**COMPILE PROCESS**

Compilation in C++ involves converting source code written in the C++ language into machine code that a computer's CPU can execute. This process generally happens in several steps:

Preprocessing: Remove comments, include header files, and expand macros.

Compilation: Convert the preprocessed source code into assembly language.

Assembly: Convert the assembly code into machine code, resulting in an object file.

Linking: Combine object files and external libraries to create the final executable.

**SYNTAX REMINDERS**

switch(variable) {

case value1:

// code to run if variable == value1

break;

case value2:

// code to run if variable == value2

break;

default:

// code to run if no case matches

}

do {

// code to be repeated

} while(condition);

**FLOATING POINT COMPARISON**

Due to the way floating-point numbers are represented in computers, direct comparison can be inaccurate. It's usually better to compare if the difference between two floating-point numbers is smaller than a tiny value (epsilon).

double a = 0.3;

double b = 0.1 + 0.2;

const double epsilon = 1e-9;

if (std::abs(a - b) < epsilon) {

std::cout << "a and b are almost equal!" << std::endl;

}

In the above code, std::abs returns the absolute value of the difference between a and b. If that difference is less than our defined epsilon, we consider a and b to be effectively equal.

**POINTERS**

*1. Declaring Pointers*

To declare a pointer, you use the asterisk (\*) operator. The type of the pointer corresponds to the type of variable its pointing to.

int \*ptr;

Here, ptr is a pointer to an integer.

*2. Initializing Pointers*

You can initialize a pointer using the address of a variable. The & operator is used to get the address of a variable.

int x = 10;

int \*ptr = &x;

Here, ptr now holds the address of x.

*3. Accessing Value Using Pointers*

The \* operator is a dereference operator, and it's used to access the value at the address held by the pointer.

int x = 10;

int \*ptr = &x;

std::cout << \*ptr; // Outputs: 10

*4. Pointer Arithmetic*

You can perform arithmetic operations on pointers. This is particularly useful when dealing with arrays.

int arr[5] = {1, 2, 3, 4, 5};

int \*ptr = arr; // Pointing to the first element

ptr++; // Move to the next element

std::cout << \*ptr; // Outputs: 2

*5. Pointers with Other Data Types*

You can have pointers for all data types:

double \*dPtr;

char \*cPtr;

*6. Pointer to Pointer*

A pointer can also store the address of another pointer.

int x = 10;

int \*ptr1 = &x;

int \*\*ptr2 = &ptr1; // Pointer to an integer pointer

*7. Null Pointer*

It's good practice to initialize pointers if they aren't immediately assigned a value. A null pointer does not point to any memory location.

int \*ptr = nullptr; // Using C++11 and later

*8. Dynamic Memory Allocation*

Pointers can be used with C++'s dynamic memory allocation functions: new and delete.

int \*ptr = new int; // Allocate memory for an integer

\*ptr = 10; // Assign value

delete ptr; // Free the allocated memory

*9. Arrays and Pointers*

Arrays in C++ are closely related to pointers. The name of the array is a pointer to the first element.

int arr[3] = {1, 2, 3};

int \*ptr = arr; // Pointing to arr[0]

*10. Pointers to Functions*

You can have pointers to functions, enabling dynamic dispatch of functions.

void myFunction() {

std::cout << "Hello, World!" << std::endl;

}

int main() {

void (\*functionPtr)() = &myFunction;

functionPtr(); // Calls myFunction

}

*11. Pointer Issues*

Dangling Pointer: A pointer pointing to a memory location that has been deleted or de-allocated.

Memory Leaks: Not releasing allocated memory (using delete or delete[]).

Wild Pointers: Pointers that haven't been initialized.

Using pointers requires careful management to avoid these issues. Modern C++ introduces smart pointers (std::unique\_ptr, std::shared\_ptr) that help manage memory more safely.

