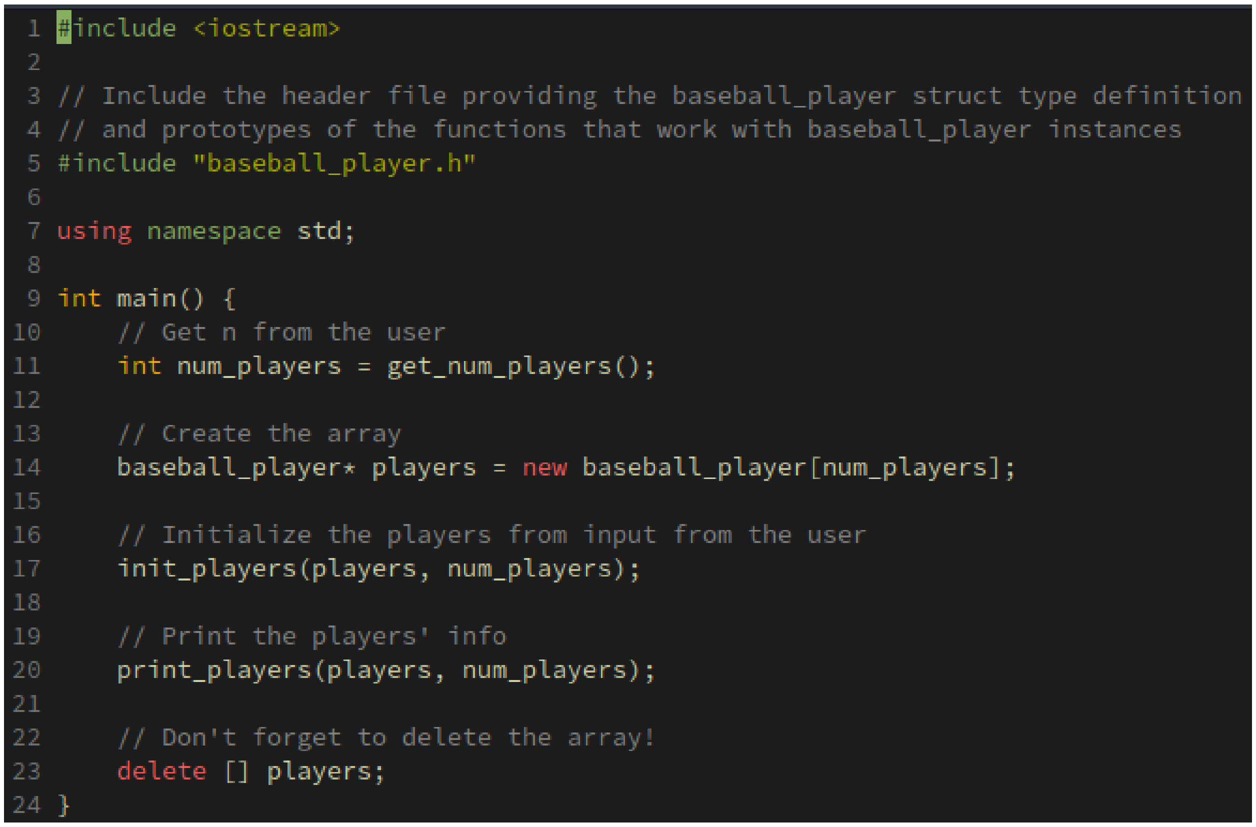

```cpp
int main() {
    // Get n from the user
    int num_players = get_num_players();

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

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

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

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

Suppose we want to break up that demo into several files. Let's create three files:

1. `main.cpp`, which will define our main() function
