Object-Oriented Programming

Lecture 12: Stream Input/Output: A Deeper Look

Agenda

Streams

- C++ I/O occurs in streams, which are sequences of bytes.
- Input Streams:
  - In output oBytes flow from a device (e.g., keyboard, disk drive, network) into your computer’s memory.
  - Example: Typing on your keyboard sends characters (bytes) to a program via cin.
- Output Streams:
  - Bytes flow from your computer’s memory to a device (e.g., screen, printer, disk).
  - Example: Displaying text on the screen with cout.
- Streams can represent anything: characters, numbers, images, audio, video, or raw data—whatever your program needs.

```
[Device] --> [Input Stream (e.g., cin)]   --> [Memory]
[Memory] --> [Output Stream (e.g., cout)] --> [Device]
```

Classic Streams vs. Standard Streams

Classic Streams:

- Limited to char-based I/O, handling only single-byte characters.
- Restricted to basic English characters—no support for international languages or symbols like emojis.
- Used in older C++ versions; less flexible for modern applications.
- Example: cout << "Hello"; (only ASCII chars).

Standard Streams:

- Support Unicode via types like wchar_t, char16_t, and char32_t for international character sets.
- Handle multi-byte characters (e.g., UTF-8, UTF-16, UTF-32) for languages and symbols worldwide.
- Essential for global software.
- Example: wcout << L"Françoise"; (Unicode support for accented characters).

lect12_01.cpp

Unicode Encodings

- UTF-8: A popular, space-efficient encoding representing Unicode characters with variable-length sequences of bytes. Often a good default choice.
- UTF-16: Uses 2-byte units for most common characters but can extend to 4 bytes if needed. Common on Windows systems.
- UTF-32: A fixed-width encoding, ensuring every character requires 4 bytes. Less common for everyday use.

C++ Data Types

- char16_t and char32_t: Unicode-specific character types designed to hold UTF-16 and UTF-32 code points respectively.
- wchar_t: A wider character. Its exact size and encoding are platform-dependent.
- Libraries: Consider libraries like ICU (International Components for Unicode) for advanced string operations.

Example: UTF-8 with Standard Strings

```
#include <iostream>
#include <string>

int main() {
   std::string name = u8"Françoise"; // u8 designates UTF-8 string literal
   std::cout << "Hello, " << name << "!" << std::endl;

   return 0;
}
```

lect12_02.cpp

Example: UTF-16 and Code Points

```
#include <iostream>
#include <string>

int main() {
   std::u16string greeting = U"こんにちは"; // U for UTF-16
   char16_t smile = 0x1F603; // Code point for a smiling face emoji

   std::cout << greeting << " " << smile << std::endl;

   return 0;
}
```

lect12_03.cpp

Types of I/O Operations

Low-level (unformatted) I/O:

- Transfers raw bytes without considering data type.
- Transfers raw bytes directly, ignoring data types (fast but hard to use).
- Example: Using write to send 10 bytes of raw data to a file, like “HAPPY BIRT” (no structure).

High-level (formatted) I/O:

- Organizes data into meaningful units (e.g., integers, strings, floats) for easy use.
- Example: Using cin >> number to read a number or cout << "Hello" to print text.

Low-level I/O

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

int main() {
   // Example of unformatted output using write
   char buffer[] = "HAPPY BIRTHDAY";
   cout.write(buffer, 10);  // Outputs the first 10 bytes of buffer
   // This will output "HAPPY BIRT"
   return 0;
}
```

Lect12_04.cpp

High-level I/O

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

int main() {
   int number;

   cout << "Enter a decimal number: ";
   cin >> number; // input number
   cout << number << " in hexadecimal is: " << hex << number << "\n";
   cout << dec << number << " in octal is: " << oct << number << "\n";
   cout << setbase(10) << number << " in decimal is: " << number << endl;

   return 0;
}
```

Lec12_05.cpp

iostream Library Headers

<iostream>: Provides fundamental input/output facilities in C++. Includes the following key objects:

- cin: Standard input stream (typically connected to the keyboard).
- cout: Standard output stream (typically connected to the console).
- cerr: Unbuffered standard error stream (outputs immediately).
- clog: Buffered standard error stream (output may be delayed).