**STD::STRING**

*1. Declaration and Initialization*

#include <iostream>

#include <string>

int main() {

std::string str1; // Empty string

std::string str2 = "Hello"; // Direct initialization

std::string str3(str2); // Copy initialization

std::string str4(5, 'a'); // Initializes str4 with "aaaaa"

std::cout << str4 << std::endl; // Outputs: aaaaa

}

*2. Access*

Access individual characters: You can use the at() method or the subscript operator [].

char ch1 = str2.at(0); // 'H'

char ch2 = str2[0]; // 'H'

Access first and last characters: The front() and back() methods give access to the first and last characters respectively.

char first = str2.front(); // 'H'

char last = str2.back(); // 'o'

*3. Modification*

Append to string: You can use the += operator or the append() method.

str1 += "World!";

str2.append(" World!");

Insert into string: The insert() method allows insertion at a specific position.

str1.insert(5, " dear");

Erase from string: The erase() method removes characters from a string.

str1.erase(5, 5); // Removes 5 characters starting from position 5

Replace characters: The replace() method substitutes a portion of the string with another string.

str1.replace(0, 5, "Hi");

*4. String Comparison*

The std::string class overloads the comparison operators (==, !=, <, >, <=, >=).

if (str1 == str2) {

std::cout << "Strings are equal." << std::endl;

}

Additionally, the compare() method can be used:

if (str1.compare(str2) == 0) { // Returns 0 if the strings are equal

std::cout << "Strings are equal." << std::endl;

}

*5. Character Operations*

Find character or substring: You can use the find() method.

size\_t pos = str1.find("World");

if (pos != std::string::npos) {

std::cout << "Found at position: " << pos << std::endl;

}

Find last occurrence: The rfind() method can be used.

size\_t lastPos = str1.rfind('o');

Iterate over characters: You can use a simple loop or range-based for loop.

for (char ch : str1) {

std::cout << ch << " ";

}

*6. Other Useful Methods*

String length: Use the length() or size() method.

size\_t len = str1.length();

Check if string is empty: Use the empty() method.

if (str1.empty()) {

std::cout << "String is empty." << std::endl;

}

Convert to C-style string: Use the c\_str() method.

const char\* cstr = str1.c\_str();

**EXCEPTIONS**

Error Handling: One of the main purposes of exceptions is to manage error situations. These situations could arise from program logic issues, invalid inputs, file not found, network failures, etc.

Separation of Concerns: Exceptions allow for separating the error-handling code from the main logic of the program, which can make the code cleaner and easier to maintain.

Propagation: When an error occurs deep within several nested function calls, exceptions provide a way to "bubble up" and handle the error at a higher level.

Standardized: Exceptions offer a standardized way to handle errors. Libraries and frameworks use them, enabling you to handle errors in a consistent manner.

*1. Basics of C++ Exceptions*

throw keyword: This keyword is used to raise an exception. You can throw any type of data: built-in, objects, arrays, etc., but it's common to throw objects.

throw "Division by zero condition!"; // throwing a string constant

try and catch blocks: The try block encloses the code that might raise an exception, and the catch block defines what to do if an exception occurs.

try {

// code that might throw exceptions

if (denominator == 0) {

throw "Division by zero!";

}

result = numerator / denominator;

} catch (const char\* e) {

std::cerr << "Error: " << e << std::endl;

}

*2. Best Practices*

Only Handle Exceptions You Can Handle: If you can't meaningfully handle an exception at a given level, it's often best to let it propagate up to a higher level where it can be dealt with appropriately.

Use Exceptions for Error Handling, Not Control Flow: Exceptions should be used for exceptional situations, not as a regular control flow mechanism.

Clean Up: If an exception might be thrown, be sure any resources acquired (like memory or file handles) are properly released.

**FUNCTIONS**

*Function Signature*: A function's signature is determined by its name and the types of its parameters (not by its return type). For example:

int add(int a, int b);

Here, the function signature is add(int, int).

*Formal Parameters*: These are variables that are used in the function declaration and act as placeholders for the values they receive. In the example above, a and b are formal parameters.

Actual Arguments: When you call a function, you pass values to it, and these values are referred to as actual arguments. These arguments can be constants, variables, or expressions.

int x = 5, y = 3;

int result = add(x, y); // Here, x and y are actual arguments

*Scope* refers to the visibility and lifetime of a variable or function:

Local Scope: Variables declared inside a function have local scope. They are created when the function is called and destroyed when the function exits.

Global Scope: Variables declared outside any function have global scope. They are accessible from any function in the file.

Static Local Variables: Variables declared as static inside a function retain their value between function calls.