2. `baseball_player.h`, which will define the `baseball_player` structure type and declare functions that work with baseball players
3. `baseball_player.cpp`, which will define the functions that work with baseball players (i.e., the functions declared in `baseball_player.h`)

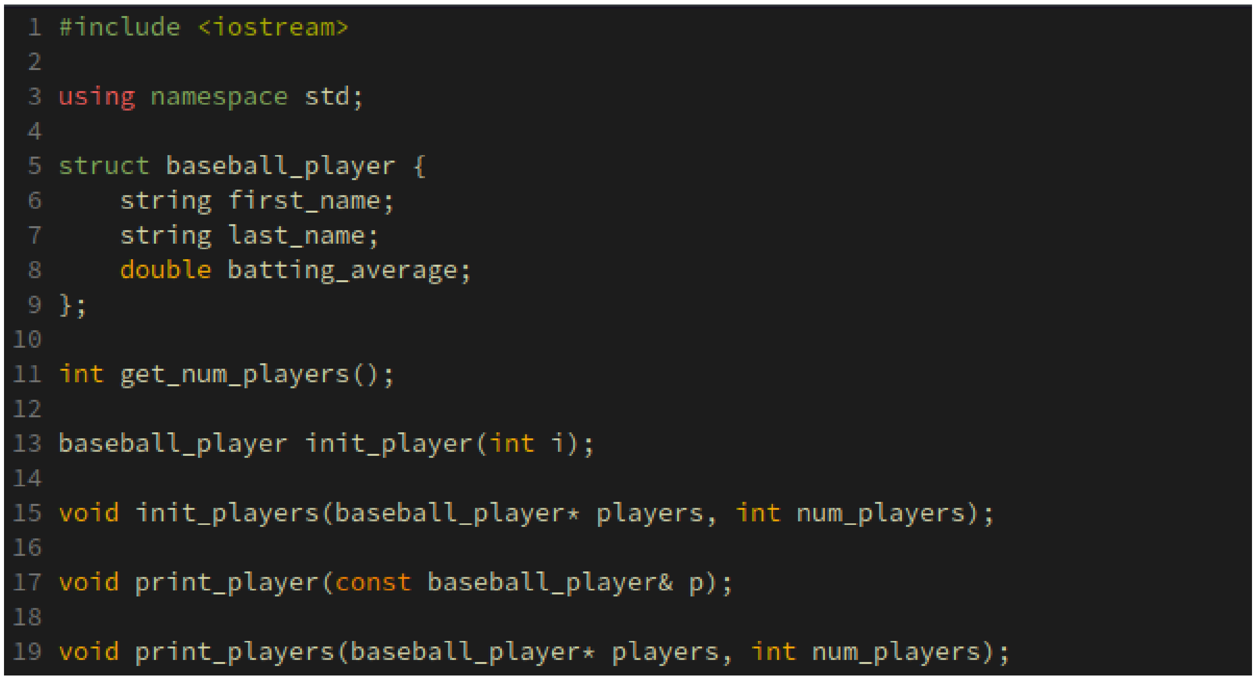
Here's what that would look like:

`main.cpp`:



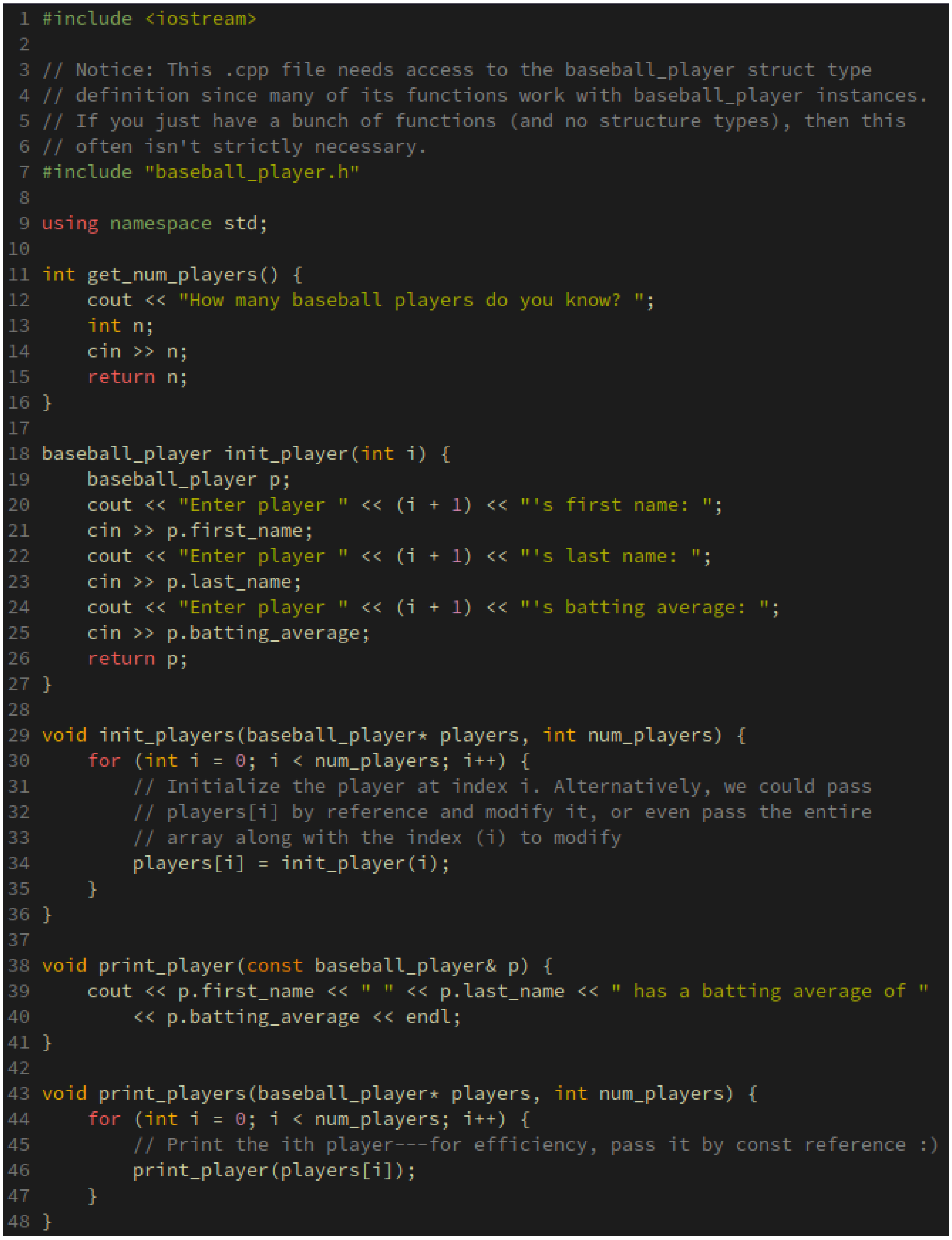
```cpp
#include <iostream>

