C++ References & CheatSheets

C++ References & CheatSheets

C++ Syntax Cheat Sheet

Table of Contents

Basic Program Structure

```
#include <iostream>  // Header file inclusion
using namespace std; // Optional namespace declaration

int main() {         // Main function - program entry point
    cout << "Hello, World!" << endl;
    return 0;        // Return statement
}
```

💡 **Tip:** Avoid `using namespace std;` in header files or large projects to prevent namespace pollution.

Data Types

Fundamental Types

```
// Integer types
int age = 25;           // 4 bytes (typically)
short year = 2024;      // 2 bytes
long population = 8000000000L;  // 4-8 bytes
long long bigNumber = 123456789LL;  // 8 bytes

// Floating-point types
float pi = 3.14f;       // 4 bytes, 'f' suffix
double precision = 3.141592653589793;  // 8 bytes
long double extended = 3.1415926535897932L;  // 10-16 bytes

// Character types
char grade = 'A';       // 1 byte
wchar_t unicode = L'π'; // Wide character

// Boolean type
bool isActive = true;   // 1 byte (true/false)
```

Type Modifiers

```
unsigned int positiveOnly = 100u;
signed int canBeNegative = -50;
const int CONSTANT_VALUE = 42;     // Cannot be modified
volatile int hardwareRegister;     // Can change unexpectedly
```

Variables and Constants

Variable Declaration and Initialization

```
// Declaration
int number;

// Initialization
int count = 0;
double rate{5.5};      // Uniform initialization (C++11)
auto result = 10.5;    // Type deduction (C++11)

// Multiple variables
int x = 1, y = 2, z = 3;
```

Constants

```cpp
// Compile-time constants
const int MAX_SIZE = 100;
constexpr double PI = 3.14159;  // C++11

// Runtime constants
const int userInput = getUserInput();

// Preprocessor constants (avoid in modern C++)
#define OLD_STYLE_CONSTANT 42
```

Operators

Arithmetic Operators

```cpp
int a = 10, b = 3;

int sum = a + b;        // Addition: 13
int diff = a - b;       // Subtraction: 7
int product = a * b;    // Multiplication: 30
int quotient = a / b;   // Division: 3 (integer division)
int remainder = a % b;  // Modulus: 1

// Compound assignment
a += 5;  // a = a + 5
a -= 2;  // a = a - 2
a *= 3;  // a = a * 3
a /= 2;  // a = a / 2

// Increment/Decrement
++a;     // Pre-increment
a++;     // Post-increment
--b;     // Pre-decrement
b--;     // Post-decrement
```

Comparison Operators

```cpp
bool equal = (a == b);      // Equal to
bool notEqual = (a != b);   // Not equal to
bool greater = (a > b);     // Greater than
bool less = (a < b);        // Less than
```

```
bool greaterEq = (a >= b);  // Greater than or equal
bool lessEq = (a <= b);     // Less than or equal
```

Logical Operators

```
bool result1 = (a > 5) && (b < 10);  // Logical AND
bool result2 = (a < 5) || (b > 10);  // Logical OR
bool result3 = !(a == b);            // Logical NOT
```

Bitwise Operators

```
int x = 5;  // Binary: 101
int y = 3;  // Binary: 011

int bitwiseAnd = x & y;   // 001 = 1
int bitwiseOr = x | y;    // 111 = 7
int bitwiseXor = x ^ y;   // 110 = 6
int bitwiseNot = ~x;      // Complement
int leftShift = x << 1;   // 1010 = 10
int rightShift = x >> 1;  // 010 = 2
```

Control Structures

Conditional Statements

```
// if-else statement
if (score >= 90) {
    grade = 'A';
} else if (score >= 80) {
    grade = 'B';
} else if (score >= 70) {
    grade = 'C';
} else {
    grade = 'F';
}

// Ternary operator
string result = (score >= 60) ? "Pass" : "Fail";

// switch statement
switch (grade) {
    case 'A':
        cout << "Excellent!";
```

```
        break;
    case 'B':
        cout << "Good!";
        break;
    case 'C':
        cout << "Average";
        break;
    default:
        cout << "Needs improvement";
        break;
}
```

