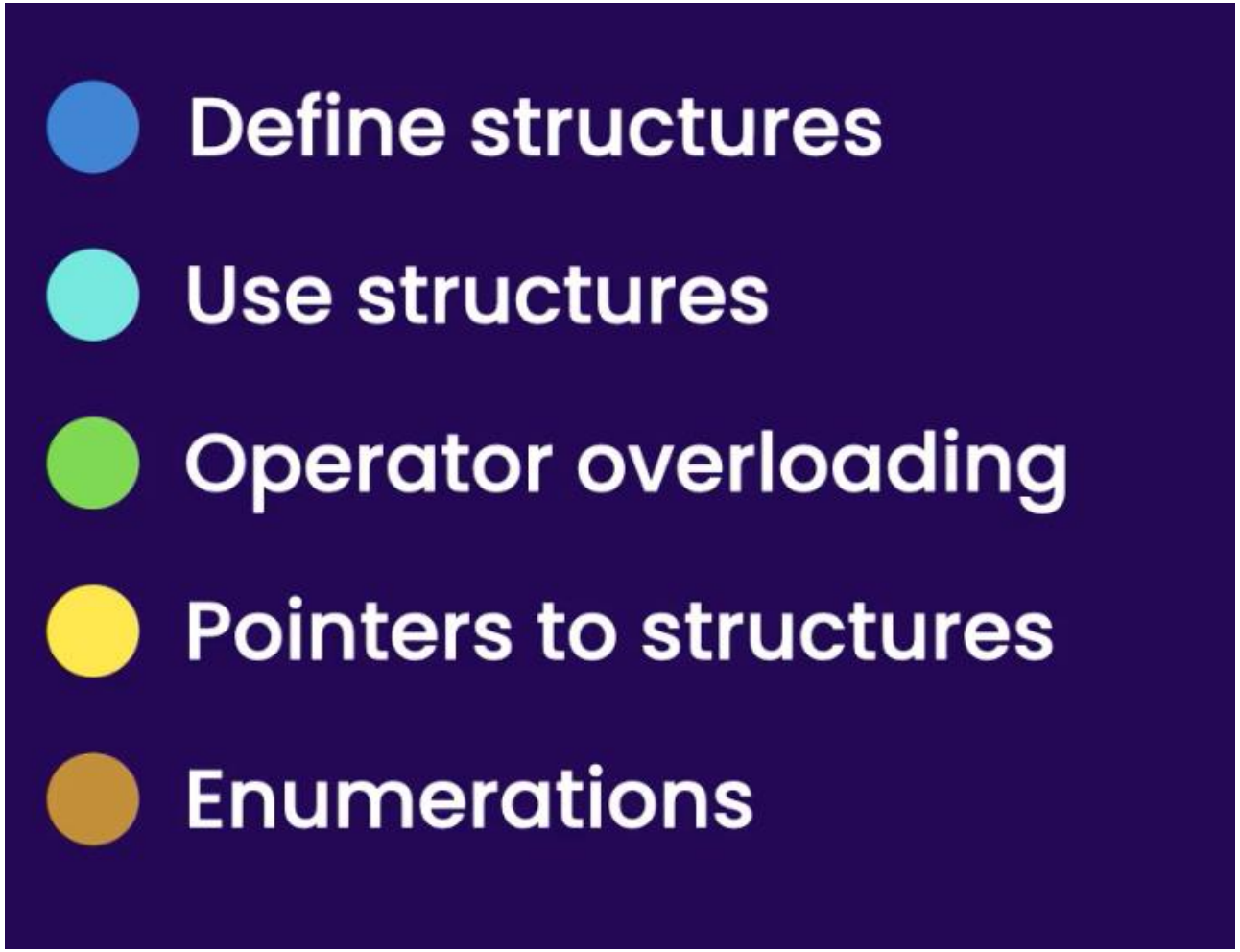
C++ Notes

Part 2-2

Slides were created from CodewithMosh.com

Structures and Enumerations



Defining Structures

```cpp
#include <iostream>

using namespace std;

// PascalCase
struct Movie {
    string title;
    int releaseYear;
};
int main() {
    Movie movie;
    movie.title = "Terminator";
    movie.releaseYear = 1984;
    cout << "Title: " << movie.title << endl
         << "Release Year: " << movie.releaseYear;

    return 0;
}
```

- With structures, we can create abstract data types (ADT).
- **Abstraction:** A general model for something.
- A custom data type.
- **PascalCase** naming convention: Capitalize the first letter of each word.
- Variable movie is an object (an instance of type Movie).

Initializing Structures

```cpp
#include <iostream>

using namespace std;

struct Movie {
    string title;
    int releaseYear = 0;
    bool isPopular;
};

int main() {
    Movie movie = { .title: "Terminator", .releaseYear: 1984 };
    cout << movie.releaseYear;

    return 0;
};
```

- We can give default values to structure members.
- String variables by default are empty.
- Boolean variables by default are false.

Unpacking Structures

```cpp
#include <iostream>

using namespace std;

int main() {
    Movie movie = { .title: "Terminator", .releaseYear: 1984 };
    string title = movie.title;
    int releaseYear = movie.releaseYear;
    bool isPopular = movie.isPopular;
    // C++: structured binding
    // JS: destructuring
    // Python: unpacking

};
```

- We can give default values to structure members.
- String variables by default are empty.
- Boolean variables by default are false.

Replace with unpacking method

```cpp
auto [title, releaseYear, isPopular] { movie };
```

```cpp
#include <iostream>

using namespace std;

struct Movie {
    string title;
    int releaseYear = 0;
    bool isPopular;
};

int main() {
    // vector
    Movie movies[5];

    return 0;
};
```

Array of Structures

- We can create arrays of structure type.
- Using **vector** type is more efficient.

Array of Structures

- We can create arrays of structure type.
- Using **vector** type is more efficient.
- Memory management is handled automatically in vectors.

```cpp
#include <iostream>
#include <vector>

using namespace std;

struct Movie {
    string title;
    int releaseYear = 0;
    bool isPopular;
};

int main() {
    vector<Movie> movies;
    Movie movie { .title: "Terminator", .releaseYear: 1984 };
    movies.push_back(movie);

    return 0;
}
```

Replace

```cpp
int main() {
    vector<Movie> movies;
    movies.push_back({ .title: "Terminator 1", .releaseYear: 1984
    movies.push_back({ .title: "Terminator 2", .releaseYear: 1987

    for (auto movie : Movie : movies)
        cout << movie.title << endl;

    return 0;
}
```

Array of Structures

- We can create arrays of structure type.
- Using **vector** type is more efficient.

```cpp
#include <iostream>
#include <vector>

using namespace std;

struct Movie {
    string title;
    int releaseYear = 0;
    bool isPopular;
};

int main() {
    vector<Movie> movies;
    movies.push_back({ .title: "Terminator 1", .releaseYear: 1984
    movies.push_back({ .title: "Terminator 2", .releaseYear: 1987

    for (auto movie : Movie : movies)
        cout << movie.title << endl;

    return 0;
}
```