// Include the header file providing the baseball_player struct type definition
// and prototypes of the functions that work with baseball_player instances
#include "baseball_player.h"

using namespace std;

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

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

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

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

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

baseball_player.h:



```cpp
#include <iostream>

using namespace std;

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

int get_num_players();

baseball_player init_player(int i);

void init_players(baseball_player* players, int num_players);

void print_player(const baseball_player& p);

void print_players(baseball_player* players, int num_players);
```

baseball_player.cpp:



```cpp
#include <iostream>

// Notice: This .cpp file needs access to the baseball_player struct type
// definition since many of its functions work with baseball_player instances.
// If you just have a bunch of functions (and no structure types), then this
// often isn't strictly necessary.
#include "baseball_player.h"

using namespace std;

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

baseball_player init_player(int i) {
    baseball_player p;
    cout << "Enter player " << (i + 1) << "'s first name: ";
    cin >> p.first_name;
    cout << "Enter player " << (i + 1) << "'s last name: ";
    cin >> p.last_name;
    cout << "Enter 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 player at index i. Alternatively, we could pass
        // players[i] by reference and modify it, or even pass the entire
        // array along with the index (i) to modify
        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 it by const reference :)
        print_player(players[i]);
    }
}
```

Again, we could build this program like so:

```
g++ main.cpp baseball_player.cpp -o <name_of_executable>
```

Multiple includes & header guards

Suppose a header file gets included twice in one .cpp file. Then you'll likely end up with:

- duplicate function prototypes
- duplicate structure type definitions

The former is not a problem, but the latter is—it will result in a compilation error.