<iomanip>: Offers tools for output formatting. Includes manipulators such as:

- setprecision: Controls floating-point number precision.
- setw: Sets the field width for output values.

Understanding I/O Stream Errors

Streams can encounter various errors during input/output operations. Some common examples include:

File not found: When trying to open a non-existent file.

Disk errors: Hardware issues preventing data access.

Insufficient permissions: Lacking necessary privileges to perform an operation.

Invalid input: User entering data that doesn't match the expected format.

Error Handling Mechanisms

Member Functions:

bad(): Checks for any error condition on the stream.

eof(): Indicates if the end-of-file has been reached (not necessarily an error).

fail(): Returns true if a previous operation failed.

Exceptions: (C++11 and later)

The <iostream> library includes exceptions like ios_base::failure or derived classes for specific errors..

Stream Output: Output of char* Variables

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

int main() {
   const char* const word{"again"};

   // display the value of char* variable word, then display
   // the value of word after a static_cast to void*
   cout << "Value of word is: " << word
      << "\nValue of static_cast<const void*>(word) is: "
      << static_cast<const void*>(word) << endl;
}
```

```
Value of word is: again
Value of static_cast<const void*>(word) is: 00DE8B30
```

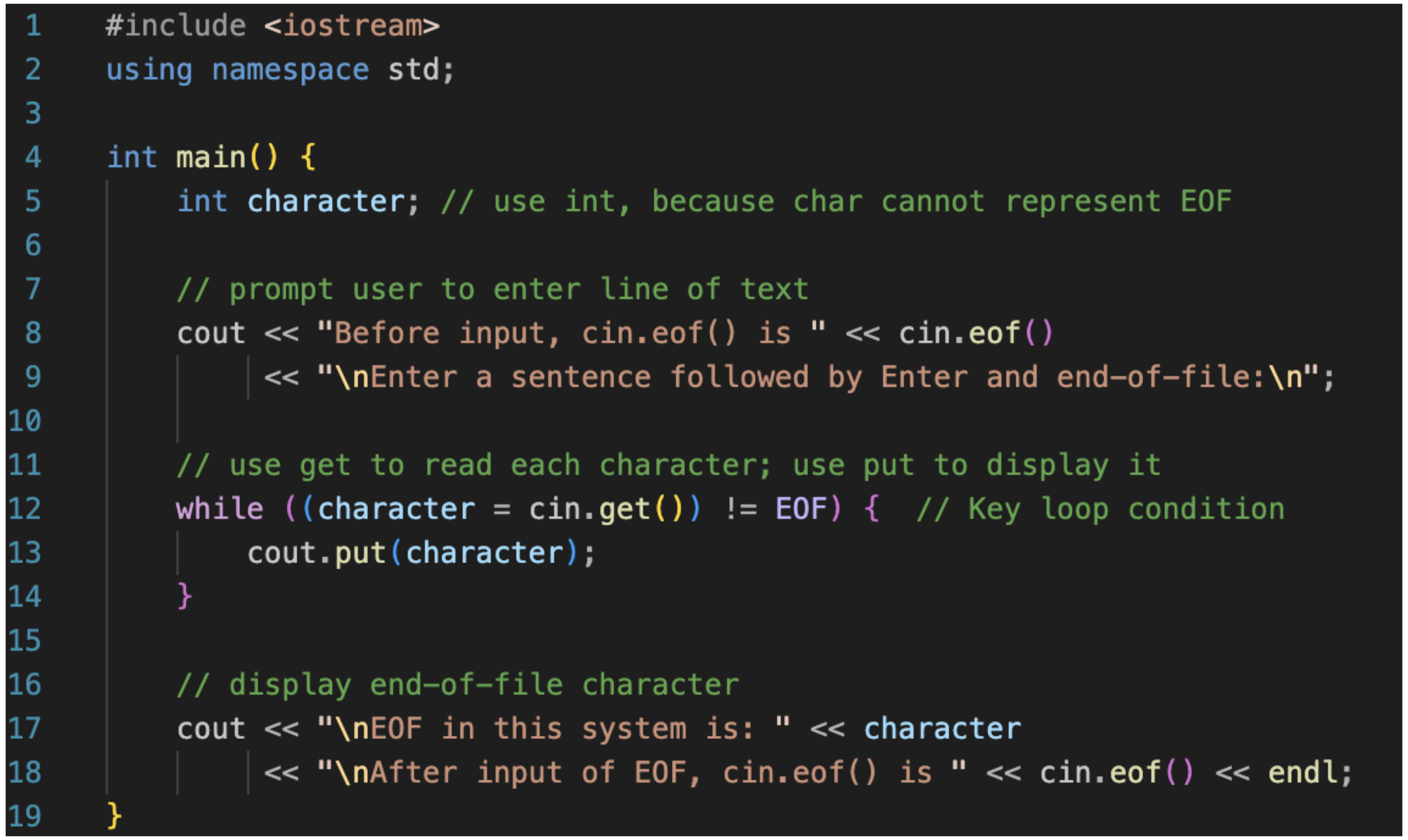
const char*: This declares a pointer named word. The pointer points to a const char, meaning the characters the pointer points to cannot be modified through this pointer.

const (after the asterisk): This second const makes the pointer itself constant. You cannot reassign word to point to a different memory location.

static_cast: This performs a specific type of conversion between compatible types. It's used when the compiler cannot implicitly do the conversion for you.

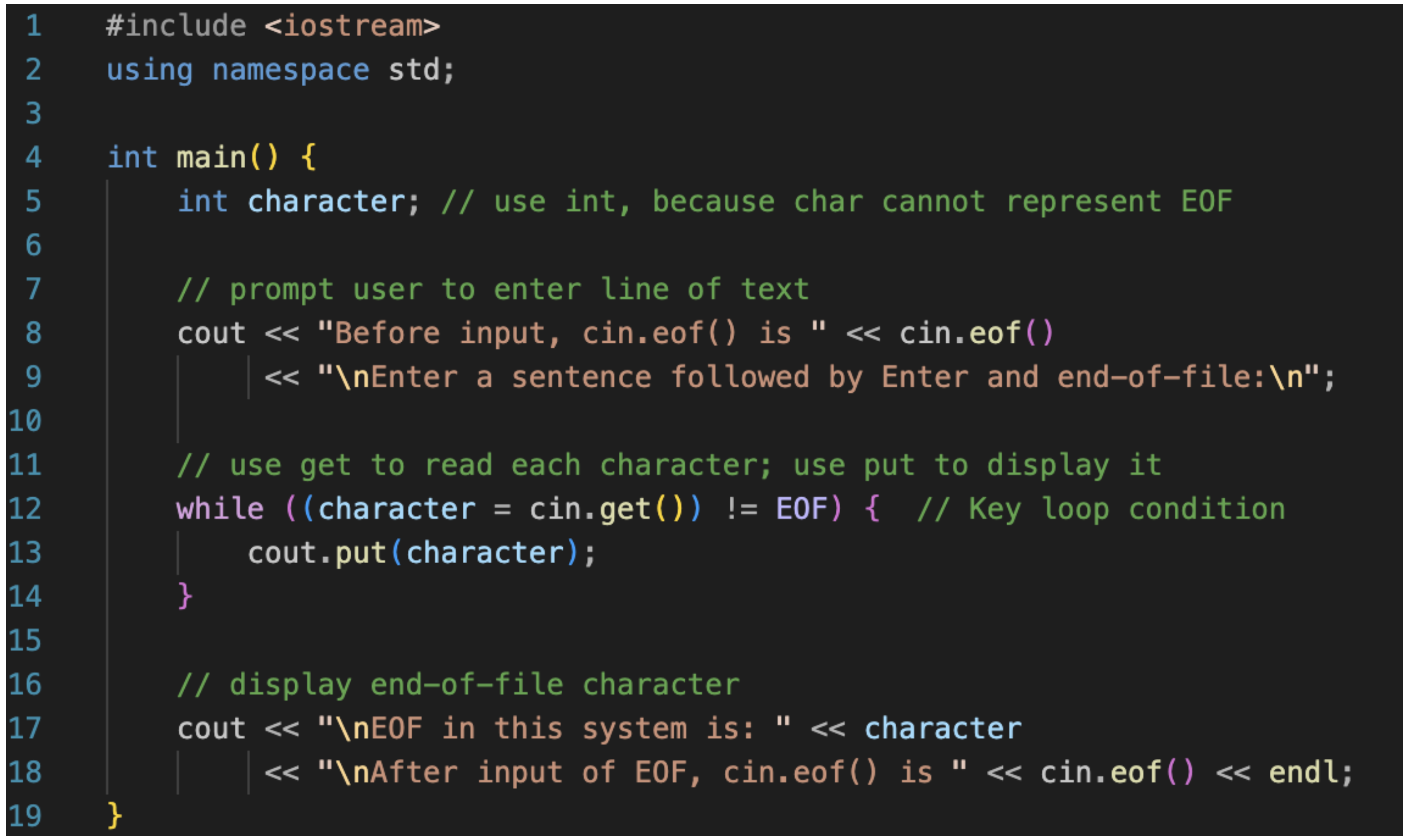
•**const void*:** This is a "generic pointer" type that does not point to any specific data type. It can hold a memory address but does not allow you to dereference (access the underlying data type).

Stream Input



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

int main() {
    int character; // use int, because char cannot represent EOF

    // prompt user to enter line of text
    cout << "Before input, cin.eof() is " << cin.eof()
         << "\nEnter a sentence followed by Enter and end-of-file:\n";

    // use get to read each character; use put to display it
    while ((character = cin.get()) != EOF) {  // Key loop condition
        cout.put(character);
    }

    // display end-of-file character
    cout << "\nEOF in this system is: " << character
         << "\nAfter input of EOF, cin.eof() is " << cin.eof() << endl;
}
```

