# 📚 C++ Course Overview

### Brief Overview

This note covers **C++ Programming Course** and was created from an uploaded audio.
It covers course logistics, learning resources, assessment outline, academic integrity, and algorithmic examples.

### Key Points

- Structure of lectures, tutorials, and the Ed platform
- Detailed grading scheme for quizzes, assignments, and the final exam
- Coding practices and copy-paste policies using Ed and Git
- Core C++ fundamentals: syntax, types, STL containers, and common patterns
- Algorithmic techniques highlighted in the coursework, such as hashing and sorting approaches

---

## 📧 Course Administration

- **Lecturer:** Troy Lee
- **Contact:** Email (use for personal questions) & **Ed discussion forum** (primary place for course queries).
- **Tutorials:**
  - Liberal attendance policy – attend any tutorial, multiple per week if desired.
  - No formal transfers needed for schedule conflicts; just join a session you can attend.
  - Tutorials start the first week; each includes **ungraded coding challenges** to reinforce lecture material.

> **Tutorial purpose:** Provide hands-on practice with concepts covered in lectures; not mandatory but highly encouraged.

- **Ed platform:** Central hub for all course materials.
  - **Discussion forum**, **course resources** (install guides, IDE setup, tutorial schedule, due dates).
  - **Lecture videos** (pre-recorded) on the **MH1200 YouTube channel**; watch before live sessions for interactive Q&A.
  - **Live lecture slides** and **tutorial exercises** are also hosted on Ed.

## 🖥️ Learning Resources & Tools

- **C++ environment:**

    - No need to install a compiler for the first tutorial – code can be written and executed directly in Ed.
    - All assignments can be completed on Ed; local installation is optional.
- **Google Test**

> *A C++ testing framework used to create automatic test cases for coding challenges and assignments.*

    - Required only if you run code locally; Ed runs tests in the background.
    - A tutorial on installing Google Test is provided for Windows users who encounter issues.
- **Style Guide**

> *Google C++ Style Guide – recommended for consistent formatting and coding style.*

    - Use **cpplint** (a Python linter) to check compliance; passing cpplint removes style-related deductions.
- **Git workflow (optional)**
    - Clone assessment repositories directly from Ed.
    - Commit **frequently** (e.g., every ~30 lines) with clear messages describing the change.
    - Enables tracking of code evolution, satisfying the copy-paste policy when working locally.

## 📊 Assessments Overview

| Assessment type | Quantity | Weight | Key features |
|---|---|---|---|
| Weekly graded exercises (quizzes) | 10 (one every Wednesday) | 2 % each (total 20 %) | • 5 multiple-choice questions<br>• One coding challenge (auto-graded via test cases)<br>• Submissions open ~10 days, due Sunday before midnight |
| Programming Assignment 1 | 1 | 20 % | • Test cases (auto-graded) + |

| | | | |
|---|---|---|---|
| | | | human-graded style & documentation |
| Programming Assignment 2 | 1 | 30 % | • Same dual marking as Assignment 1 |
| Final exam (take-home) | 1 | 30 % | • 20-25 multiple-choice questions<br>• Coding portion (harder than weekly quizzes)<br>• 48-hour window, ~3 hours effort, completed on Ed |

- **Attendance:** Neither lectures nor tutorials are mandatory, but participation is strongly encouraged.
- **Solution access:** Quiz solutions become visible only after the deadline; coding challenges provide test case feedback but not full solutions.

## 🛠️ Coding & Academic Integrity Policies

- **Copy-paste prohibition:**
  - **Option 1:** Write code directly in Ed; Ed logs every keystroke. A sudden jump from a blank function to a complete one flags misconduct.
  - **Option 2:** Work locally with Git; commit small, logical increments with explanatory messages. A single massive commit from empty to complete will also be flagged.
- **Violation consequence:** Referral to the misconduct team, typically resulting in a zero for the affected assessment.

## 📚 Algorithm Design Example: Distinct Integers

**Problem:** Determine whether all integers in an array are unique.

### Proposed Solutions

1. **HashMap / unordered_set** (C++ equivalent)
   - Insert each element; if insertion fails (already present), the array is not distinct.
2. **Sorting + Adjacent Comparison**
   - Sort the array (e.g., ascending).
   - Scan once, checking if any adjacent pair is equal.

3. **Double for-loop (brute force)**
    - Compare every pair (i, j) with i ≠ j; if any match is found, duplicates exist.

### Complexity Comparison

| Approach | Time Complexity | Memory Usage |
|---|---|---|
| Double for-loop | $O(N^2)$ | $O(1)$ |
| HashMap / unordered_set | $O(N)$ average | $O(N)$ |
| Sorting + scan | $O(N \log N)$ | $O(1)$ (in-place) or $O(N)$ (extra array) |

> **Key insight:** As $N$ grows, $O(N^2)$ becomes prohibitive, while $O(N)$ and $O(N \log N)$ remain tractable.

