C++ Programming Basics Lecture Notes

These notes cover fundamental concepts in C++ programming, including namespaces, data types, input/output operations, type conversions, control flow statements, and functions.

. Namespace Fundamentals

Namespaces are used to organize code into logical groups and to prevent name collisions that can occur when different libraries use the same names for different entities. A namespace defines a scope.

Basic Syntax:

Plain Text

```
namespace MyNamespace {
 // Declarations and definitions
 int myVariable;
 void myFunction();
}
```

Illustrative Example:

Plain Text

```
#include <iostream>

namespace First {
 int x = 10;
}

namespace Second {
 int x = 20;
}

int main() {
 std::cout << "From First Namespace: " << First::x << std::endl;  std::cout << "From Second
Namespace: " << Second::x << std::endl;  return 0;
}
```

. The `using` Directive

The `using` directive simplifies the use of names from a namespace. Instead of repeatedly prefixing names with the namespace name, you can bring them into the current scope.

Basic Syntax:

Plain Text

```
using namespace NamespaceName;
// or
using NamespaceName::name;
```

Illustrative Example:

Plain Text

```
#include <iostream>

namespace MyNamespace {
 void greet() {
 std::cout << "Hello from MyNamespace!" << std::endl;  }
}

int main() {
 using namespace MyNamespace; // Bring all names from MyNamespace into scope
 greet(); // Now you can call greet() directly

 // You can also bring specific names into scope:
 // using MyNamespace::greet;
 // greet();

 return 0;
}
```

. The Standard Namespace (std)

C++ standard library components (like cout , cin , endl) are declared in the std namespace. It's common practice to use using namespace std; for convenience, especially in smaller programs, but in larger projects, it's often preferred to qualify names with std:: to avoid potential name clashes.

Illustrative Example:

Plain Text

```
#include <iostream>

int main() {
 // Using std:: prefix
 std::cout << "Hello, " << std::endl;
 std::cout << "World!" << std::endl;

 // Using directive (common in examples, but be mindful in large projects)  using namespace std;
 cout << "This is easier to type." << endl;

 return 0;
}
```

. Data Types

Data types specify the type of data that a variable can hold, such as integer, floating-point, character, etc. C++ has a rich set of built-in data types and allows for user-defined types.

Common Built-in Data Types:

Type Description Size (typically)

Integer numbers
Single-precision floating-point numbers
Double-precision floating-point numbers
Single characters
Boolean values (true or false)

int bytes float bytes double bytes char byte

bool byte void Represents the absence of type N/A

Illustrative Example:

Plain Text

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

int main() {
 int age = 30;
 double price = 19.99;
 char initial = 'J';
 bool isStudent = true;
 std::string name = "Alice"; // string is part of the standard library, not a built-in type

 std::cout << "Age: " << age << std::endl;
 std::cout << "Price: " << price << std::endl;
 std::cout << "Initial: " << initial << std::endl;
 std::cout << "Is Student: " << isStudent << std::endl;
 std::cout << "Name: " << name << std::endl;

 return 0;
}
```

. Input with cin

The cin object is used to read input from the standard input device (usually the keyboard). It is part of the iostream library.

Basic Syntax:

Plain Text

```
std::cin >> variable;
```

Illustrative Example:

Plain Text

```
#include <iostream>
#include <string>
int main() {
 std::string name;
 int age;

 std::cout << "Enter your name: ";
 std::cin >> name; // Reads a single word

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

 std::cout << "Hello, " << name << "! You are " << age << " years old." << std::endl;

 // To read a line with spaces, use std::getline(std::cin, variable);  std::cin.ignore(); // Clear the
buffer before using getline  std::string full_name;
 std::cout << "Enter your full name: ";
 std::getline(std::cin, full_name);
 std::cout << "Your full name is: " << full_name << std::endl;
 return 0;
}
```

. Output Using cout

The cout object is used to display output to the standard output device (usually the screen). It is also part of the iostream library.

Basic Syntax:

Plain Text

```
std::cout << expression << another_expression << std::endl;
```

Illustrative Example:

Plain Text

```
#include <iostream>

int main() {
 int score = 100;
 double pi = 3.14159;
 char grade = 'A';

 std::cout << "Your score is: " << score << std::endl;
 std::cout << "Value of Pi: " << pi << std::endl;
 std::cout << "Your grade: " << grade << std::endl;
 std::cout << "This is a multi-part output." << " It's easy!" << std::endl;

 return 0;
}
```

. Type Conversion: Automatic Conversions and Casts

Type conversion is the process of converting a value from one data type to another. This can happen automatically (implicit conversion) or explicitly (explicit conversion or casting).

Automatic Conversions (Implicit Type Conversion)

Automatic conversions occur when the compiler automatically converts one data type to another without any explicit instruction from the programmer. This usually happens when a smaller data type is assigned to a larger data type (promotion) or during arithmetic operations.

Illustrative Example:

Plain Text

```
#include <iostream>

int main() {
 int intValue = 10;
 double doubleValue = 5.5;

 // Implicit conversion: int to double
 double result1 = intValue + doubleValue; // intValue is promoted to double
 std::cout << "int + double (result is double): " << result1 << std::endl; // Output: 15.5

 // Implicit conversion: char to int
 char charValue = 'A'; // ASCII value of 'A' is 65
 int asciiValue = charValue;
                                                std::cout << "char to int: " << asciiValue << std::endl; // Output: 65

 return 0;
}
```

Casts (Explicit Type Conversion)

Casts are explicit instructions to the compiler to convert a value from one data type to another. This is often necessary when converting a larger data type to a smaller one (demotion), which might result in data loss.

Basic Syntax:

Plain Text