Compare cin and cin.get



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

int main() {
    int character; // use int, because char cannot represent EOF

    // prompt user to enter line of text
    cout << "Before input, cin.eof() is " << cin.eof()
        << "\nEnter a sentence followed by Enter and end-of-file:\n";

    // use get to read each character; use put to display it
    while ((character = cin.get()) != EOF) {  // Key loop condition
        cout.put(character);
    }

    // display end-of-file character
    cout << "\nEOF in this system is: " << character
        << "\nAfter input of EOF, cin.eof() is " << cin.eof() << endl;
}
```

```
Enter a sentence:
Contrasting string input with cin and cin.get

The string read with cin was:
Contrasting

The string read with cin.get was:
 string input with cin and cin.get
```

Using Member Function getline

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

int main() {
   const int SIZE{80};
   char buffer[SIZE]; // create array of 80 characters

   // input characters in buffer via cin function getline
   cout << "Enter a sentence:\n";
   cin.getline(buffer, SIZE);

   // display buffer contents
   cout << "\nThe sentence entered is:\n" << buffer << endl;
}
```

```
Enter a sentence:
Using the getline member function

The sentence entered is:
Using the getline member function
```

istream Member Functions peek, putback and ignore

peek() - "Peek" at the next character in the input stream without removing it.
putback() - "Put back" the previously extracted character into the input stream (essentially undoing a get()).
ignore() - Discard characters from the input stream.

