Lecture: Introduction to Basic Programming Constructs

1. The Simplest Program

1. The Simplest Program

```
#include <iostream>
int main() {
    return 0;
}
```

- #include <iostream>: includes input/output functionality.
- int main(): the entry point of the program.
- return 0;: indicates successful execution.

2. Data Types

Fundamental built-in types include:

- Integer types: int, short, long, long long
- Floating-point types: float, double
- Character types: char, wchar_t
- Boolean type: bool (true/false)

Each type has a specific size and range (e.g., int is typically 32 bits).

2. Data Types

1. Fundamental (Built-in) Types

These are provided by the language itself and fall into several categories.

a) Integral Types

Used to represent whole numbers and characters.

2. Data Types

Type	Typical Size	Value Range (approx.)
bool	1 byte	true or false
char	1 byte	–128 to 127 **or** 0 to 255*
short	2 bytes	–32,768 to 32,767
unsigned short	2 bytes	0 to 65,535
int	4 bytes	–2,147,483,648 to 2,147,483,647
unsigned int	4 bytes	0 to 4,294,967,295
long	4 or 8 bytes	Platform-dependent
unsigned long	4 or 8 bytes	—
long long (C++11)	8 bytes	$\approx \pm 9 \times 10^{18}$
unsigned long long	8 bytes	0 to $\approx 1.8 \times 10^{19}$

2. Data Types

For portable code, use fixed-width integer types from <cstdint>:

```
#include <cstdint>
std::int32_t i; // exactly 32-bit signed integer
std::uint8_t flag;//exactly 8-bit unsigned
integer
```

2. Data Types

Represent real (fractional) numbers.

Type	Size	Precision	Approximate Range
float	4 bytes	~6–7 digits	$\pm 3.4 \times 10^{38}$
double	8 bytes	~15–16 digits	$\pm 1.7 \times 10^{308}$
long double	8–16 bytes	platform-dependent	usually ≥ double

⚠ Warning: Floating-point arithmetic is not exact—avoid using it for financial calculations or equality comparisons without tolerance.

3. Variables and Initialization in C++

What Is a Variable?

In C++, a variable is a named storage location in memory that holds a value of a specific data type. Before using a variable, you must declare it, specifying its type and name.

Example:

```
int age;            // declaration
age = 25;           // assignment
```

Why Initialization Matters

Uninitialized variables (especially of built-in types like int, double, etc.) contain garbage values—whatever data happened to be in that memory location. Reading such values leads to undefined behavior, which can cause bugs that are hard to detect.

✅ Always initialize your variables!

3. Variables and Initialization in C++

C++ supports several initialization syntaxes. The most important ones are:

1. Copy Initialization

Uses the = symbol.

```
int x = 42;
double price = 99.99;
```

Simple and familiar.

May involve implicit conversions (e.g., int x = 3.14; truncates to 3).

3. Variables and Initialization in C++

2. Uses parentheses.

```
int x(42);
std::string name("Alice");
```

- Explicit and often used with constructors.
- Allows more control in object creation.

3. Uniform (Brace) Initialization — C++11 and later

Uses curly braces {}.

```
int x{42};
double y{3.14};
std::vector<int> v{1, 2, 3};
```

Also known as list initialization.

3. Variables and Initialization in C++

☑ Advantages:

Prevents narrowing conversions (e.g., int z{3.14}; → compiler error).

Works consistently across all contexts (scalars, arrays, objects).

Avoids the "most vexing parse" problem.

⚠ Example of narrowing prevention:

```
int bad{3.14};  // Error: cannot narrow double
to int
int ok = 3.14;  // Warning (or silent
truncation) — allowed!
```

3. Variables and Initialization in C++

auto and Initialization

The auto keyword deduces the type from the initializer:

```
auto count = 10;         // int
auto price = 19.99;      // double
auto message = "Hello"; // const char*
auto list = {1, 2, 3};  //
std::initializer_list<int>
```

⚠ Be cautious: auto preserves the exact type, including reference or const-ness if used with & or const.