Loops

```
// for loop
for (int i = 0; i < 10; i++) {
    cout << i << " ";
}

// Range-based for loop (C++11)
vector<int> numbers = {1, 2, 3, 4, 5};
for (const auto& num : numbers) {
    cout << num << " ";
}

// while loop
int count = 0;
while (count < 5) {
    cout << count << endl;
    count++;
}

// do-while loop
int input;
do {
    cout << "Enter a positive number: ";
    cin >> input;
} while (input <= 0);
```

Loop Control

```
for (int i = 0; i < 10; i++) {
    if (i == 3) continue;    // Skip iteration
    if (i == 8) break;       // Exit loop
```

```
    cout << i << " ";
}
```

Functions

Function Declaration and Definition

```
// Function declaration (prototype)
int add(int a, int b);
void printMessage(const string& message);

// Function definition
int add(int a, int b) {
    return a + b;
}

void printMessage(const string& message) {
    cout << message << endl;
}

// Inline function
inline int square(int x) {
    return x * x;
}
```

Function Overloading

```
int multiply(int a, int b) {
    return a * b;
}

double multiply(double a, double b) {
    return a * b;
}

int multiply(int a, int b, int c) {
    return a * b * c;
}
```

Default Parameters

```
void greet(const string& name, const string& greeting = "Hello") {
    cout << greeting << ", " << name << "!" << endl;
```

```
}

// Usage
greet("Alice");           // Uses default greeting
greet("Bob", "Hi");       // Uses custom greeting
```

Lambda Functions (C++11)

```
// Basic lambda
auto lambda = [](int x, int y) { return x + y; };
int result = lambda(5, 3);

// Lambda with capture
int multiplier = 10;
auto multiply = [multiplier](int x) { return x * multiplier; };

// Capture by reference
auto increment = [&multiplier]() { multiplier++; };
```

Arrays

Static Arrays

```
// Declaration and initialization
int numbers[5] = {1, 2, 3, 4, 5};
int grades[10];  // Uninitialized

// Accessing elements
numbers[0] = 10;
int first = numbers[0];

// Multidimensional arrays
int matrix[3][4] = {
    {1, 2, 3, 4},
    {5, 6, 7, 8},
    {9, 10, 11, 12}
};
```

Dynamic Arrays (C-style)

```
int size = 10;
int* dynamicArray = new int[size];
```

```cpp
// Use the array
for (int i = 0; i < size; i++) {
    dynamicArray[i] = i * 2;
}

// Don't forget to free memory
delete[] dynamicArray;
```

STL Arrays and Vectors

```cpp
#include <array>
#include <vector>

// std::array (fixed size)
array<int, 5> arr = {1, 2, 3, 4, 5};

// std::vector (dynamic size)
vector<int> vec = {1, 2, 3, 4, 5};
vec.push_back(6);        // Add element
vec.pop_back();          // Remove last element
int size = vec.size();   // Get size
```

Pointers and References

Pointers

```cpp
int value = 42;
int* ptr = &value;       // Pointer declaration and initialization

cout << ptr;             // Address of value
cout << *ptr;            // Dereference - prints 42

// Null pointers
int* nullPtr = nullptr;  // C++11 (preferred)
int* oldNull = NULL;     // C-style (avoid)

// Pointer arithmetic
int arr[] = {1, 2, 3, 4, 5};
int* p = arr;
p++;                     // Points to next element
```

References

```cpp
int original = 100;
int& ref = original;     // Reference must be initialized

ref = 200;               // Changes original to 200
cout << original;        // Prints 200

// Function parameters
void increment(int& value) {
    value++;
}

increment(original);     // original is now 201
```

Smart Pointers (C++11)

```cpp
#include <memory>

// unique_ptr - exclusive ownership
unique_ptr<int> uptr = make_unique<int>(42);

// shared_ptr - shared ownership
shared_ptr<int> sptr = make_shared<int>(42);
shared_ptr<int> sptr2 = sptr;  // Shared ownership

// weak_ptr - non-owning observer
weak_ptr<int> wptr = sptr;
```