Replace using reference variable

```cpp
for (const auto& movie : const Movie & : movies)
    cout << movie.title << endl;
```

Nesting Structures

```
struct Movie {
    string title;
    int releaseYear = 0;
    int releaseMonth = 0;
    int releaseDay = 0;
    bool isPopular;
};
```


```
struct Date {
    int year = 1900;
    short month = 1;
    short day = 1;
};
struct Movie {
    string title;
    Date releaseDate;
    bool isPopular;
};
```

Nesting Structures

```
int main() {
    Date date { .year: 1984, .month: 6, .day: 1 };
    Movie movie { .title: "Terminator", .releaseDate: date };
    cout << movie.releaseDate.year;

    return 0;
}
```

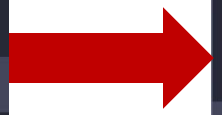
Equivalent

```
int main() {
    Movie movie {
        .title: "Terminator",
        .releaseDate: { .year: 1984, .month: 6, .day: 1 }
    };
    cout << movie.releaseDate.year;

    return 0;
}
```

Comparing Structures

- We need to compare each element of the vectors to test for equality.
- This method is cumbersome and not efficient.
- A better way is to use methods.

```cpp
#include <iostream>
#include <vector>

using namespace std;

struct Date {
    int year = 1900;
    short month = 1;
    short day = 1;
};

struct Movie {
    string title;
    Date releaseDate;
    bool isPopular;
};

int main() {
    Movie movie1 = {
        .title: "Terminator",
        .releaseDate: { .year: 1984, .month: 6, .day: 1 }
    };

    Movie movie2 = {
        .title: "Terminator",
        .releaseDate: { .year: 1984, .month: 6, .day: 1 }
    };
```



```cpp
if (movie1.title == movie2.title &&
    movie1.releaseDate.year == movie2.releaseDate.year &&
    movie1.releaseDate.month == movie2.releaseDate.month &&
    movie1.releaseDate.day == movie2.releaseDate.day
)
    cout << "Equal";

    return 0;
}
```

Working with Methods

- A better way is to use methods.

```cpp
#include <iostream>
#include <vector>

using namespace std;

struct Date {
    int year = 1900;
    short month = 1;
    short day = 1;
};
```

```cpp
struct Movie {
    string title;
    int releaseYear = 0;
    bool isPopular;

    bool equals(Movie movie) {
        return (
            title == movie.title &&
            releaseDate.year == movie.releaseDate.year &&
            releaseDate.month == movie.releaseDate.month &&
            releaseDate.day == movie.releaseDate.day
        );
    }
};
```

Method of Movie Structure

```cpp
int main() {
    Movie movie1 = {
        .title: "Terminator",
        .releaseDate: { .year: 1984, .month: 6, .day: 1 }
    };

    Movie movie2 = {
        .title: "Terminator",
        .releaseDate: { .year: 1984, .month: 6, .day: 1 }
    };

    if (movie1.equals( movie: movie2))
        cout << "Equal";

    return 0;
}
```

Working with Methods

- A better way is to use methods.

```cpp
#include <iostream>
#include <vector>

using namespace std;

struct Date {
    int year = 1900;
    short month = 1;
    short day = 1;
};
```

```cpp
struct Movie {
    string title;
    int releaseYear = 0;
    bool isPopular;

    bool equals(const Movie& movie) {
        return (
            title == movie.title &&
            releaseDate.year == movie.releaseDate.year &&
            releaseDate.month == movie.releaseDate.month &&
            releaseDate.day == movie.releaseDate.day
        );
    }
};
```

Method of Movie Structure

```cpp
int main() {
    Movie movie1 = {
        .title: "Terminator",
        .releaseDate: { .year: 1984, .month: 6, .day: 1 }
    };

    Movie movie2 = {
        .title: "Terminator",
        .releaseDate: { .year: 1984, .month: 6, .day: 1 }
    };

    if (movie1.equals( movie: movie2))
        cout << "Equal";

    return 0;
}
```

Operator Overloading

- Create a comparison (==) method.

```cpp
#include <iostream>
#include <vector>

using namespace std;

struct Date {
    int year = 1900;
    short month = 1;
    short day = 1;
};

struct Movie {
    string title;
    Date releaseDate;
    bool isPopular;
};

    bool equals(const Movie& movie) {
        return (
            title == movie.title &&
            releaseDate.year == movie.releaseDate.year &&
            releaseDate.month == movie.releaseDate.month &&
            releaseDate.day == movie.releaseDate.day
        );
    }
};

    bool operator==(const Movie& movie) const {
        return (
            title == movie.title &&
            releaseDate.year == movie.releaseDate.year &&
            releaseDate.month == movie.releaseDate.month &&
            releaseDate.day == movie.releaseDate.day
        );
    }
};
```

```cpp
int main() {
    Movie movie1 = {
    .title: "Terminator",
    .releaseDate: { .year: 1984, .month: 6, .day: 1 }
    };

    Movie movie2 = {
    .title: "Terminator",
    .releaseDate: { .year: 1984, .month: 6, .day: 1 }
    };

    if (movie1 == movie2)
        cout << "Equal";

    return 0;
}
```

Operator Overloading

- Creating a function == outside of the structure.
- **Restriction:** Some operator overloading should be implemented outside of structures.
- It is a good practice to implement all operator overloading outside of structures.

```cpp
#include <iostream>
#include <vector>

using namespace std;
```

```cpp
struct Date {
    int year = 1900;
    short month = 1;
    short day = 1;
};
```

```cpp
struct Movie {
    string title;
    Date releaseDate;
    bool isPopular;
};
```

Not a method of Movie Structure anymore

```cpp
bool operator==(const Movie& first, const Movie& second) {
    return (
            first.title == second.title &&
            first.releaseDate.year == second.releaseDate.year &&
            first.releaseDate.month == second.releaseDate.month &&
            first.releaseDate.day == second.releaseDate.day
    );
}
```

```cpp
int main() {
    Movie movie1 = {
        .title: "Terminator",
        .releaseDate: { .year: 1984, .month: 6, .day: 1 }
    };

    Movie movie2 = {
        .title: "Terminator",
        .releaseDate: { .year: 1984, .month: 6, .day: 1 }
    };

    if (movie1 == movie2)
        cout << "Equal";

    return 0;
}
```

Operator Overloading

- Implementation of << operator overloading.

```cpp
#include <iostream>
#include <vector>

using namespace std;

struct Date {
    int year = 1900;
    short month = 1;
    short day = 1;
};

struct Movie {
    string title;
    Date releaseDate;
    bool isPopular;
};
```

```cpp
ostream& operator<<(ostream& stream, const Movie& movie) {
    stream << movie.title;
    return stream;
}
```

```cpp
int main() {
    Movie movie1 = {
    .title: "Terminator",
    .releaseDate: { .year: 1984, .month: 6, .day: 1 }
    };

    Movie movie2 = {
    .title: "Terminator",
    .releaseDate: { .year: 1984, .month: 6, .day: 1 }
    };

    cout << movie1;

    return 0;
}
```