Unformatted I/O Using read, write and gcount

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

int main() {
   const int SIZE{80};
   char buffer[SIZE]; // create array of 80 characters

   // use function read to input characters into buffer
   cout << "Enter a sentence:\n";
   cin.read(buffer, 20);

   // use functions write and gcount to display buffer characters
   cout << "\nThe sentence entered was:\n";
   cout.write(buffer, cin.gcount());
   cout << endl;
}
```

```
Enter a sentence:
Using the read, write, and gcount member functions
The sentence entered was:
Using the read, writ
```

Stream Manipulators: A Deeper Look

peek() - "Peek" at the next character in the input stream without removing it.
putback() - "Put back" the previously extracted character into the input stream (essentially undoing a get()).
ignore() - Discard characters from the input stream.

Stream Manipulators: A Deeper Look

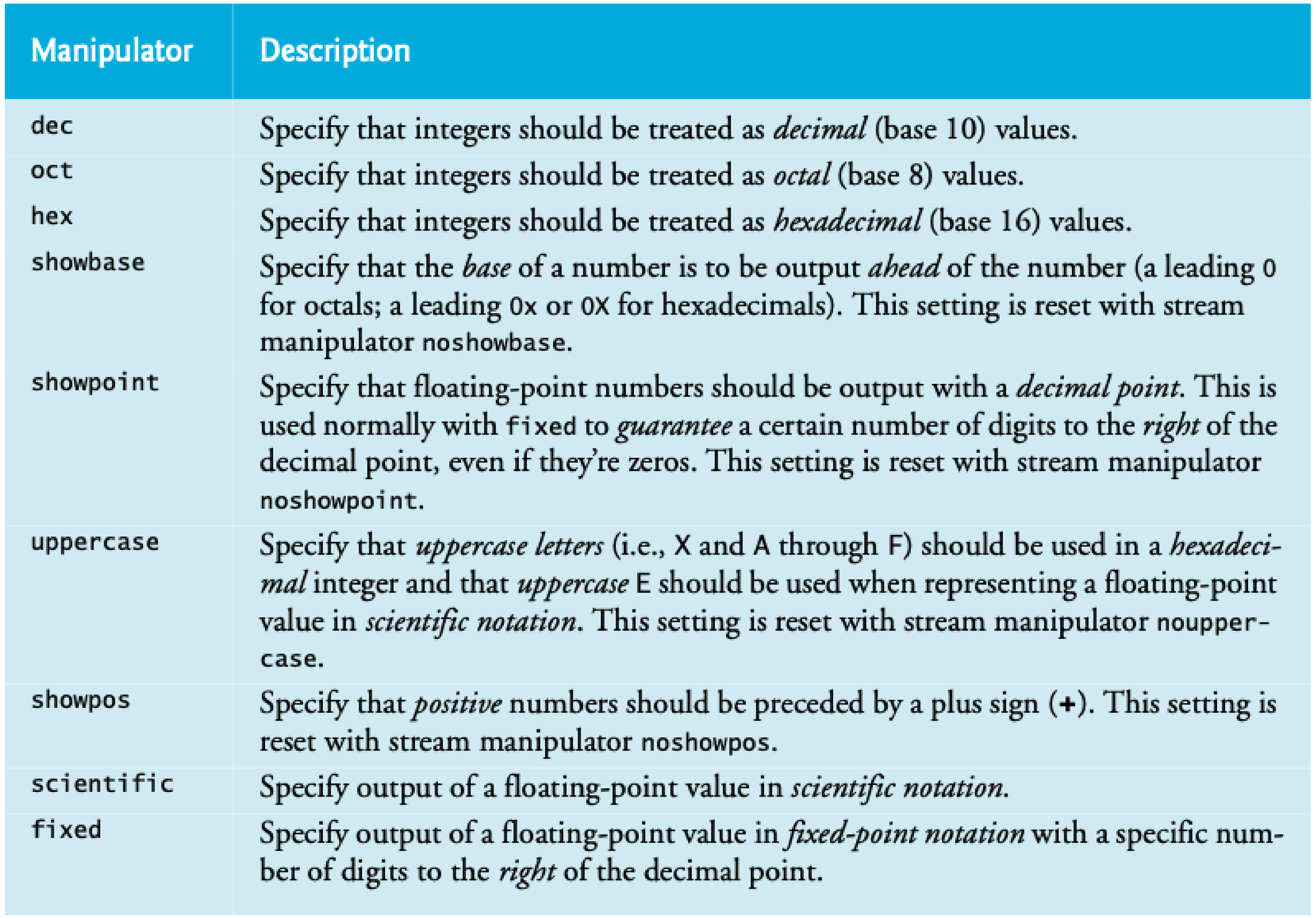
setting field widths
setting precision
setting and unsetting format state
setting the fill character in fields
flushing streams
inserting a newline into the output stream (and flushing the stream)
inserting a null character into the output stream
skipping white space in the input stream

lec12_11.cpp

Stream Format States and Stream Manipulators

Manipulator	Description
`skipws`	*Skip white-space characters* on an input stream. This setting is reset with stream manipulator `noskipws`.
`left`	*Left justify* output in a field. *Padding* characters appear to the *right* if necessary.
`right`	*Right justify* output in a field. *Padding* characters appear to the *left* if necessary.
`internal`	Indicate that a number's *sign* should be *left justified* in a field and a number's *magnitude* should be *right justified* in that same field (i.e., *padding* characters appear *between* the sign and the number).
`boolalpha`	Specify that *`bool` values* should be displayed as the word `true` or `false`. The manipulator `noboolalpha` sets the stream back to displaying bool values as 1 (true) and 0 (false).

Stream Format States and Stream Manipulators



Manipulator	Description
`dec`	Specify that integers should be treated as *decimal* (base 10) values.
`oct`	Specify that integers should be treated as *octal* (base 8) values.
`hex`	Specify that integers should be treated as *hexadecimal* (base 16) values.
`showbase`	Specify that the *base* of a number is to be output *ahead* of the number (a leading `0` for octals; a leading `0x` or `0X` for hexadecimals). This setting is reset with stream manipulator `noshowbase`.
`showpoint`	Specify that floating-point numbers should be output with a *decimal point*. This is used normally with `fixed` to *guarantee* a certain number of digits to the *right* of the decimal point, even if they're zeros. This setting is reset with stream manipulator `noshowpoint`.