Classes and Objects

Basic Class Structure

```cpp
class Rectangle {
private:
    double width, height;

public:
    // Constructor
    Rectangle(double w = 0, double h = 0) : width(w), height(h) {}

    // Destructor
    ~Rectangle() {
        // Cleanup code
    }
```

```
    // Member functions
    double getArea() const {
        return width * height;
    }

    void setDimensions(double w, double h) {
        width = w;
        height = h;
    }

    // Getter and setter
    double getWidth() const { return width; }
    void setWidth(double w) { width = w; }
};

// Usage
Rectangle rect(5.0, 3.0);
double area = rect.getArea();
```

Inheritance

```
class Shape {
protected:
    string color;

public:
    Shape(const string& c) : color(c) {}
    virtual double getArea() const = 0;  // Pure virtual function
    virtual ~Shape() = default;          // Virtual destructor
};

class Circle : public Shape {
private:
    double radius;

public:
    Circle(double r, const string& c) : Shape(c), radius(r) {}

    double getArea() const override {
        return 3.14159 * radius * radius;
    }
};
```

Polymorphism

```
// Base class pointer to derived object
Shape* shape = new Circle(5.0, "red");
double area = shape->getArea();  // Calls Circle's getArea()

// Virtual function table ensures correct function is called
delete shape;
```

Standard Template Library (STL)

Containers

```
#include <vector>
#include <list>
#include <map>
#include <set>
#include <queue>
#include <stack>

// Vector - dynamic array
vector<int> vec = {1, 2, 3, 4, 5};

// List - doubly linked list
list<string> names = {"Alice", "Bob", "Charlie"};

// Map - key-value pairs
map<string, int> ages;
ages["Alice"] = 25;
ages["Bob"] = 30;

// Set - unique elements
set<int> uniqueNumbers = {1, 2, 3, 3, 4};  // Only stores {1, 2, 3, 4}

// Queue and Stack
queue<int> q;
stack<int> s;
```

Iterators

```
vector<int> numbers = {1, 2, 3, 4, 5};

// Iterator types
```

```cpp
vector<int>::iterator it;
auto it2 = numbers.begin();  // C++11 auto

// Traversing with iterators
for (auto it = numbers.begin(); it != numbers.end(); ++it) {
    cout << *it << " ";
}

// Range-based for loop (preferred)
for (const auto& num : numbers) {
    cout << num << " ";
}
```

Algorithms

```cpp
#include <algorithm>

vector<int> vec = {5, 2, 8, 1, 9};

// Sorting
sort(vec.begin(), vec.end());

// Finding
auto it = find(vec.begin(), vec.end(), 8);
if (it != vec.end()) {
    cout << "Found at position: " << distance(vec.begin(), it);
}

// Other useful algorithms
reverse(vec.begin(), vec.end());
random_shuffle(vec.begin(), vec.end());
```

Memory Management

Dynamic Memory Allocation

```cpp
// Single object
int* ptr = new int(42);
delete ptr;

// Array
int* arr = new int[10];
delete[] arr;
```

```
// Always match new with delete, new[] with delete[]
```

RAII (Resource Acquisition Is Initialization)

```
class FileManager {
private:
    FILE* file;

public:
    FileManager(const string& filename) {
        file = fopen(filename.c_str(), "r");
        if (!file) throw runtime_error("Cannot open file");
    }

    ~FileManager() {
        if (file) fclose(file);
    }

    // Prevent copying to avoid double-free
    FileManager(const FileManager&) = delete;
    FileManager& operator=(const FileManager&) = delete;
};
```

Exception Handling

Try-Catch Blocks

```
#include <stdexcept>

try {
    int divisor = 0;
    if (divisor == 0) {
        throw runtime_error("Division by zero!");
    }
    int result = 10 / divisor;
}
catch (const runtime_error& e) {
    cout << "Runtime error: " << e.what() << endl;
}
catch (const exception& e) {
    cout << "General exception: " << e.what() << endl;
}
catch ( ... ) {
```

```
    cout << "Unknown exception caught" << endl;
}
```