### Benchmark Snapshot (illustrative)

- **Double loop:** Rapid increase in runtime with larger $N$ .
- **HashMap / set:** Near-linear scaling, consistently faster than double loop.
- **Sorting:** Slightly slower than hash-based method but still far better than quadratic approach.

**Practical relevance:** Detecting duplicate submissions among ~700 students would be infeasible with a double loop; a hash-based or sorting solution is essential.

## 🔍 Key Concepts (Definitions)

> **HashMap / unordered_set:** A data structure that provides average-case $O(1)$ time for *insert* and *contains* operations by hashing keys.

> **Google Test:** A unit-testing framework for C++ that automates verification of program correctness via predefined test cases.

> **Copy-paste policy:** A rule requiring incremental code development (keystroke logging in Ed or frequent Git commits) to ensure individual work and prevent plagiarism.

> **Google C++ Style Guide:** A set of conventions for formatting, naming, and structuring C++ code; adherence can be validated with the cpplint tool.

> **Complexity notation:**

> - $O(N)$ – linear time, proportional to input size.
> - $O(N \log N)$ – linearithmic, typical of efficient sorting algorithms.
> - $O(N^2)$ – quadratic, often impractical for large inputs.

## 🛠️ Compiling & Running C++ Programs 🚀

- **Compilers**
  - **Clang++** – preferred on macOS (Apple's version works well with M-series chips).
  - **g++** – the GNU compiler, common on Linux/Windows.
- **Typical compile command** (C++20, all warnings, address sanitizer):

```
clang++ -std=c++20 -Wall -Wextra -fsanitize=address -o myProgram main.cpp
```

- -std=c++20 → selects the C++20 language standard.
- -Wall -Wextra → enable most warning groups.
- -fsanitize=address → activates **address sanitizer** to catch out-of-bounds memory accesses.
- -o myProgram → names the output executable; without -o the default is a.out.
- **Running the executable**

```
./myProgram
```

- **Return value of main**
  - 0 → successful termination (convention from early Unix).
  - Non-zero → indicates an error code (e.g., segmentation fault may return 11).

> **Note:** The course's automated assessment environment compiles with the above flags by default.

---

## 📖 Basic C++ Program Structure

### 📦 Minimal program

```
int main() {
    return 0;
}
```

- Every C++ program must contain a main function that returns an int.
- Omitting an explicit return is allowed for main; the compiler implicitly returns 0.

### 🖨️ Printing with std::cout

- Include the I/O library:

```
#include
```

- **Insertion operator (<<)** sends data to an output stream:

```
std::cout << "Hello, world!" << std::endl;
```

> **Definition** – *Insertion operator*: an overloaded operator that takes a stream on the left and a value on the right, writing the value into the stream.

- std::endl inserts a newline **and** flushes the output buffer. Flushing after every line can degrade performance; use it only when an immediate display is required.

> **Definition** – *Buffer*: a temporary storage area that accumulates output before it is actually written to the console, reducing the cost of many small I/O operations.

### ⚙️ Using using namespace std;

- The directive brings all names from the std namespace into the global scope:

```
using namespace std;
```

- **Recommendation:** avoid it in early learning stages to keep the origin of identifiers (e.g., std::cout) explicit.

---

## 📊 Built-in Types Overview

| Type | Typical size | Description |
|---|---|---|

| | | |
|---|---|---|
| int | 4 bytes | Signed integer. |
| float | 4 bytes | Single-precision floating-point. |
| double | 8 bytes | Double-precision floating-point. |
| char | 1 byte | Single character (often used for ASCII). |

- Example of concatenating output with different types:

```
char letter = 'A';
std::cout << "The letter is: " << letter << std::endl;
```

---

## 📚 Data Structures Mentioned Earlier

- **Resizable array** – std::vector (C++), analogous to Java's ArrayList. Allows O(1) amortized push_back and random access by index.
- **Hash map** – std::unordered_map / std::unordered_set. Provides average-case O(1) insertion, deletion, and membership tests; ideal for duplicate detection (as used in the "identical submissions" check).
- **Balanced binary search tree** – std::set. Supports O(log N) insertion, deletion, and lookup while maintaining sorted order.

These structures will appear repeatedly in interview-style algorithm problems.

---

## 🧩 Algorithmic Techniques Covered in the Course

- **Divide and conquer** – recursively break a problem into sub-problems, solve each, then combine.
- **Greedy algorithms** – make the locally optimal choice at each step with the hope of reaching a global optimum.
- **Graph algorithms** – traversal (e.g., DFS/BFS), shortest-path (e.g., Dijkstra), connectivity.