4. Input and Output in C++: Console I/O

C++ uses the Standard Library for input and output operations. The primary tools are:

- `std::cin` — for input (typically from the keyboard)
- `std::cout` — for output (to the console)
- `std::cerr / std::clog` — for error/logging messages

These objects are defined in the <iostream> header.

4. Input and Output in C++: Console I/O

1. Including the Required Header

Always start with:

```
#include <iostream>
```

This makes `std::cin, std::cout`, and related stream objects available.

4. Input and Output in C++: Console I/O

2. Output with `std::cout`

Use the insertion operator << to send data to the console.

Basic Example:

```
#include <iostream>
int main() {
    std::cout << "Hello, World!" << std::endl;
    return 0;
}
```

Key Points:

- You can chain multiple items:

```
std::cout << "The value is: " << 42 << '\n’;
```

- std::endl inserts a newline and flushes the output buffer.
- Prefer '\n' over std::endl unless you explicitly need flushing (for performance reasons).

4. Input and Output in C++: Console I/O

Formatting:

For advanced formatting (e.g., decimal precision), include <iomanip>:

```
#include <iomanip>
std::cout << std::fixed << std::setprecision(2)
<< 3.14159; // prints 3.14
```

4. Input and Output in C++: Console I/O

3. Input with `std::cin`

Use the extraction operator >> to read data from the keyboard.

Basic Example:

```
int age;
std::cout << "Enter your age: ";
std::cin >> age;
```

How It Works:

- `std::cin >>` variable reads formatted input based on the variable’s type.
- Skips leading whitespace (spaces, tabs, newlines).
- Stops reading at the next whitespace or invalid character.

Reading Multiple Values:

```
int x, y;
std::cin >> x >> y;  // e.g., input: "10 20"
```

6. Implicit and Explicit Type Conversions in C++

In C++, type conversion (or type casting) refers to the process of converting a value from one data type to another. These conversions can happen automatically (implicitly) or be explicitly requested by the programmer.

1. Implicit Type Conversion (Coercion)

Implicit conversion is performed automatically by the compiler without any programmer intervention. It occurs when an expression of one type is used in a context that expects a different, but compatible, type.

6. Implicit and Explicit Type Conversions in C++

Common Scenarios:

Assignment:

```
int i = 42;
double d = i;  // int → double (widening — safe)
```

Arithmetic operations:

```
char c = 'A';
int result = c + 10;  // char promoted to int
```

6. Implicit and Explicit Type Conversions in C++

Standard Conversions Include:

- Integral promotions (e.g., char → int)
- Floating-point promotions (float → double)
- Conversions between numeric types (int ↔ double, bool → int, etc.)
- Pointer conversions (e.g., derived class pointer → base class pointer)

6. Implicit and Explicit Type Conversions in C++

Narrowing conversions may lose

```
double pi = 3.14159;
int n = pi;  // n = 3 — fractional part lost!
```

Signed/unsigned mismatches can cause logic errors:

```
if (-1 < 1U) { /* false! */ }
// -1 becomes a large unsigned value
```

🔓 Safety Note: Brace initialization ({}) blocks narrowing implicit conversions:

```
int x{3.14};  // Compile-time error!
```

6. Implicit and Explicit Type Conversions in C++

When you need to override the compiler’s default behavior or perform a conversion that isn’t allowed implicitly, you must use explicit casting.

C++ provides four named cast operators, each with a specific purpose and level of safety.

`static_cast` — General-purpose, compile-time cast

```
double d = 3.99;
int i = static_cast<int>(d);  // i = 3
```

6. Implicit and Explicit Type Conversions in C++

Avoid C-Style Casts

C-style casts look like:

```
int i = (int)3.14;
int j = int(3.14);
```

They are dangerous because they:

- Combine multiple C++ cast types silently
- Bypass compiler safety checks
- Are hard to find in code (no clear keyword)

❌ Avoid them in modern C++. Use named casts instead.