Structures and Functions

```cpp
#include <iostream>
#include <vector>

using namespace std;

struct Date {
    int year = 1900;
    short month = 1;
    short day = 1;
};

struct Movie {
    string title;
    Date releaseDate;
    bool isPopular;
};
```

```cpp
ostream& operator<<(ostream& stream, const Movie& movie) {
    stream << movie.title;
    return stream;
}
```

```cpp
Movie getMovie() {
    return { .title: "Terminator", .releaseDate.year: 1984 };;
}

void showMovie(Movie& movie) {
    cout << movie.title;
}

int main() {
    auto movie : Movie = getMovie();
    showMovie( &: movie);

    return 0;
}
```

Pointers to Structures

- When we pass objects to functions, it is safer to use reference parameters.

```cpp
#include <iostream>
#include <vector>

using namespace std;

struct Date {
    int year = 1900;
    short month = 1;
    short day = 1;
};

struct Movie {
    string title;
    Date releaseDate;
    bool isPopular;
};
```

```cpp
ostream& operator<<(ostream& stream, const Movie& movie) {
    stream << movie.title;
    return stream;
}
Movie getMovie() {
    return { .title: "Terminator", .releaseDate.year: 1984 };;
}
```

```cpp
void showMovie(Movie* movie) {
    cout << movie->title;
}

int main() {
    auto movie : Movie = getMovie();
    showMovie( movie: &movie);

    return 0;
}
```

Equivalent (indirection operator)

```cpp
(*movie).title
```

Defining Enumerations

- Used to define a new custom data type that represents a group of related constants.
- Enum is internally represented using an integer.

```cpp
#include <iostream>
#include <vector>

using namespace std;

enum Action {
    List = 1,
    Add,
    Update
};

int main() {

    cout <<
        "1: List invoices" << endl <<
        "2: Add invoice" << endl <<
        "3: Update invoice" << endl <<
        "Select: ";
```

```cpp
    int input;
    cin >> input;

    if (input == Action::List) {
        cout << "List invoices";
    }

    return 0;
}
```

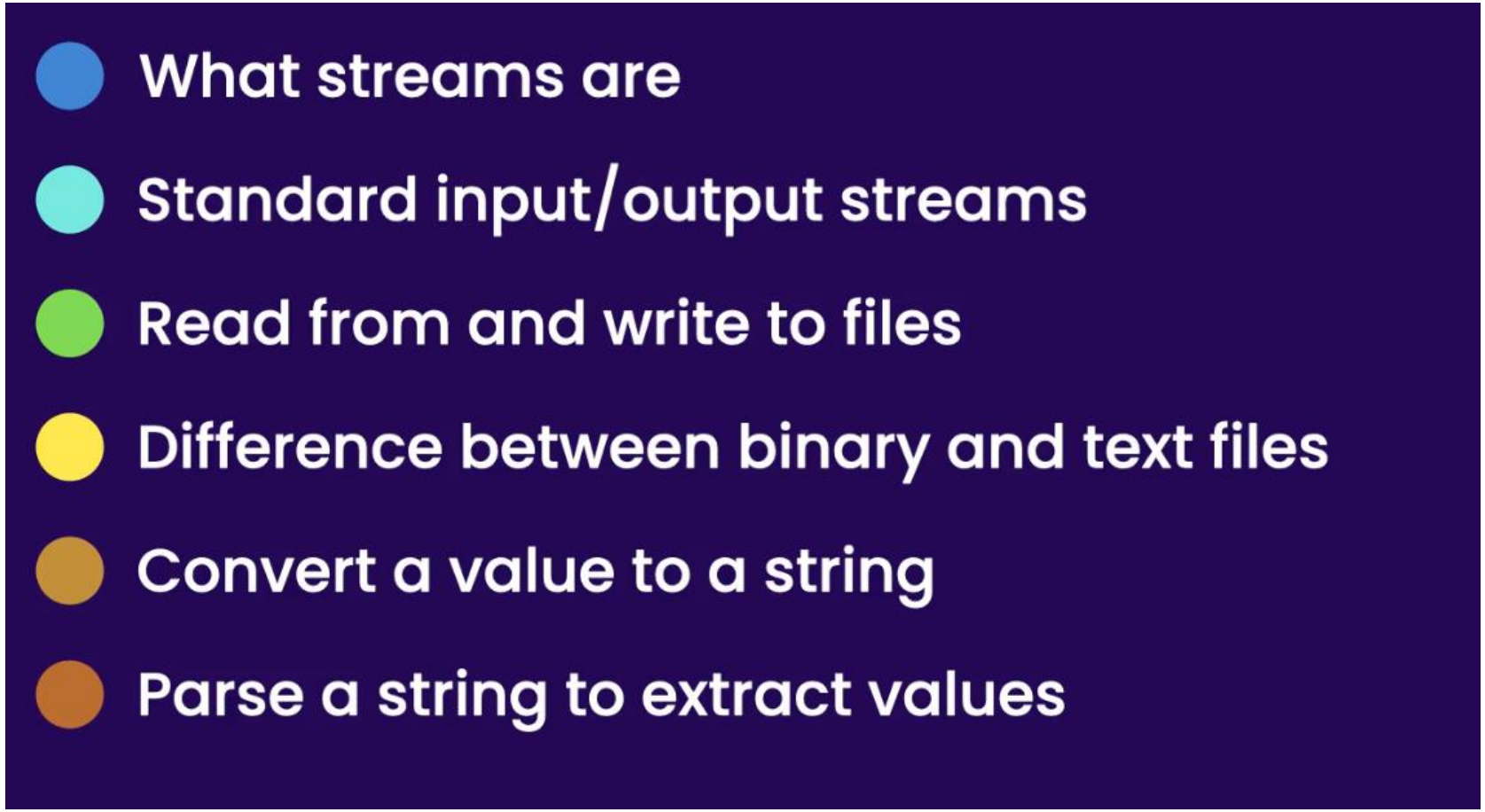
Strongly Typed Enumerations

- C++ 11 introduced strongly types enumerations.