`uppercase`	Specify that *uppercase letters* (i.e., `X` and `A` through `F`) should be used in a *hexadecimal* integer and that *uppercase* `E` should be used when representing a floating-point value in *scientific notation*. This setting is reset with stream manipulator `nouppercase`.
`showpos`	Specify that *positive* numbers should be preceded by a plus sign (`+`). This setting is reset with stream manipulator `noshowpos`.
`scientific`	Specify output of a floating-point value in *scientific notation*.
`fixed`	Specify output of a floating-point value in *fixed-point notation* with a specific number of digits to the *right* of the decimal point.

Setting and Resetting the Format State via Member Function flags

The flags member function of an I/O stream object without an argument is used to retrieve the current format state of the stream. The format state includes various flags that control how the stream performs input and output operations, such as whether integers are displayed in decimal or hexadecimal format, whether floating-point numbers show trailing zeros, and so forth.
The format state is returned as a value of the fmtflags data type, which is a bitmask type defined within the I/O stream classes. You can use this value to store the current format state before making any changes to the stream's formatting options, allowing you to restore the original state later if needed.

lec12_1.cpp

Multiprocessing

Multiprocessing has allowed us to spawn other processes to do tasks or run programs Powerful; can execute/ wait on other programs, secure (separate memory space), communicate with pipes and signals But limited; interprocess communication is cumbersome, hard to share data/coordinate
Is there another way we can have concurrency beyond multiprocessing that handles these tradeoffs differently?

Multithreading

We can have concurrency within a single process using threads: <u>independent</u> execution sequences within a single process.
Threads let us run multiple functions in our program concurrently
Multithreading is very common to parallelize tasks, especially on multiple cores
In C++: spawn a thread using thread() and the thread variable type and specify what function you want the thread to execute (optionally passing parameters!)
Thread manager switches between executing threads like the OS scheduler switches between executing processes
Each thread operates within the same process, so they share a virtual address space (global, text, data, and heap segments)
The processes's stack segment is divided into a "ministack" for each thread.
Threads are often called lightweight processes.

C++ thread

A thread object can be spawned to run the specified function with the given arguments.