7. Implicit and Explicit Type Conversions in C++

In C++, operators are symbols that perform operations on variables and values. Among the most fundamental are:

- Arithmetic operators – for mathematical calculations
- Logical operators – for combining or inverting Boolean conditions

Understanding these is essential for writing expressions, controlling program flow, and implementing algorithms.

7. Basic Arithmetic and Logical Operations in C++

Arithmetic Operators

C++ supports the standard mathematical operations. They work primarily with numeric types (int, float, double, etc.).

Operator	Name	Example	Result (if a = 10, b = 3)
+	Addition	a + b	13
-	Subtraction	a - b	7
*	Multiplication	a * b	30
/	Division	a / b	3 (integer division)
%	Modulus (remainder)	a % b	1

7. Basic Arithmetic and Logical Operations in C++

Important Notes:

Integer vs. Floating-Point Division

If both operands are integers, / performs truncated integer division:

```
int x = 10 / 3;   // x = 3
```

If at least one operand is floating-point, the result is floating-point:

```
double y = 10.0 / 3;  // y ≈ 3.333...
```

Modulus (%)

- Only works with integer types (not float or double).
- Returns the remainder after division:

```
17 % 5 → 2
-17 % 5 → -2  // sign follows dividend (implementation-
defined in older standards)
```

7. Basic Arithmetic and Logical Operations in C++

Compound Assignment Operators

Combine operation and assignment:

```
a += 5;   // equivalent to a = a + 5;
b *= 2;   // b = b * 2;
c %= 4;   // c = c % 4;
```

Available for: +=, -=, *=, /=, %=

7. Basic Arithmetic and Logical Operations in C++

Increment and Decrement Operators

Special operators to increase or decrease a variable by 1.

Operator	Name	Example	Effect
++x	Pre-increment	++a	Increments, then uses value
x++	Post-increment	a++	Uses value, then increments
--x	Pre-decrement	--a	Decrements, then uses value
x--	Post-decrement	a--	Uses value, then decrements

7. Basic Arithmetic and Logical Operations in C++

Example:

```
int x = 5;
int y = ++x;  // x = 6, y = 6
int z = x++;  // z = 6, x = 7
```

⚠ Avoid complex expressions like a[++i] = i++ — they lead to undefined behavior.

7. Basic Arithmetic and Logical Operations in C++

Logical Operators

Used to combine or invert Boolean expressions (true/false). Operands are typically relational expressions (e.g., x > 5).

Operator	Name	Description	Example
!	Logical NOT	Inverts the Boolean value	!(5 > 3) → false
&&	Logical AND	True if **both** operands are true	(x > 0) && (x < 10)
||	Logical OR	True if at least **one of** operands is true	(x < 0) || (x > 10)

7. Basic Arithmetic and Logical Operations in C++

Short-Circuit Evaluation

C++ uses lazy evaluation:

- For A && B: if A is false, B is not evaluated.
- For A || B: if A is true, B is not evaluated.

This improves performance and enables safe checks.

Operator Precedence

From highest to lowest:

1. ! (NOT)
2. && (AND)
3. || (OR)

Use parentheses to clarify intent:

```
if (!(age < 18) && (hasLicense || isSupervised))
```

7. Basic Arithmetic and Logical Operations in C++

Though not logical themselves, they produce Boolean results used in logical expressions:

Operator	Meaning	Example
==	Equal to	x == y
!=	Not equal to	x != y
<	Less than	x < y
>	Greater than	x > y
<=	Less than or equal	x <= y
>=	Greater than or equal	x >= y

❗ Common mistake: using = (assignment) instead of == (comparison):

```
if (x = 5) { ... }  // Always true! Assigns 5 to x.
```

7. Basic Arithmetic and Logical Operations in C++

Operator Precedence Summary (Relevant Subset)

From highest to lowest priority:

1. ++, -- (postfix)
2. ++, --, ! (prefix)
3. *, /, %
4. +, - (binary)
5. <, <=, >, >=
6. ==, !=
7. &&
8. ||
9. =, +=, -=, etc.