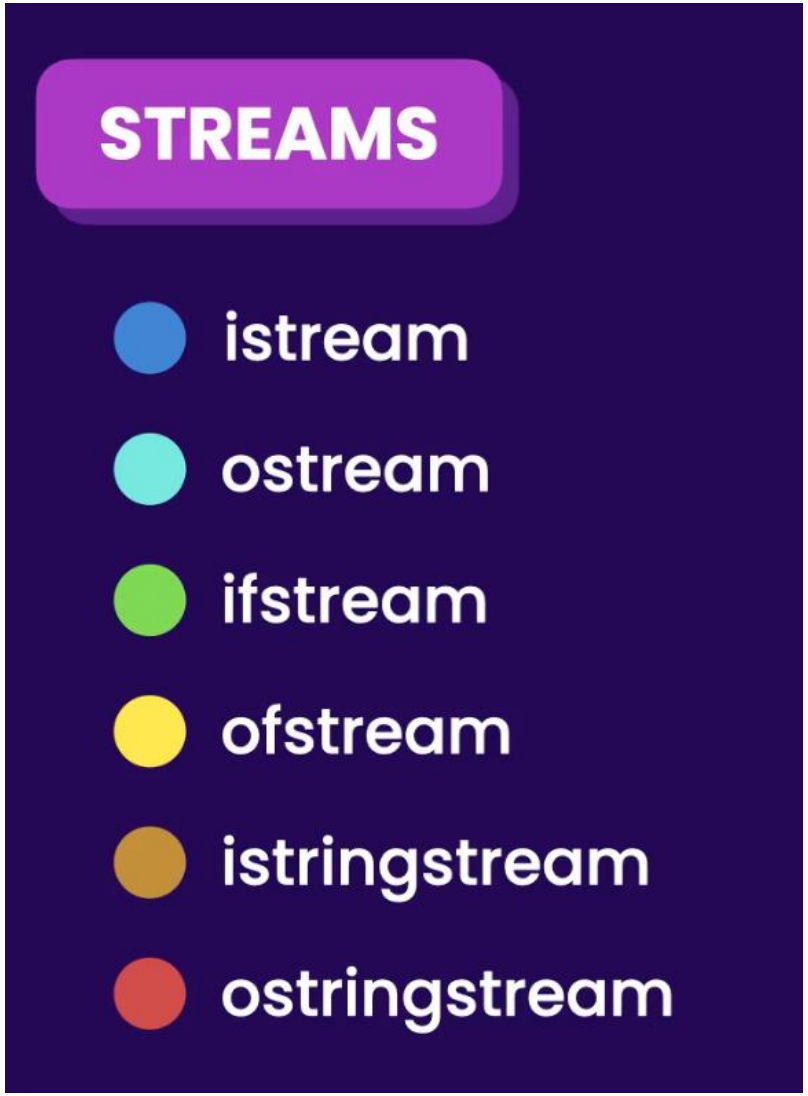
```cpp
#include <iostream>
#include <vector>

using namespace std;

enum class Action {
    list = 1,
    add,
    update
};

enum class Operation {
    list = 1,
    add,
    update
};
```

```cpp
int main() {

    cout <<
        "1: List invoices" << endl <<
        "2: Add invoice" << endl <<
        "3: Update invoice" << endl <<
        "Select: ";

    int input;
    cin >> input;

    if (input == static_cast<int>(Action::list)) {
        cout << "List invoices";
    }

    return 0;
}
```