*Declarations vs. Definitions*

Declaration: It tells the compiler about the function's name, return type, and parameters. A function can be declared multiple times.

Definition: It provides the actual body of the function. A function must be defined only once.

Note: In C++, it's common to declare functions in header files (.h or .hpp) and define them in source files (.cpp).

**INPUT OUTPUT STRING STREAM**

In C++, the header <sstream> provides string stream classes that facilitate in-memory string parsing and formatting operations. Essentially, they allow strings to be treated as streams, enabling you to perform input/output operations on them as you would with files.

*1. std::istringstream (Input String Stream)*

This class is used for reading (input) operations on strings.

Initialization:

std::string str = "123 456";

std::istringstream iss(str);

Reading from istringstream:

int a, b;

iss >> a >> b; // a will be 123 and b will be 456

Parsing a line of mixed types:

std::string mixed = "John 25";

std::istringstream mixstream(mixed);

std::string name;

int age;

mixstream >> name >> age; // name will be "John", age will be 25

*2. std::ostringstream (Output String Stream)*

This class is used for writing (output) operations to strings.

Initialization: std::ostringstream oss;

Writing to ostringstream:

oss << "Hello, " << "World!" << 123;

std::string output = oss.str(); // output will be "Hello, World!123"

Formatting strings:

int x = 42;

double y = 3.14;

oss << "int: " << x << ", double: " << y;

std::string formatted = oss.str(); // formatted will be "int: 42, double: 3.14"

*3. Common Member Functions*

.str(): Returns the contents of the string stream as a string.

.str(const std::string &s): Sets the contents of the string stream to the given string.

.clear(): Clears the string stream's error flags, it doesn’t clear the content.

.seekg() and .seekp(): Used to set the position of the get (read) and put (write) pointers, respectively.

**LIMITS**

The <limits> header in C++ is a part of the C++ standard library. It defines the std::numeric\_limits template class, allowing programmers to obtain properties of arithmetic types (such as int, float, double, etc.) like their minimum and maximum possible values, and more.

#include <iostream>

#include <limits>

int main() {

std::cout << "Max int value: " << std::numeric\_limits<int>::max() << std::endl;

std::cout << "Min int value: " << std::numeric\_limits<int>::min() << std::endl;

std::cout << "Max float value: " << std::numeric\_limits<float>::max() << std::endl;

std::cout << "Min positive float value: " << std::numeric\_limits<float>::min() << std::endl;

}

**C STRINGS**

In C, strings are essentially arrays of characters terminated by a null character (\0). They are different from C++ std::string objects, which are part of the C++ Standard Library.

*1. Declaring and Initializing C Strings:*

Using Character Arrays:

char str[6] = "hello"; // Explicitly allocating size

char str[] = "hello"; // Letting compiler determine size

*1.2 Using Pointers:*

const char\* str = "hello";

Remember, while you can modify characters in a character array, you should not modify the characters pointed to by a string literal through a pointer, as it can lead to undefined behavior.

*2. Accessing Characters:*

You can access individual characters of a string like you would with any array:

char firstChar = str[0]; // 'h'

*3. String Functions in <cstring>:*

C provides a range of functions to manipulate strings which are available in the <cstring> header (previously <string.h> in older code):

strlen: Gets the length of the string.

size\_t length = strlen(str);

strcpy: Copies one string to another.

char dest[10];

strcpy(dest, str);

strcat: Concatenates two strings.

char src[10] = "world";

char dest[20] = "hello ";

strcat(dest, src); // dest becomes "hello world"

strcmp: Compares two strings. Returns 0 if equal, negative if str1 < str2, and positive if str1 > str2.

int result = strcmp(str1, str2);

*4. Common Pitfalls:*

Buffer Overflows: One of the most frequent issues with C strings is buffer overflow. Always ensure there's enough space in your arrays to accommodate characters plus the null terminator.

char buffer[5];

strcpy(buffer, "hello"); // Error: No space for null terminator

String Literals are Immutable: As mentioned earlier, attempting to modify a string literal through a pointer can lead to undefined behavior.

const char\* str = "hello";

str[0] = 'H'; // Undefined behavior

*5. Differences from C++ std::string:*

Memory Management: With C strings, memory management (like allocation and deallocation) is manual. With std::string, it's handled automatically.