Custom Exceptions

```
class CustomException : public exception {
private:
    string message;

public:
    CustomException(const string& msg) : message(msg) {}

    const char* what() const noexcept override {
        return message.c_str();
    }
};

// Usage
throw CustomException("Something went wrong!");
```

File I/O

Basic File Operations

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

// Writing to file
ofstream outFile("output.txt");
if (outFile.is_open()) {
    outFile << "Hello, File!" << endl;
    outFile << "Line 2" << endl;
    outFile.close();
}

// Reading from file
ifstream inFile("input.txt");
string line;
if (inFile.is_open()) {
    while (getline(inFile, line)) {
        cout << line << endl;
    }
    inFile.close();
```

```
}

// Read/Write file
fstream file("data.txt", ios::in | ios::out | ios::app);
```

Binary File I/O

```
// Writing binary data
ofstream binFile("data.bin", ios::binary);
int numbers[] = {1, 2, 3, 4, 5};
binFile.write(reinterpret_cast<char*>(numbers), sizeof(numbers));
binFile.close();

// Reading binary data
ifstream binInput("data.bin", ios::binary);
int readNumbers[5];
binInput.read(reinterpret_cast<char*>(readNumbers), sizeof(readNumbers));
binInput.close();
```

Best Practices

Code Organization

```
// Use header guards or #pragma once
#ifndef MYHEADER_H
#define MYHEADER_H
// Header content
#endif

// Or modern alternative:
#pragma once
```

Naming Conventions

```
// Variables and functions: camelCase or snake_case
int studentCount;
int student_count;

// Constants: ALL_CAPS
const int MAX_STUDENTS = 100;

// Classes: PascalCase
class StudentManager;
```

```cpp
// Private members: trailing underscore (optional)
class MyClass {
private:
    int value_;
};
```

Modern C++ Features (C++11 and later)

```cpp
// Use auto for type deduction
auto result = someComplexFunction();

// Use range-based for loops
for (const auto& item : container) {
    // Process item
}

// Use nullptr instead of NULL
int* ptr = nullptr;

// Use uniform initialization
int value{42};
vector<int> vec{1, 2, 3, 4, 5};

// Use smart pointers instead of raw pointers
auto ptr = make_unique<MyClass>();
```

Performance Tips

- Pass large objects by const reference: `void func(const LargeObject& obj)`
- Use prefix increment for iterators: `++it` instead of `it++`
- Reserve vector capacity when size is known: `vec.reserve(1000)`
- Use `emplace_back()` instead of `push_back()` for constructing in place
- Prefer algorithms over hand-written loops
- Use `const` wherever possible for compiler optimizations

Memory Safety

- Always initialize variables: `int count = 0;`
- Use smart pointers instead of raw pointers
- Follow RAII principles
- Check array bounds

- Match every `new` with `delete`, every `new[]` with `delete[]`

Useful Resources

- **Official Documentation:** cppreference.com
- **Learning Platform:** learncpp.com
- **Compiler Explorer:** godbolt.org - See assembly output
- **Best Practices:** Core Guidelines
- **STL Reference:** cplusplus.com

C++ Pseudocode Comments Reference

Table of Contents

Program Structure Comments

Basic Program Flow

```
// BEGIN program
// START main execution
// INITIALIZE program
// SET UP environment
// CONFIGURE settings
```

```
// PREPARE data structures
// END program
// TERMINATE execution
// CLEANUP resources
// EXIT with status code
```

Module Organization

```
// INCLUDE necessary headers
// IMPORT required libraries
// DECLARE global constants
// DEFINE global variables
// DECLARE function prototypes
// IMPLEMENT main logic
// DEFINE helper functions
```

Variable and Data Comments

Variable Operations

```
// DECLARE variable name
// INITIALIZE variable to value
// SET variable = value
// ASSIGN value to variable
// UPDATE variable with new value
// INCREMENT variable by amount
// DECREMENT variable by amount
// RESET variable to default
// CLEAR variable contents
// COPY source to destination
```

Data Type Operations