Understanding when to apply each technique is essential for the coding interview questions that resemble the weekly exercises.

---

## 💡 Learning Resources & Practice Platforms

- **LeetCode** – practice C++ syntax on easy problems, then progress to harder "Blind 75" list.
  - Suggested habit: 10 minutes of coding daily for consistent reinforcement.
- **Ed discussion forum** – primary venue for course-specific questions; keeps content organized.
- **Programming Society Discord** – community support and peer collaboration.
- **U-Pass** – provides additional learning materials and resources.

---

## 🗂️ Historical Context of C++

- **Origin** – Bjarne Stroustrup extended C with classes at Bell Labs (first compiler released 1982).
- **C compatibility** – early C++ could compile pure C code, easing adoption.
- **Standard evolution** – major modern update in 2011 (C++11), followed by revisions every three years: C++14, C++17, C++20, C++23.
- **Generic programming** – championed by Alexander Stepanov, leading to the Standard Template Library (STL), which supplies generic containers (vector, set, …) and algorithms.

These milestones explain many of the language features you will use throughout the course.

## 📏 Data Type Sizes & Limits

- **char**
  - sizeof(char) is defined as **1** (one *character*).
  - Usually occupies **1 byte** (8 bits), but exact size can depend on the system.
- **int**
  - sizeof(int) typically returns **4**, meaning **4 bytes** (32 bits).
  - Range (using two's-complement representation):

| Type | Bits | Minimum value | Maximum value |
|---|---|---|---|
| int | 32 | $-2^{31} \approx$ **-2,147,483,648** | $2^{31} - 1 \approx$ **2,147,483,647** |

- **unsigned int** – same storage (4 bytes) but only non-negative values.

| Type | Bits | Minimum | Maximum |
|---|---|---|---|
| unsigned int | 32 | 0 | $2^{32} - 1 \approx$ **4,294,967,295** |

> **std::numeric_limits::max()** – *Provides the largest representable value for type T (requires <limits> header).*

```
#include
#include

int main() {
    std::cout << "max int: " << std::numeric_limits::max() << '\n';
    std::cout << "max unsigned int: " << std::numeric_limits::max() << '\n';
}
```

---

## 🧩 The std::string Type

- Not a built-in type, but a **standard library** class that handles mutable text.
- Typical usage:

```
#include
#include

int main() {
    std::string greeting = "Hello, world!";
    std::cout << greeting << '\n';
}
```

- std::string offers convenient operations (concatenation, length, etc.) and will appear frequently in assignments.

---

## ⚙️ Variable Initialization Styles

| Style | Syntax | Characteristics |
|---|---|---|
| **Copy initialization** | int x = 3; | Allows implicit conversions; may generate warnings. |
| **Direct initialization** | int x(3); | Same semantics as copy, but without =. |
| **Brace (list) initialization** | int x{3}; | Introduced in C++11; **prevents narrowing** (e.g., int x{3.3}; is an error). |
| **Aggregate initialization (arrays)** | int a[3] = {1, 2, 3}; | Initializes each element; note the brackets denote an array, not a single int. |

> **Brace initialization** – *A uniform initialization syntax that disallows implicit narrowing conversions, improving type safety.*

### Example: Preventing a narrowing conversion

```
int x{3.3};   // error: cannot narrow double to int
int y = 3.3;  // compiles with warning: implicit conversion from double to int
```

---

## 🚨 Uninitialized Variables & Undefined Behavior

- Declaring a variable **without** an initializer leaves it with an *indeterminate* value:

```
int x;        // x is uninitialized; using it triggers undefined behavior
std::cout << x; // undefined behavior, may print any garbage value
```

- Compilers typically emit a warning such as **"warning: 'x' is used uninitialized."**
- Relying on such values can cause nondeterministic program behavior across runs.

> **Undefined behavior** – *Program actions that the C++ standard does not prescribe, leading to unpredictable results.*

---

## 🔀 Implicit Conversions & Narrowing

- **Numeric conversion** (e.g., double → int) is allowed but may truncate data. Compilers issue warnings:

```
int x = 3.3;  // warning: implicit conversion from double to int
```

- **Character arithmetic**: char is essentially an integer holding its ASCII code. Adding an integer promotes the char to an int.

```
char letter = 'a';
std::cout << letter + 1 << '\n'; // prints 98 (ASCII 'a' = 97)
```

- Using brace initialization blocks such implicit narrowing:

```
int z{3.3};   // error: cannot narrow
```

---

## const Qualifier

- Declaring a variable const makes it **immutable** after initialization.

```
const int z = 5;
z = 6;   // error: cannot assign to a variable that is const
```

- Good practice: mark values that should never change as const to improve code readability and enable compiler checks.

> **const** – *A type qualifier that enforces read-only semantics for the declared object.*