👉 When in doubt, use parentheses!

8. The Conditional (if) Statement in C++

The if statement is a fundamental control structure in C++ that allows your program to make decisions based on conditions. It executes a block of code only if a specified condition evaluates to true.

Basic Syntax

```
if (condition) {
    // code to execute if condition is true
}
```

- condition must be an expression of Boolean type (bool) or convertible to bool (e.g., integers, pointers).
- If the condition is true (non-zero for numeric types), the block inside {} is executed.
- If the condition is false (zero or false), the block is skipped.

8. The Conditional (if) Statement in C++

Example:

```
int age = 20;
if (age >= 18) {
    std::cout << "You are an adult.\n";
}
```

💡 Note: If the block contains only one statement, braces {} are optional—but always recommended for clarity and safety.

8. The Conditional (if) Statement in C++

if-else Statement

Adds an alternative path when the condition is false:

```
if (condition) {
    // executed if condition is true
} else {
    // executed if condition is false
}
```

Example:

```
int temperature = 5;
if (temperature > 20) {
    std::cout << "It's warm.\n";
} else {
    std::cout << "It's cold.\n";
}
```

8. The Conditional (if) Statement in C++

if-else if-else Chain

For multiple mutually exclusive conditions:

```
if (condition1) {
    // ...
} else if (condition2) {
    // ...
} else if (condition3) {
    // ...
} else {
    // default case
}
```

Conditions are checked top to bottom.

Only the first true condition is executed; the rest are skipped.

8. The Conditional (if) Statement in C++

Example:

```
int score = 85;
if (score >= 90) {
    std::cout << "Grade: A\n";
} else if (score >= 80) {
    std::cout << "Grade: B\n";
} else if (score >= 70) {
    std::cout << "Grade: C\n";
} else {
    std::cout << "Grade: F\n";
}
```

⚠ Common mistake: Using multiple independent if statements instead of else if when conditions are mutually exclusive—this can lead to unintended multiple executions.

8. The Conditional (if) Statement in C++

Important Notes

Condition Evaluation

Any non-zero value is treated as true; zero (or nullptr) is false.

```
int x = 5;
if (x) { /* true */ }
if (x - 5) { /* false */ }
```

Scope

Variables declared inside an if block exist only within that block

```
if (true) {
    int temp = 42;
}
// temp is not accessible here
```

8. The Conditional (if) Statement in C++

Dangling else Problem

The else always belongs to the nearest unmatched if

```
if (a)
 if (b)
    foo();
 else
    bar();//this belongs to
          // "if (b)", not "if (a)"
```

→ Use braces to avoid ambiguity.

8. The Conditional (if) Statement in C++

Best Practices

✓ Always use braces {}, even for single-line blocks.

✓ Keep conditions simple; extract complex logic into well-named Boolean variables:

```
bool isEligible = (age >= 18) && (hasLicense);
if (isEligible) { ... }
```

✓ Prefer else if over nested if when checking multiple alternatives.

✓ Avoid side effects in conditions (e.g., if (x = getValue()) — use == for comparison!).

8. The Conditional (if) Statement in C++

Comparison with Ternary Operator

For simple assignments based on a condition, consider the ternary operator:

```
// Instead of:
if (a > b)
    max = a;
else
    max = b;

// Use:
max = (a > b) ? a : b;
```

But for complex logic or multiple statements, stick with if-else.

9. Enumerations and the switch Statement in C++

Enumerations (enum)

An enumeration (or enum) is a user-defined type that consists of a set of named integral constants called enumerators. It improves code readability, maintainability, and type safety by giving meaningful names to related values.

a) Unscoped Enumeration (Traditional enum)

```
enum Color {
    Red,
    Green,
    Blue
};
```

By default, enumerators are assigned integer values starting from 0 (Red = 0, Green = 1, Blue = 2).
You can specify custom values:

```
enum StatusCode {
    OK = 200,
    NotFound = 404,
    ServerError = 500
};
```

9. Enumerations and the switch Statement in C++

Weak typing: enumerators implicitly convert to integers and "leak" into the surrounding scope:

```
Color c = Red;            // OK
int x = Red;              // Allowed (implicit
                          // conversion)
```

⚠ Problem: Name collisions can occur if two enums define the same enumerator name.