Streams and Files

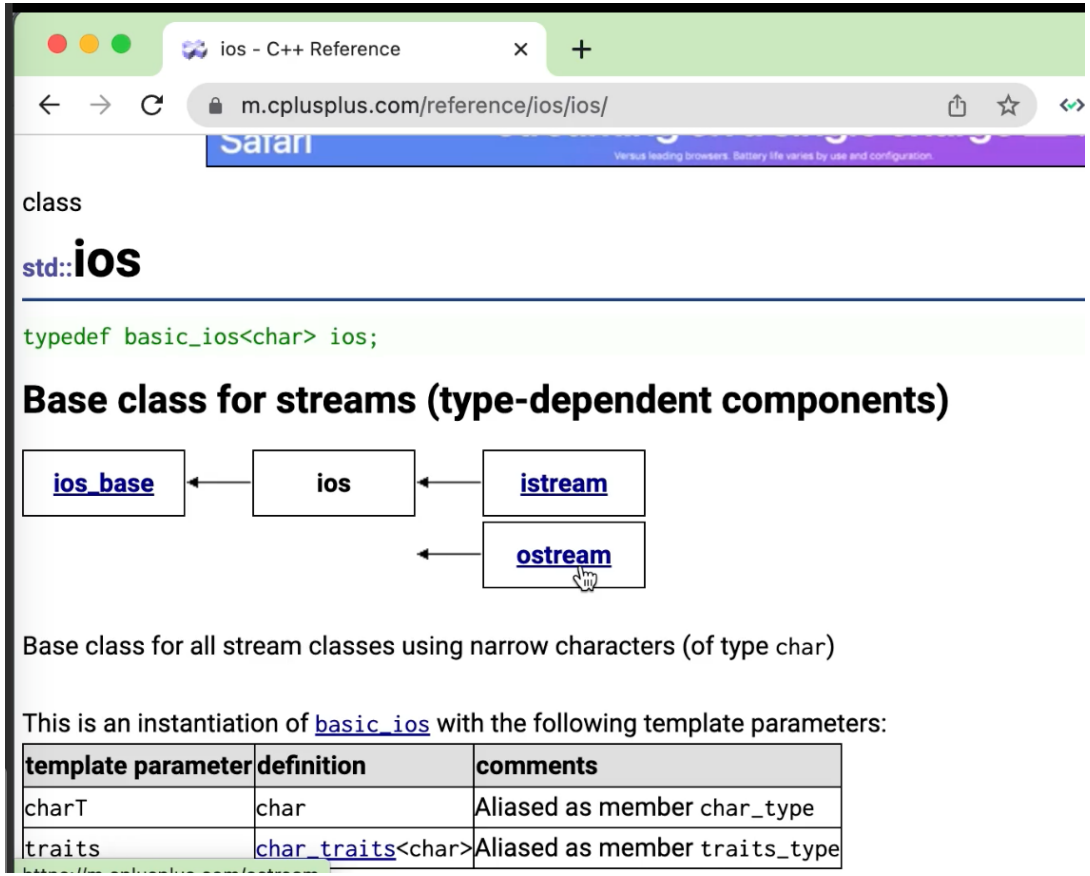


Understanding Streams

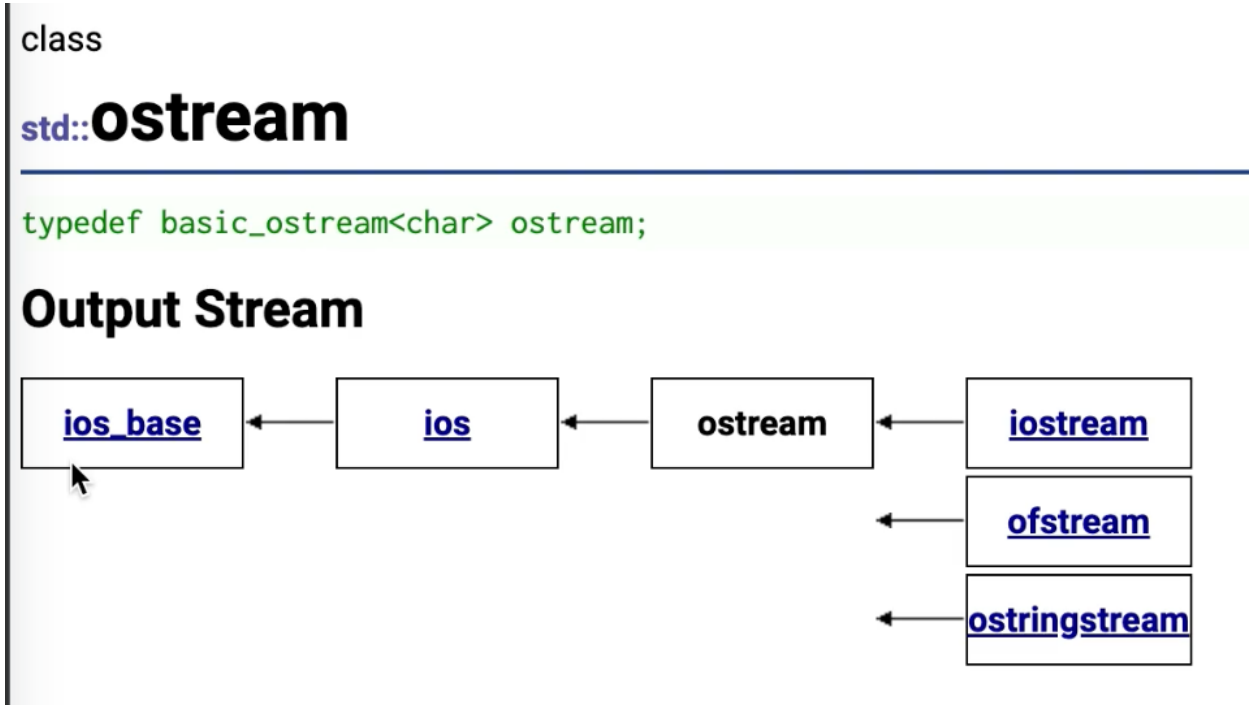


- We can think streams as data source or destination.
- Have the same interface and functions.

Understanding Streams



- We can think streams as data source or destination.



Writing to Streams

object

std::cout

<iostream>

```
extern ostream cout;
```

Standard output stream

Object of class ostream that represents the *standard output stream* oriented to narrow characters (of type char). It corresponds to the *C stream* stdout.

The *standard output stream* is the default destination of characters determined by the environment. This destination may be shared with more standard objects (such as cerr or clog).

As an object of class ostream, characters can be written to it either as formatted data using the insertion operator (operator<<) or as unformatted data using member functions such as write

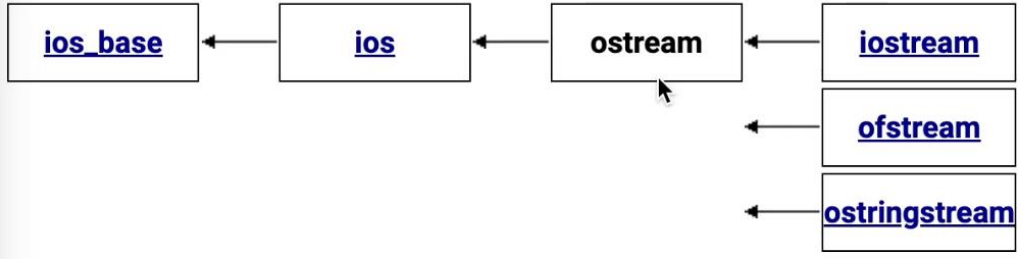
class

std::ostream

<ostream> <iostream

```
typedef basic_ostream<char> ostream;
```

Output Stream



Output stream objects can write sequences of characters and represent other kinds of data. Specific members are provided to perform these output operations (see functions below).

Writing to Streams

Public member functions

(constructor)	Construct object (public member function)
(destructor)	Destroy object (public member function)

Formatted output:

operator<<	Insert formatted output (public member function)

Unformatted output:

put	Put character (public member function)
write	Write block of data (public member function)

Positioning:

tellp	Get position in output sequence (public member function)
seekp	Set position in output sequence (public member function)

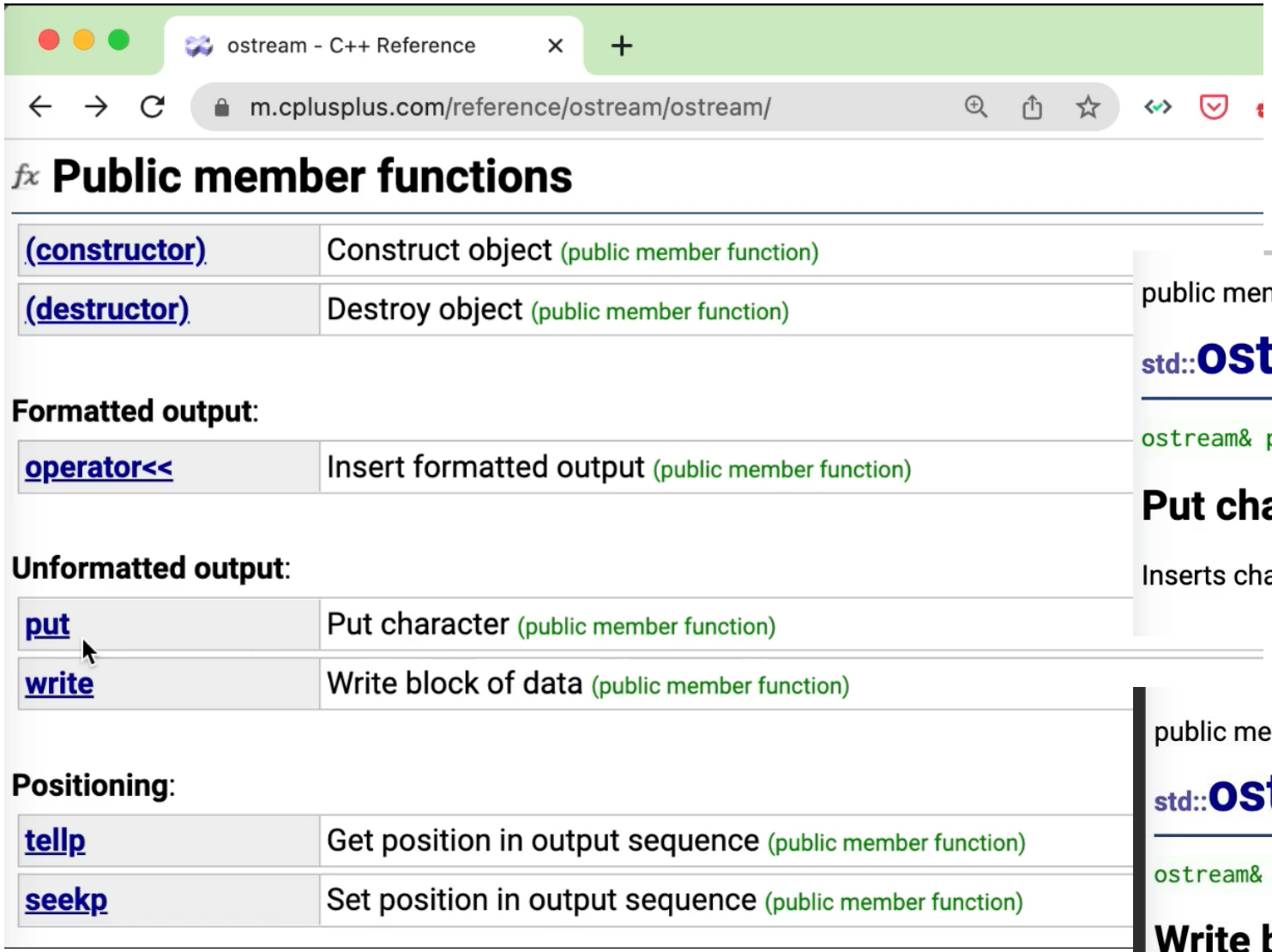
public member function

std::ostream::operator<<

C++98 C++11

```
                    ostream& operator<< (bool val);
                    ostream& operator<< (short val);
                    ostream& operator<< (unsigned short val);
                    ostream& operator<< (int val);
                    ostream& operator<< (unsigned int val);
                    ostream& operator<< (long val);
arithmetic types (1) ostream& operator<< (unsigned long val);
                    ostream& operator<< (long long val);
                    ostream& operator<< (unsigned long long val);
                    ostream& operator<< (float val);
                    ostream& operator<< (double val);
                    ostream& operator<< (long double val);
```