How do we avoid this? One simple idea is "don't include the same header file twice in one .cpp file". Unfortunately, it's not that simple. Consider:

a.cpp includes b.h and c.h

b.h includes d.h

c.h includes d.h

Therefore, a.cpp implicitly includes d.h twice, and there's no obvious way around that. Indeed, this will result in a compiler error if d.h defines a structure type, as per the second bullet point above.

How do we fix this? Once again, we make use of the preprocessor. #include is just one preprocessor directive. Let's learn about a few more:

- #define <constant name>
    - Defines a preprocessor constant called <constant name>. You can also give values to these preprocessor constants, but we don't need to do that for our purposes.
    - Convention: <constant name> should be snake case and all caps. E.g., MY_CONSTANT
- #ifdef <constant name>
    - Begins a preprocessor if statement, conditioned on the existence of a preprocessor constant called <constant name>. If the preprocessor constant <constant name> has not yet been defined, then all of the code contained within the preprocessor if statement (up to the corresponding #endif) will be ignored by your compiler
- #endif
    - Ends a preprocessor if statement
- #ifndef <constant name>

- The opposite of #ifdef. Begins a preprocessor if statement, conditioned on the non-existence of a preprocessor constant called <constant name>. If the preprocessor constant <constant name> has been defined, then all of the code contained within the preprocessor if statement (up to the corresponding #endif) will be ignored by your compiler.

Since each .cpp file is preprocessed and compiled separately, each gets its own preprocessor constants. We can use these preprocessor directives to solve the problem of multiple includes. In particular, the problem is solved if you **put these two lines at the beginning of every header file you ever write**:

```
#ifndef SOME_CONSTANT_H
#define SOME_CONSTANT_H
```

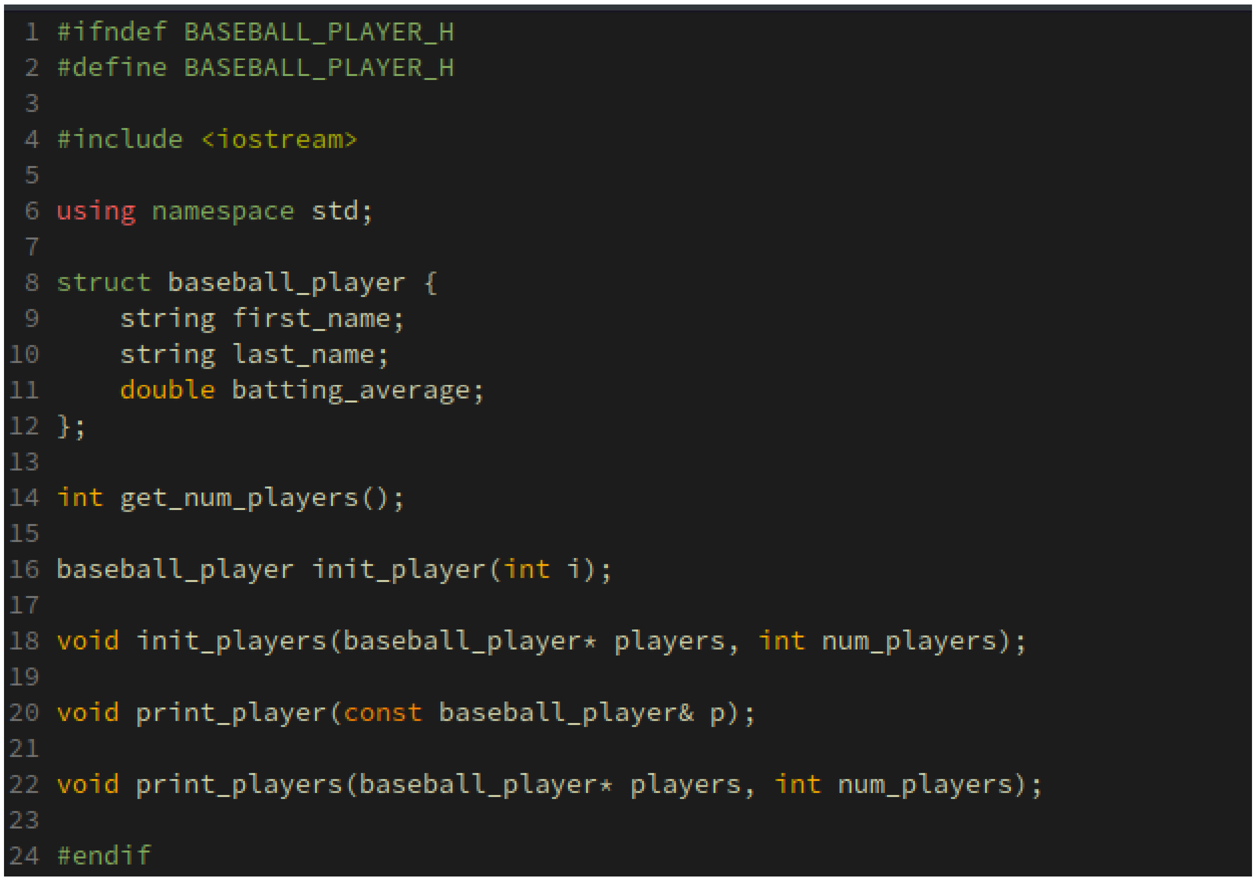
where SOME_CONSTANT_NAME is unique to the header file, **and put this line at the end of every header file you ever write**:

```
#endif
```