9. Enumerations and the switch Statement in C++

b) Scoped Enumeration (enum class) — Preferred in Modern C++

```
enum class Status {
    Pending,
    Approved,
    Rejected
};
```

- Strongly typed: no implicit conversion to integers.
- Scoped: enumerators must be accessed with the enum name

```
Status s = Status::Approved;  // required
// int x = s;                // ERROR: no implicit conversion
```

Underlying type can be specified (default is int):

```
enum class Direction : char {
    North, South, East, West
};
```

✓ Best Practice: Always prefer enum class over plain enum for better type safety and clarity.

9. Enumerations and the switch Statement in C++

The switch statement provides multi-way branching based on the value of an integral or enumeration expression. It is often used with enum types for clean, readable state handling.

Basic Syntax

```
switch (expression) {
    case constant1:
        // code
        break;
    case constant2:
        // code
        break;
    default:
        // optional fallback
}
```

- expression must be of an integral type, enumeration, or a type convertible to one (e.g., char, int, enum).
- Each case label must be a compile-time constant.
- Execution starts at the matching case and continues until a break or the end of the switch.

9. Enumerations and the switch Statement in C++

Example with enum class

```
enum class LogLevel { Debug, Info, Warning, Error };
void log(LogLevel level) {
    switch (level) {
        case LogLevel::Debug:
            std::cout << "[DEBUG] ";  break;
        case LogLevel::Info:
            std::cout << "[INFO] "; break;
        case LogLevel::Warning:
            std::cout << "[WARNING] "; break;
        case LogLevel::Error:
            std::cout << "[ERROR] "; break;
        // No 'default' needed if all cases are covered
    }
}
```

9. Enumerations and the switch Statement in C++

Fallthrough Behavior

If you omit break, execution falls through to the next case:

```
switch (x) {
    case 1:
        std::cout << "One";
        // fallthrough
    case 2:
        std::cout << " or Two\n";
        break;
}
```

Intentional fallthrough should be marked with [[fallthrough]] (C++17):

```
case 1:
    handleOne();
    [[fallthrough]];
case 2:
    handleTwo();
    break;
```

9. Enumerations and the switch Statement in C++

default Case

Acts as a catch-all for unmatched values.

Useful for error handling

or unexpected states:

```
default:
    std::cerr << "Unknown status!\n";
    break;
```

9. Enumerations and the switch Statement in C++

Key Rules & Best Practices

☑ Use enum class with switch — it's type-safe and self-documenting.

☑ Always include break unless fallthrough is intentional (and documented).

☑ Cover all enumerators in switch statements when possible — omitting default can help catch missing cases at compile time.

☑ Avoid switch on raw integers — prefer enums to give meaning to values.

Common Pitfalls

❌ Using unscoped enum with switch → risk of name pollution and accidental conversions.

❌ Forgetting break → unintended fallthrough bugs.

❌ Switching on non-integral types (e.g., std::string, double) → not allowed in C++.

❌ Assuming order or values — always define explicit values if they matter.

10. Loops with a Fixed Number of Iterations in C++

In programming, a fixed-count loop (also called a definite loop) is a loop that executes a known, predetermined number of times. In C++, the for loop is the primary and most idiomatic construct for this purpose.

The Classic for Loop

The traditional for loop has three components: initialization, condition, and iteration expression.

Syntax:

```
for (initialization; condition; iteration) {
    // body — executed repeatedly
}
```

- Initialization: Runs once before the loop starts (e.g., declare and initialize a counter).
- Condition: Checked before each iteration. If true, the loop continues; if false, it stops.
- Iteration: Executed after each loop body (e.g., increment the counter).