```
// CONVERT type1 to type2
// CAST variable as new_type
// VALIDATE data type
// CHECK type compatibility
// ENSURE proper type conversion
// PARSE string to number
// FORMAT number as string
// SERIALIZE data structure
// DESERIALIZE from format
```

Control Flow Comments

Basic Control Structures

```
// IF condition THEN
// ELSE IF condition THEN
// ELSE
// END IF

// BEGIN block
// END block
// EXECUTE statement
// PERFORM action
// CALL function
// RETURN value
// EXIT function
// BREAK from loop
// CONTINUE to next iteration
// GOTO label
```

Decision Making

```
// CHECK if condition is true
// VERIFY that condition holds
// ENSURE condition is met
// TEST condition
// EVALUATE expression
// COMPARE values
// DETERMINE best option
// SELECT appropriate case
// CHOOSE between alternatives
// DECIDE based on criteria
```

Function and Method Comments

Function Structure

```
// FUNCTION function_name
// PROCEDURE procedure_name
// METHOD method_name
// BEGIN function_name
// END function_name
// RETURN result
```

```
// RETURN nothing (void)
// EXIT function early
```

Parameter Handling

```
// ACCEPT parameter list
// RECEIVE input parameters
// VALIDATE input parameters
// CHECK parameter bounds
// PROCESS input arguments
// PASS parameters to function
// PASS by value
// PASS by reference
// RETURN multiple values
```

Function Calls

```
// CALL function with arguments
// INVOKE method on object
// EXECUTE function
// APPLY function to data
// TRIGGER callback function
// DISPATCH to handler
// DELEGATE to helper function
```

Loop Comments

Loop Types

```
// FOR each item IN collection
// FOR counter FROM start TO end
// FOR counter FROM start TO end STEP increment
// WHILE condition is true
// DO WHILE condition is true
// REPEAT UNTIL condition is false
// LOOP indefinitely
// ITERATE through collection
// TRAVERSE data structure
```

Loop Control

```
// BEGIN loop
// END loop
// CONTINUE with next iteration
// BREAK out of loop
// SKIP current iteration
// EXIT loop early
// RESTART loop
// PAUSE loop execution
```

Loop Patterns

```
// FOREACH element in array
// SCAN through list
// VISIT each node
// PROCESS each item
// EXAMINE all elements
// WALK through structure
// STEP through sequence
// CYCLE through options
```

Condition Comments

Conditional Expressions

```
// IF condition THEN action
// WHEN condition occurs
// PROVIDED that condition
// ASSUMING condition is true
// GIVEN that condition holds
// IN CASE condition happens
// UNLESS condition is false
// ONLY IF condition is met
```

Logical Operations

```
// condition1 AND condition2
// condition1 OR condition2
// NOT condition
// condition1 XOR condition2
// ALL conditions are true
// ANY condition is true
```

```
// NONE of the conditions
// EITHER condition1 OR condition2
```

Comparison Operations

```
// COMPARE value1 with value2
// CHECK if equal
// CHECK if not equal
// CHECK if greater than
// CHECK if less than
// CHECK if greater or equal
// CHECK if less or equal
// DETERMINE relationship
// ESTABLISH ordering
```

Array and Data Structure Comments

Array Operations

```
// DECLARE array of size
// INITIALIZE array with values
// ACCESS element at index
// SET element at index to value
// GET element at index
// APPEND element to array
// INSERT element at position
// DELETE element at position
// REMOVE element from array
// FIND element in array
// SEARCH for value
// SORT array elements
// REVERSE array order
// COPY array contents
// RESIZE array
```

Data Structure Operations

```
// CREATE new data structure
// DESTROY data structure
// ADD item to structure
// REMOVE item from structure
// UPDATE item in structure
// SEARCH structure for item
```

```
// TRAVERSE entire structure
// COUNT items in structure
// CHECK if structure is empty
// CLEAR all items
// MERGE two structures
// SPLIT structure
```

Stack Operations

```
// PUSH item onto stack
// POP item from stack
// PEEK at top item
// CHECK if stack is empty
// GET stack size
```

Queue Operations