Writing to Streams



public member function

std::ostream::put

```
ostream& put (char c);
```

Put character

Inserts character *c* into the stream.

public member function

std::ostream::write

```
ostream& write (const char* s, streamsize n);
```

Write block of data

Inserts the first *n* characters of the array pointed by *s* into the stream.

Reading from Streams

```cpp
#include <iostream>

using namespace std;

int main() {
    cout << "First: ";
    int first;
    cin >> first;

    cout << "Second: ";
    int second;
    cin >> second;

    cout << "You entered " << first << " and " << second;
};
```

- We can think streams as data source or destination.

Reading from Streams

```cpp
#include <iostream>

using namespace std;

int main() {
    // Buffer: temporary storage
    // [10 20]
    cout << "First: ";
    int first;
    cin >> first;

    cout << "Second: ";
    int second;
    cin >> second;
```

- If we type 10 20 as an input, then 10 will be output to the variable ”first”.

Reading from Streams

```cpp
#include <iostream>

using namespace std;

int main() {
    cout << "First: ";
    int first;
    cin >> first;

    cout << "Second: ";
    int second;
    cin >> second;

    cout << "You entered " << first << " and " << second;
};
```

- We can think streams as data source or destination.

Reading from Streams

```cpp
#include <iostream>

using namespace std;

int main() {
    // Buffer: temporary storage
    // [ 20]
    cout << "First: ";
    int first;
    cin >> first;

    cout << "Second: ";
    int second;
    cin >> second;
```

- Since the buffer is not empty, the program will not execute the second **cin** statement. Instead, it will get the number in the buffer and assign to the variable “second”.

Reading from Streams

```cpp
#include <iostream>

using namespace std;

int main() {
    // Buffer: temporary storage
    // [ 20]

    cout << "First: ";
    int first;
    cin >> first;
    cin.ignore( n: numeric_limits<streamsize>::max(),'\n');

    cout << "Second: ";
    int second;
    cin >> second;
```

- Use **cin.ignore** method to clear the buffer.

```
First: a
Second: You entered 0 and 32766
Process finished with exit code 0
```

- **Problem:** What we get if we enter a nonnumeric value.

Handling Input Errors

```cpp
#include <iostream>

using namespace std;

int main() {
    int first;
    while (true) {
        cout << "First: ";
        cin >> first;
        if (cin.fail()) {
            cout << "Enter a valid number!" << endl;
            cin.clear();
            cin.ignore( n: numeric_limits<streamsize>::max(), '\n');
        }
        else break;
    }
```

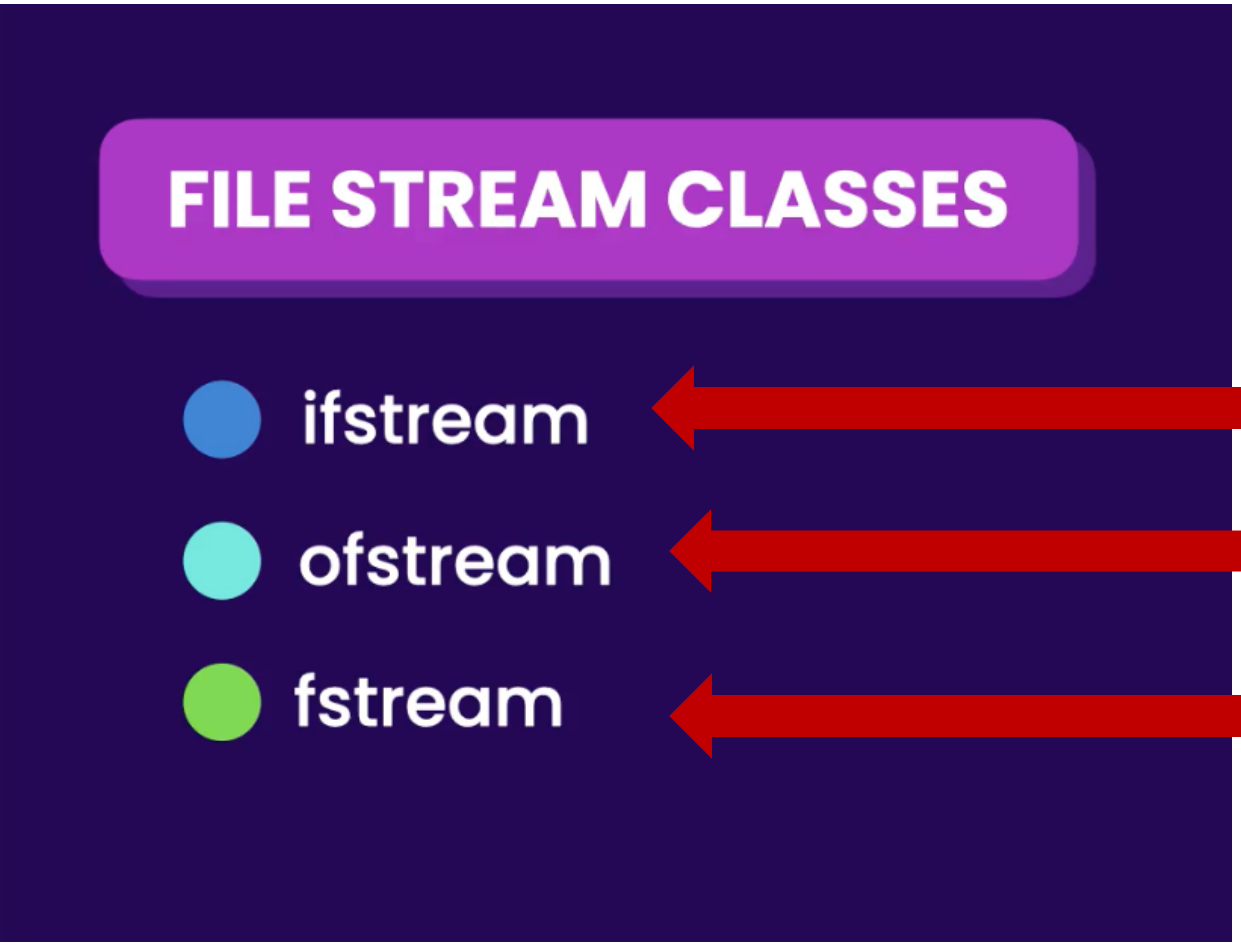
- Use **cin.clear() function** to put the cin in a clear state.
- Use **cin.ignore** function to clear the buffer.