10. Loops with a Fixed Number of Iterations in C++

Example: Print numbers 0 to 9

```
for (int i = 0; i < 10; ++i) {
    std::cout << i << " ";
}
// Output: 0 1 2 3 4 5 6 7 8 9
```

✅ This loop runs exactly 10 times — a fixed, predictable count.

Key Features:

The loop counter (i) is typically an integer.

The scope of the counter is limited to the loop (if declared in the initialization part).

You can count forward, backward, or by any step size

```
for (int i = 10; i >= 1; --i) { /* countdown */ }
for (int i = 0; i < 100; i += 10) { /* step by 10 */ }
```

11. Conditional Loops in C++

In C++, conditional loops are loops that continue executing as long as a specified condition remains true. Unlike fixed-count loops (like the classic for loop), the number of iterations in a conditional loop is not known in advance—it depends on runtime conditions.

C++ provides two main conditional loop statements:

while loop

do-while loop

Both are used when you need to repeat an action until a certain logical condition changes.

11. Conditional Loops in C++

The while Loop

The while loop checks the condition before each iteration. If the condition is true, the loop body executes; if false, the loop terminates immediately.

```
while (condition) {
    // loop body
}
```

11. Conditional Loops in C++

Example: Read input until valid

```
int number;
std::cout << "Enter a positive number: ";
std::cin >> number;

while (number <= 0) {
    std::cout << "Invalid! Try again: ";
    std::cin >> number;
}
std::cout << "You entered: " << number << "\n";
```

🔁 This loop may execute zero times if the condition is initially false.

11. Conditional Loops in C++

Common Use Cases:

- Processing data until end-of-file (while (std::cin >> value))
- Waiting for user input to meet criteria
- Simulating events until a state changes

The do-while Loop

The do-while loop checks the condition after each iteration. This guarantees that the loop body executes at least once.

Syntax:

```
do {
    // loop body
} while (condition);
```

⚠ Note the semicolon (;) after while(condition) — it’s required

11. Conditional Loops in C++

Example: Menu-driven program

```
char choice;
do {
    std::cout << "Menu:\n"
              << "A. Option A\n"
              << "B. Option B\n"
              << "Q. Quit\n"
              << "Choose: ";
    std::cin >> choice;
    // Process choice...
} while (choice != 'Q' && choice != 'q’);
```

☑ The menu is displayed at least once, even if the user immediately enters 'Q'.

When to Use do-while:

- When the loop body must run at least once (e.g., input prompts, initialization)
- In interactive programs where user feedback drives repetition

11. Conditional Loops in C++

Comparison with for Loop

Although for loops are often used for fixed counts, they can also implement conditional logic:

```
for (; condition; ) {
    // same as while (condition)
}
```

However, use while for pure condition-based repetition—it’s more readable and idiomatic.

11. Conditional Loops in C++

The break statement immediately terminates the innermost loop or switch statement it appears in. Control is transferred to the statement following the loop or switch.

Usage in Loops

```
for (int i = 0; i < 10; ++i) {
    if (i == 5) {
        break; // exit the loop immediately
    }
    std::cout << i << " ";
}
// Output: 0 1 2 3 4
```

11. Conditional Loops in C++

The continue statement skips the rest of the current iteration and jumps directly to the loop’s update and condition check.

In for Loops

```
for (int i = 0; i < 5; ++i) {
    if (i == 2) {
        continue; // skip printing 2
    }
    std::cout << i << " ";
}
// Output: 0 1 3 4
```

→ After continue, the loop executes ++i and checks i < 5.

11. Conditional Loops in C++

In while and do-while Loops

```
int i = 0;
while (i < 5) {
    ++i;
    if (i == 3) continue;
    std::cout << i << " ";
}
// Output: 1 2 4 5
```

🔁 In while/do-while, ensure the loop variable is updated before continue, or you risk an infinite loop.