```
// ENQUEUE item
// DEQUEUE item
// CHECK front of queue
// CHECK if queue is empty
// GET queue size
```

Tree Operations

```
// INSERT node into tree
// DELETE node from tree
// SEARCH tree for value
// TRAVERSE tree (preorder/inorder/postorder)
// FIND parent node
// FIND child nodes
// CALCULATE tree height
// BALANCE tree
```

Algorithm Pattern Comments

Searching Algorithms

```
// LINEAR search through array
// BINARY search in sorted array
// HASH table lookup
// FIND first occurrence
```

```
// FIND last occurrence
// FIND all occurrences
// LOCATE target element
// DISCOVER matching items
```

Sorting Algorithms

```
// BUBBLE sort algorithm
// SELECTION sort algorithm
// INSERTION sort algorithm
// QUICK sort algorithm
// MERGE sort algorithm
// HEAP sort algorithm
// ARRANGE in ascending order
// ARRANGE in descending order
// ORDER by criteria
// RANK elements
```

Recursive Patterns

```
// BASE case: condition
// RECURSIVE case: function calls itself
// DIVIDE problem into subproblems
// CONQUER subproblems
// COMBINE solutions
// REDUCE problem size
// SOLVE smaller instance
```

Input/Output Comments

Input Operations

```
// READ input from user
// GET value from keyboard
// ACCEPT user input
// PROMPT user for input
// REQUEST data entry
// CAPTURE user response
// RECEIVE input stream
// PARSE input format
// VALIDATE input data
// SANITIZE input
```

Output Operations

```
// PRINT message to screen
// DISPLAY result
// SHOW output
// OUTPUT formatted data
// WRITE to console
// PRESENT information
// RENDER display
// GENERATE report
// PRODUCE output
```

File Operations

```
// OPEN file for reading
// OPEN file for writing
// CLOSE file
// READ from file
// WRITE to file
// APPEND to file
// SEEK to position
// CREATE new file
// DELETE file
// COPY file
// MOVE file
// CHECK if file exists
```

Error Handling Comments

Error Detection

```
// CHECK for errors
// VALIDATE operation success
// DETECT error condition
// VERIFY operation completed
// ENSURE no errors occurred
// MONITOR for exceptions
// WATCH for failures
// GUARD against errors
```

Error Response

```
// HANDLE error condition
// CATCH exception
// RECOVER from error
// RETRY operation
// FALLBACK to alternative
// ABORT operation
// REPORT error
// LOG error message
// THROW exception
// RAISE error condition
```

Error Prevention

```
// PREVENT error condition
// AVOID potential problem
// PROTECT against failure
// SAFEGUARD operation
// ENSURE valid state
// MAINTAIN data integrity
// ESTABLISH preconditions
// VERIFY postconditions
```

Memory Management Comments

Memory Allocation

```
// ALLOCATE memory for object
// RESERVE memory space
// REQUEST memory block
// ASSIGN memory location
// CREATE dynamic object
// INSTANTIATE new object
```

Memory Deallocation

```
// DEALLOCATE memory
// FREE allocated memory
// RELEASE memory block
// DESTROY object
// CLEANUP resources
// RETURN memory to system
```

Memory Operations

```
// COPY memory block
// MOVE memory contents
// CLEAR memory area
// INITIALIZE memory
// ZERO memory block
// SET memory pattern
```

Object-Oriented Comments

Class Operations

```
// DEFINE class
// DECLARE class members
// IMPLEMENT class methods
// CREATE class instance
// INSTANTIATE object
// INITIALIZE object state
// DESTROY object
// CLEANUP object resources
```

Inheritance Comments

```
// INHERIT from base class
// EXTEND base functionality
// OVERRIDE base method
// CALL parent method
// IMPLEMENT interface
// SPECIALIZE behavior
```

Encapsulation Comments

```
// HIDE implementation details
// PROTECT internal state
// PROVIDE public interface
// EXPOSE necessary methods
// MAINTAIN data integrity
// CONTROL access to data
```

Mathematical Operation Comments

Basic Math