Handling Input Errors

```cpp
#include <iostream>

using namespace std;

int getNumber(const string& prompt) {
    int number;
    while (true) {
        cout << prompt;
        cin >> number;
        if (cin.fail()) {
            cout << "Enter a valid number!" << endl;
            cin.clear();
            cin.ignore( n: numeric_limits<streamsize>::max(), '\n');
        }
        else break;
    }
    return number;
}

int main() {
    int first = getNumber( prompt: "First: ");
    int second = getNumber( prompt: "Second: ");
    cout << "You entered " << first << " and " << second;

    return 0;
}
```

- ❑ Create a function to get the number.
- ❑ A more general implementation.

File Streams



- We can think streams as data source or destination.
- Have the same interface and functions.

ifstream — Just for reading.

ofstream — Just for writing.

fstream — Both for reading and writing.

Writing to Text Files

```cpp
#include <iostream>
#include <fstream>
#include <iomanip>

main() {
ofstream file;
file.open( s: "data.txt");
if (file.is_open()) {
    file << setw( n: 20) << "Hello" << setw( n: 20) << "World" << endl;
    file.close();
}

return 0;
```

- **file.open():** If the file does not exist, it will be created, otherwise it will be overwritten.
- **file.close():** File should always be closed.

Writing to Text Files

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

int main() {
    ofstream file;
    file.open( s: "data.csv");
    if (file.is_open()) {
        // CSV: Comma Separated Value
        file << "id,title,year\n"
             << "1,Terminator 1,1984\n"
             << "2,Terminator 2,1991\n";
        file.close();
    }
```

```cpp
file << "id,title,year" << endl;
file << "1,Terminator 1,1984" << endl;
file << "2,Terminator 2,1991" << endl;
```

- Output streams, similar to input streams, have a buffer.
- When data is output, it is not written to the file immediately.
- Instead, it is put in a buffer.
- Use \n instead of \endl for a better performance.
- **\n:** Buffer only gets flushed once.
- **\endl:** Buffer is flushed for each record.

Reading from Text Files

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

int main() {
    ifstream file;
    file.open( s: "data.csv");
    if (file.is_open()) {
        string str;
        file >> str;
        cout << str;
        file.close();
    }

    return 0;
}
```

- >> reads a file until it finds a delimeter like space or endline.

Reading from Text Files

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

int main() {
    ifstream file;
    file.open( s: "data.csv");
    if (file.is_open()) {
        string str;
        getline( &: file,  &: str);
        cout << str;
        file.close();
    }

    return 0;
}
```

- **getline** reads entire row (until \ character).

Reading from Text Files

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

int main() {
    ifstream file;
    file.open( s: "data.csv");
    if (file.is_open()) {
        string str;
        while (!file.eof()) {
            getline( &: file,  &: str);
            cout << str << endl;
        }
        file.close();
    }

    return 0;
}
```

- **getline** reads entire row (until \ character).
- eof() function returns True if it reaches end of the file.

Reading from Text Files

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

int main() {
    ifstream file;
    file.open( s: "data.csv");
    if (file.is_open()) {
        string str;
        getline( &: file,  &: str);
        while (!file.eof()) {
            getline( &: file,  &: str,  dlm: ',');
            cout << str << endl;
        }
        file.close();
    }

    return 0;
}
```

- **getline** reads entire row (until \ character).
- eof() function returns True if it reaches end of the file.

- Read until the delimeter specified (‘ in this example).
- Default delimeter is newline.

Reading from Text Files

- ❑ Full implementation.

```cpp
#include <iostream>
#include <fstream>
using namespace std;
int main() {
    ifstream file;
    file.open( s: "data.csv");
    if (file.is_open()) {
        string str;
        getline( &: file,  &: str);
        while (!file.eof()) {
            getline( &: file,  &: str,  dlm: ',');
            if (str.empty()) continue;

            Movie movie;
            movie.id = stoi(str);

            getline( &: file,  &: str,  dlm: ',');
            movie.title = str;

            getline( &: file,  &: str);
            movie.year = stoi(str);

            cout << movie.title << endl;
        }
        file.close();
    return 0;
}
```

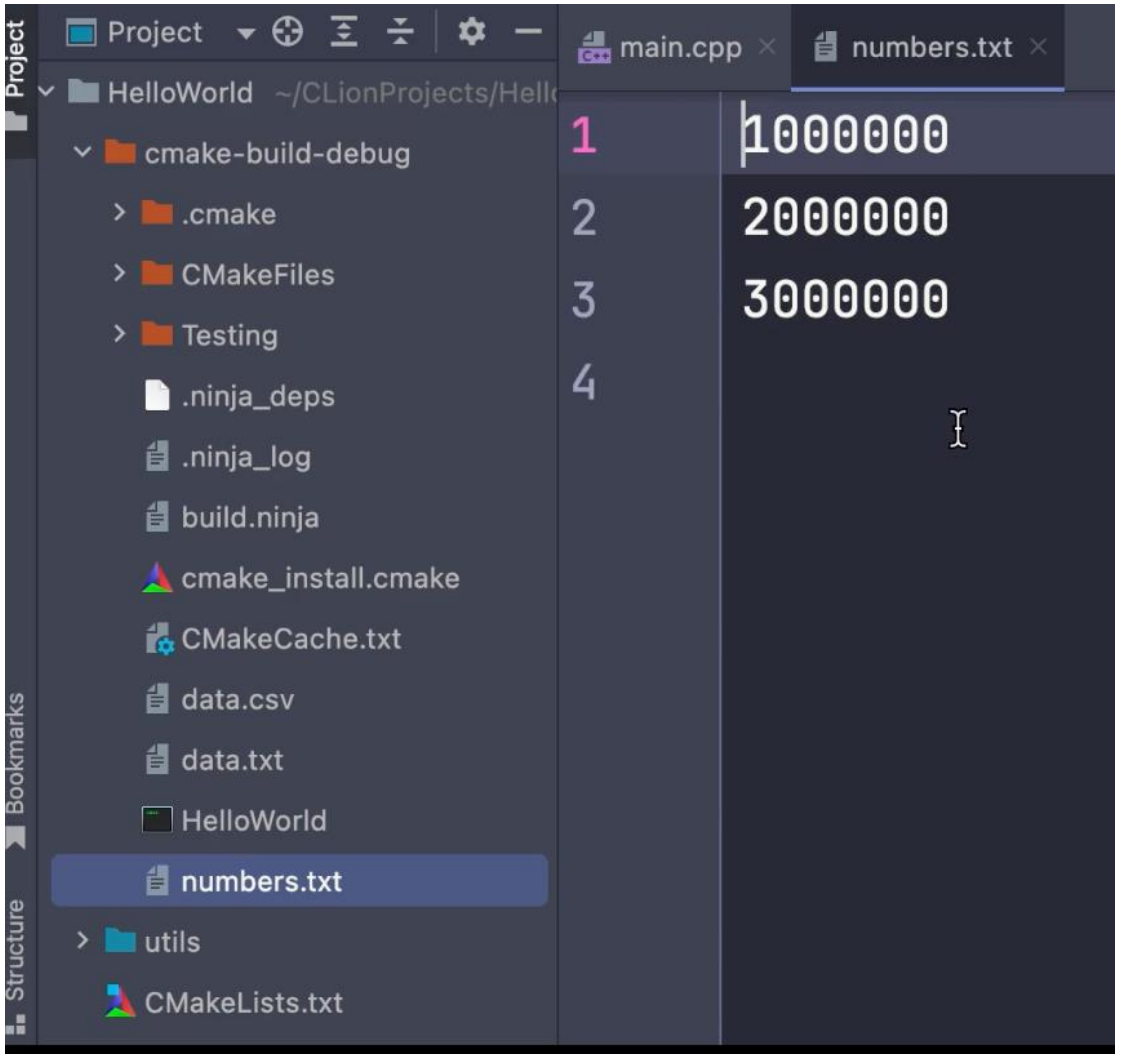
Writing to Binary Files

- ❑ Create to a text file.

FILES

- Text files
- Binary files (images, audio files, PDFs, etc)

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

int main() {
    int numbers[] = {1'000'000, 2'000'000, 3'000'000};
    ofstream file( s: "numbers.txt");
    if (file.is_open()) {
        for (auto number : int : numbers)
            file << number << endl;
        file.close();
    }

    return 0;
```