```
 (type)expression; // C-style cast
static_cast<type>(expression); // C++ style cast (recommended)
```

Illustrative Example:

Plain Text

```
#include <iostream>

int main() {
 double doubleValue = 10.75;
 // Explicit conversion: double to int using C-style cast  int intResultC =
(int)doubleValue;
 std::cout << "double to int (C-style cast): " << intResultC << std::endl; // Output: 10

 // Explicit conversion: double to int using static_cast (recommended)  int intResultCpp =
static_cast<int>(doubleValue);
 std::cout << "double to int (static_cast): " << intResultCpp << std::endl; // Output: 10

 return 0;
}
```

. Loops and Decision-Making Statements

Control flow statements determine the order in which instructions are executed. Loops allow a block of code to be executed repeatedly, while decision-making statements allow different blocks of code to be executed based on certain conditions.

Decision-Making Statements

if-else Statement

The if-else statement executes a block of code if a specified condition is true, and another block if the condition is false.

Basic Syntax:

Plain Text

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

switch Statement

The switch statement allows a variable to be tested for equality against a list of values. Each value is called a case, and the variable being switched on is checked for each switch case.

Basic Syntax:

Plain Text

```
switch (expression) {
 case value1:
 // code to be executed if expression equals value1
 break;
 case value2:
 // code to be executed if expression equals value2
 break;
 default:
 // code to be executed if expression doesn't match any case }
```

Loops

`for` Loop

The `for` loop is used to iterate a block of code a certain number of times.

Basic Syntax:

Plain Text

```
for (initialization; condition; update) {
 // code to be executed repeatedly
}
```

`while` Loop

The `while` loop executes a block of code as long as a specified condition is true.

Basic Syntax:

Plain Text

```
while (condition) {
 // code to be executed repeatedly
}
```

`do-while` Loop

The `do-while` loop is similar to the `while` loop, but it guarantees that the block of code is executed at least once, before checking the condition.

Basic Syntax:

Plain Text

```
do {
 // code to be executed repeatedly
} while (condition);
```

Illustrative Example (from loops_decision_making.cpp):

Plain Text

```
#include <iostream>

int main() {
 // Decision Making: if-else statement
 int num = 10;
 if (num > 0) {
 std::cout << "Number is positive." << std::endl;  } else {
 std::cout << "Number is non-positive." << std::endl;  }

 // Decision Making: switch statement
 char grade = 'B';
 switch (grade) {
 case 'A':
 std::cout << "Excellent!" << std::endl;
 break;
 case 'B':
 std::cout << "Good!" << std::endl;
 break;
 case 'C':
 std::cout << "Fair!" << std::endl;
 break;
 default:
 std::cout << "Needs improvement." << std::endl;  }

 // Loop: for loop
 std::cout << "\nFor loop from 1 to 5:" << std::endl;  for (int i = 1; i <= 5; ++i) {
 std::cout << i << " ";
 }
 std::cout << std::endl;

 // Loop: while loop
 std::cout << "\nWhile loop from 5 to 1:" << std::endl;  int i = 5;
 while (i >= 1) {
 std::cout << i << " ";
 --i;
 }
 std::cout << std::endl;

 // Loop: do-while loop
 std::cout << "\nDo-while loop (executes at least once):" << std::endl;  int j = 0;
 do {
 std::cout << "Value of j: " << j << std::endl;
 j++;
 } while (j < 1);

 return 0;
}
```

. Functions

Functions are blocks of code that perform a specific task. They help in organizing code, making it more modular, reusable, and easier to understand. Functions can take parameters (inputs) and return a value (output).

Basic Syntax:

Plain Text

```
return_type function_name(parameter_list) {
 // function body
 // ...
 return value; // if return_type is not void
}
```

Illustrative Example (from `functions.cpp`):

Plain Text

```
#include <iostream>

// Function declaration (prototype)
void greet();

// Function definition
void greet() {
 std::cout << "Hello from a function!" << std::endl;
}

// Function with parameters
int add(int a, int b) {
 return a + b;
}
// Function with default arguments
void printMessage(std::string message = "Default message") {  std::cout <<
message << std::endl;
}

int main() {
 greet(); // Calling the greet function

 int sum = add(5, 3); // Calling add function with arguments  std::cout << "Sum: " <<
sum << std::endl;

 printMessage(); // Calling with default argument
 printMessage("Custom message"); // Calling with custom argument

 return 0;
}
```

. Function Overloading

Function overloading allows you to define multiple functions with the same name but different parameters. The compiler determines which function to call based on the number and types of arguments passed during the function call.

Illustrative Example (from function_overloading.cpp):

Plain Text

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

// Function to add two integers
int add(int a, int b) {
 return a + b;
}

// Function to add two doubles (overloaded function)
double add(double a, double b) {
 return a + b;
}

// Function to concatenate two strings (overloaded function) std::string
add(std::string s1, std::string s2) {
 return s1 + s2;
}

int main() {
 // Call the integer add function
 std::cout << "Sum of integers: " << add(5, 10) << std::endl;

 // Call the double add function
 std::cout << "Sum of doubles: " << add(5.5, 10.5) << std::endl;

 // Call the string add function
 std::cout << "Concatenated string: " << add("Hello, ", "World!") << std::endl;

 return 0;
}
```