Together, these three preprocessor directives are referred to as header guards because they prevent your header file from being included multiple times.

Common convention: Name the header guard preprocessor constant after the header file itself. If your header file is called my_header_file.h, then you could call the preprocessor constant in the header guards MY_HEADER_FILE_H.

For example, we could update the header file from our baseball player demo, like so:



```
 1 #ifndef BASEBALL_PLAYER_H
 2 #define BASEBALL_PLAYER_H
 3
 4 #include <iostream>
 5
 6 using namespace std;
 7
 8 struct baseball_player {
 9     string first_name;
10     string last_name;
11     double batting_average;
12 };
13
14 int get_num_players();
15
16 baseball_player init_player(int i);
17
18 void init_players(baseball_player* players, int num_players);
19
20 void print_player(const baseball_player& p);
21
22 void print_players(baseball_player* players, int num_players);
23
24 #endif
```

Again—lines 1, 2, and 24 combined are referred to as header guards.

Why does this work? Well, suppose we *try* to include baseball_player.h twice within main.cpp.

1. The first time main.cpp includes baseball_player.h, BASEBALL_PLAYER_H does not yet exist—it has not yet been defined as a preprocessor constant. This results in a true condition for the preprocessor if statement on line 1 of baseball_player.h above. Therefore, the compiler and preprocessor proceed to acknowledge lines 1-24 (the entire header file), and it will be included like normal. Immediately, at line 2, BASEBALL_PLAYER_H is defined.
2. The second time (and all subsequent times) main.cpp tries to include baseball_player.h, BASEBALL_PLAYER_H *has* been defined, due to line 2 of baseball_player.h from the first time it was included in step 1 above. This results in a false condition for the preprocessor if statement on line 1 in baseball_player.h, so the compiler and preprocessor proceed to ignore all of the code between lines 1-24 (the entire header file).

Function header comments

From now on, you should write your function header comments directly above the respective function prototype in its header file. No need to copy them in the implementation (.cpp) file.

The implementation (.cpp) file should still contain in-line comments where appropriate. For assignments in this course, both the implementation (.cpp) and header (.h) files should contain file header comments (author, description, etc).

Summary

1. Put structure type definitions in header files.
2. Put function prototypes (declarations) in header files
3. For every XXX.h header file that declares a function, define the function in a corresponding XXX.cpp
4. Whenever a .cpp / .h file needs access to function declarations / structure type definitions from some header file, #include that header file
5. Make sure all of your header files have header guards (#ifndef, #define, #endif)
6. Compile all of your .cpp files together to produce an executable
    a. And don't directly compile your .h files! Compile / build the .cpp files, which in-turn #include .h files.