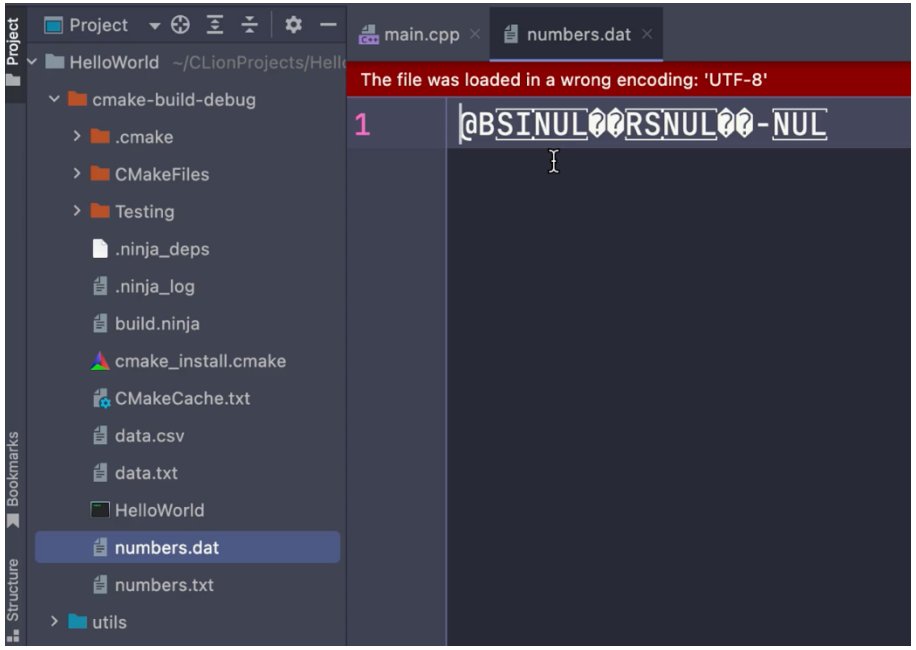
Writing to Binary Files

- Writing to a binary file.
- Binary files are stored in the same way they are stored in memory.

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

int main() {
    ifstream file;
    file.open( s: "data.csv");
```

```cpp
int main() {
    int numbers[] = {1'000'000, 2'000'000, 3'000'000};
    ofstream file( s: "numbers.dat", mode: ios::binary);
    if (file.is_open()) {
        file.write(reinterpret_cast<char*>(&numbers), n: sizeof(numbers));
        file.close();
    }

    return 0;
}
```

```
~/CLionProjects/HelloWorld/cmake-build-debug
❯ ls -l numbers.*
-rw-r--r--  1 moshfeghhamedani  staff  12 Aug  2 11:07 numbers.d
at
-rw-r--r--  1 moshfeghhamedani  staff  24 Aug  2 10:51 numbers.t
xt
```



Reading from Binary Files

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

int main() {
    ifstream file;
    file.open( s: "data.csv");
    ifstream file( s: "numbers.dat",  mode: ios::binary);
    if (file.is_open()) {
        int number;
        while (file.read( s: reinterpret_cast<char*>(&number  n: sizeof(numbers));
            cout << number;
        file.close();
    }

    return 0;
}
```

Working with Binary Files

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

int main() {

    fstream file;
    file.open( s: "file.txt",  mode: ios::in | ios::out | ios::app | ios::binary);
    if (file.is_open()) {
        file.close();
    }

    return 0;
}
```

- **fstream:** Works in both input and output.

String Streams

- We use these classes to convert some value to a string or visa versa.

Converting Values to Strings

```cpp
#include <iostream>

using namespace std;

int main() {
    double number = 12.34;
    string str = to_string( val: number);
    cout << str;

    return 0;
}
```

```
12.340000
Process finished with exit code 0
```

- **to_string:** Converts a number to a string.
- We have no control over the string representation.

Converting Values to Strings

```cpp
#include <iostream>

using namespace std;

int main() {
    double number = 12.34;
    string str = to_string( val: number);
    cout << str;

    return 0;
}
```

```
12.340000
Process finished with exit code 0
```

- **to_string:** Converts a number to a string.
- We have no control over how the number is represented as a string.
- To do that, we use **stringstream**.

Converting Values to Strings

```cpp
#include <iostream>
#include <sstream>
#include <iomanip>
using namespace std;

int main() {
    double number = 12.34;
    stringstream stream;
    stream << fixed << setprecision( n: 1) << number;
    string str = stream.str();
    cout << str;

    return 0;
}
```

```
12.34
Process finished with exit code 0
```

- **to_string:** Converts a number to a string.
- We have no control over how the number is represented as a string.
- To do that, we use **stringstream**.

Converting Values to Strings

```cpp
#include <iostream>
#include <sstream>
#include <iomanip>
using namespace std;

string to_string(double number, int precision) {
    stringstream stream;
    stream << fixed << setprecision( n: precision) << number
    return stream.str();
}

int main() {
    double number = 12.34;
    cout << to_string(number,  precision: 2);

    return 0;
}
```

- Creating a **to_string** function.

Parsing Strings

```cpp
#include <iostream>
#include <sstream>
#include <iomanip>
using namespace std;

int main() {
    string str = "10 20";
    stringstream stream;
    stream.str( s: str);

    int first;
    stream >> first;

    int second;
    stream >> second;

    cout << first + second;

    return 0;
}
```