```
// ADD numbers
// SUBTRACT numbers
// MULTIPLY numbers
// DIVIDE numbers
// CALCULATE remainder
// COMPUTE power
// FIND square root
// CALCULATE absolute value
```

Advanced Math

```
// COMPUTE trigonometric function
// CALCULATE logarithm
// FIND factorial
// GENERATE random number
// ROUND to nearest integer
// TRUNCATE decimal
// FIND minimum value
// FIND maximum value
// CALCULATE average
// COMPUTE standard deviation
```

Search and Sort Comments

Search Patterns

```
// SEQUENTIAL search
// BINARY search
// DEPTH-first search
// BREADTH-first search
// PATTERN matching
// SUBSTRING search
// APPROXIMATE search
// FUZZY matching
```

Sort Patterns

```
// STABLE sort
// UNSTABLE sort
// IN-PLACE sort
```

```
// EXTERNAL sort
// COMPARISON-based sort
// NON-comparison sort
// ADAPTIVE sort
// ONLINE sort
```

File Processing Comments

File Reading

```
// OPEN file for input
// READ entire file
// READ line by line
// READ character by character
// READ binary data
// PARSE file format
// PROCESS file contents
// EXTRACT information
```

File Writing

```
// OPEN file for output
// WRITE data to file
// APPEND data to file
// WRITE formatted output
// SAVE binary data
// CREATE backup copy
// UPDATE file contents
// GENERATE output file
```

File Management

```
// LIST directory contents
// CHECK file permissions
// GET file size
// GET file timestamp
// CREATE directory
// REMOVE directory
// NAVIGATE file system
// RESOLVE file path
```

Usage Guidelines

Best Practices for Pseudocode Comments

1. **Be Descriptive but Concise**

```
// GOOD: Calculate monthly payment amount
// AVOID: Do some math stuff
```

2. **Use Action Verbs**

```
// GOOD: VALIDATE user credentials
// AVOID: User credentials check
```

3. **Indicate Data Flow**

```
// INPUT: username, password
// PROCESS: authenticate user
// OUTPUT: authentication status
```

4. **Show Algorithm Steps**

```
// STEP 1: Initialize variables
// STEP 2: Read input data
// STEP 3: Process calculations
// STEP 4: Display results
```

5. **Mark Important Sections**

```
// PRECONDITION: array must be sorted
// POSTCONDITION: target found or not found
// INVARIANT: loop counter always positive
```

Comment Formatting Conventions

```
// Single-line pseudocode comment

/* Multi-line pseudocode comment
   for complex algorithm descriptions
   or detailed explanations */

/**
```

```
 * Documentation-style comment
 * for function/class descriptions
 * @param input - description
 * @return description
 */

// TODO: Implement error handling
// FIXME: Handle edge case
// NOTE: This assumes sorted input
// WARNING: Potential performance issue
// HACK: Temporary workaround
```

Pseudocode Comment Templates

Algorithm Template

```
// ALGORITHM: [Algorithm Name]
// PURPOSE: [What it does]
// INPUT: [Input parameters]
// OUTPUT: [Return values]
// PRECONDITION: [Requirements]
// POSTCONDITION: [Guarantees]
//
// BEGIN [Algorithm Name]
//   [Step 1 description]
//   [Step 2 description]
//   ...
// END [Algorithm Name]
```

Function Template

```
// FUNCTION: [Function Name]
// PARAMETERS: [Parameter list with types]
// RETURNS: [Return type and description]
// DESCRIPTION: [What the function does]
// COMPLEXITY: [Time/Space complexity]
```

Loop Template

```
// LOOP: [Loop purpose]
// INITIALIZE: [Initial conditions]
// CONDITION: [Continue condition]
```

```
// UPDATE: [How variables change]
// BODY: [What happens each iteration]
```

This reference provides a comprehensive collection of pseudocode comments commonly used in C++ programming. Use these comments to document your algorithms, explain complex logic, and make your code more readable and maintainable.

Back Matter

Source

- based_on:: C++ Cheat Sheet & Quick Reference.pdf

References

- see::

Terms

-

Target

- used_in::

Tasks

-

Questions

- question::