Operations: std::string supports many inbuilt operations like + for concatenation, == for comparison, etc., making them easier to use.

Safety: std::string is generally safer as it manages its own memory and avoids issues like buffer overflows, which are common pitfalls with C strings.

**TYPE CASTING**

*1. Implicit Conversion (Automatic Type Conversion)*

This is when the compiler automatically converts one data type into another without the programmer's intervention. This usually happens in mixed-type expressions.

int i = 10;

double d = i; // i is automatically converted to double

Rules:

Small data types can be converted to larger data types (e.g., int to double).

Non-const can be converted to const.

If there's a risk of data loss, or when converting from larger to smaller types, C++ may give a warning.

*2. Explicit Conversion (Type Casting)*

C-style Cast:

double d = 10.5;

int i = (int)d; // d is explicitly converted to int

C++ Casts:

C++ introduces four named cast operators which offer better clarity and control compared to C-style casts.

static\_cast: Used for most common type-to-type conversions.

double d = 10.5;

int i = static\_cast<int>(d);

dynamic\_cast: Mainly used with pointers/references of polymorphic types.

Base\* bPtr = new Derived();

Derived\* dPtr = dynamic\_cast<Derived\*>(bPtr);

const\_cast: Used to add/remove const qualification.

const int a = 10;

int\* ptr = const\_cast<int\*>(&a);

*3. Standard Conversion Functions*

The C++ Standard Library provides several functions/templates for conversions, especially between fundamental and string types:

std::stoi, std::stod, ...: Convert string to int, double, etc.

std::string s = "123";

int i = std::stoi(s); // i = 123 as an integer

std::to\_string: Convert fundamental types to strings.

int i = 123;

std::string s = std::to\_string(i);

**ASCII**

ASCII is a character encoding standard that represents each character as a number. In the ASCII standard, every character is assigned a unique 7-bit integer between 0 and 127. For instance, the ASCII value of 'A' is 65, and the ASCII value of 'a' is 97.

Get ASCII value of a character:

char ch = 'A';

int asciiValue = static\_cast<int>(ch);

std::cout << "ASCII value of " << ch << ": " << asciiValue << std::endl; // Outputs: ASCII value of A: 65

Convert ASCII value to a character:

int asciiVal = 97;

char character = static\_cast<char>(asciiVal);

std::cout << "Character for ASCII " << asciiVal << ": " << character << std::endl; // Outputs: Character for ASCII 97: a

Iterate over printable ASCII characters:

for (int i = 32; i < 127; i++) {

std::cout << i << ": " << static\_cast<char>(i) << " ";

}

Converting lowercase to uppercase (and vice versa) by leveraging their ASCII difference:

char lower = 'c';

char upper = lower - ('a' - 'A');

std::cout << "Uppercase of " << lower << " is " << upper << std::endl; // Outputs: Uppercase of c is C

**PARSING NUMBERS**

*1. Understand the Basics of long long Manipulation*

The modulo (%) operation can help us get the last digit of a number, and division (/) by 10 can help us remove the last digit. For example:

12345 % 10 gives 5 (the last digit)

12345 / 10 gives 1234 (the number without its last digit)

*2. Filter Out Only Odd Digits*

We can use the modulo operation to determine if a digit is odd. If digit % 2 is not zero, then the digit is odd.

*3. Constructing the New Number*

As we extract and filter digits from the input, we need to build our new number. Starting with an initial result of 0, for every new odd digit we find, the result can be updated using multiplication and addition:

Multiply the current result by 10 (this "shifts" the number to the left)

Add the new odd digit

*4. Iterate until the Input is Zero*

We'll repeat the above processes in a loop until our input number is reduced to 0.

*Example:*long long extractOddDigits(long long num) {

long long result = 0;

while (num != 0) {

long long digit = num % 10; // Get the last digit

if (digit % 2 != 0) { // Check if the digit is odd

result = result \* 10 + digit; // Construct the new number

}

num /= 10; // Remove the last digit

}

// Reversing the result as our approach constructs the number in reverse

long long finalResult = 0;

while (result != 0) {

finalResult = finalResult \* 10 + result % 10;

result /= 10;

}

return finalResult;

}