```
thread myThread(myFunc, arg1, arg2, ...);
```

args: a list of arguments (any length, or none) to pass to the function upon execution Once initialized with this constructor, the thread may execute at any time!

Thread function's return value is ignored (can pass by reference instead)

Joining Threads.

Detaching Threads.

Race Conditions

Like with processes, threads can execute in unpredictable orderings.
A race condition is an unpredictable ordering of events where some orderings may cause undesired behavior.
A thread-safe function is one that will always execute correctly, even when called concurrently from multiple threads.
printf is thread-safe, but operator<< is not. This means e.g. cout statements could get interleaved!

Threads Share Memory

Like with processes, threads can execute in unpredictable orderings.
Unlike parent/child processes, threads execute in the same virtual address space This means we can e.g. pass parameters by reference and have all threads access/modify them!
To pass by reference with thread(), we must use the special ref() function around any reference parameters:

Thread Safety and Data Races

Mutex
A mutex (mutual exclusion) is a synchronization primitive used to prevent multiple threads from simultaneously accessing shared data. std::mutex in C++ provides basic locking and unlocking functionality.

Lec12_04.cpp

Resource Acquisition Is Initialization

RAII (Resource Acquisition Is Initialization) is a programming idiom that ensures resources are properly released when objects go out of scope. std::lock_guard and std::unique_lock are RAII wrappers for mutexes that automatically lock the mutex when they are constructed and unlock it when they are destroyed.

```
void increment() {
    std::lock_guard<std::mutex> guard(mtx); // Automatically locks the mutex
    ++shared_data;
    // Mutex is automatically unlocked when guard goes out of scope
}
```

std::lock_guard

Provides a simple, lightweight wrapper for owning a mutex on a scoped basis.
Automatically locks the associated mutex when constructed and unlocks it when destroyed.
Cannot be manually unlocked or relocked, and it doesn't support condition variables directly.
Suitable for most scenarios where you need to lock a mutex for the duration of a scope and where exception safety is a concern.

```
std::mutex mtx;
{
  std::lock_guard<std::mutex> lock(mtx);
  // Protected code here
} // mtx is automatically released here
```

std::unique_lock

More flexible and heavier than std::lock_guard.
Can be constructed without immediately locking the mutex, allowing for deferred locking.
Supports manual locking and unlocking, lock upgrading and downgrading, and recursive locking.
Can be used with condition variables (std::condition_variable), making it suitable for more complex synchronization tasks like waiting for conditions or implementing producer-consumer patterns.

```
std::mutex mtx;
std::unique_lock<std::mutex> lock(mtx, std::defer_lock); // Defer locking
// Do something without lock...
lock.lock(); // Manually lock the mutex
// Protected code here
lock.unlock(); // Manually unlock the mutex
// Do something else without lock...
```

std::condition_variable

Homewrok 3

Homework Overview: TCP Socket Programming

1. Run and analyze provided sender (client) and receiver (server) programs to understand TCP communication.
2. Write a brief explanation of how these programs use sockets, IP addresses, and ports.
3. Create a single program that both sends and receives messages, using a Program ID to determine ports (e.g., ID 1 = port 5001). The program sends to a specified IP and ID, and only displays received messages if its ID matches the target port.
4. Test the program on the same machine (IP 127.0.0.1) and different machines, documenting the results.

Deliverables include the combined program’s code and a document with explanations and test

Q&A