



Object Oriented Programming for Verification

Anoushka Tripathi

Preface

In today's world of hardware design and verification, learning **Object-Oriented Programming (OOP)** is more important than ever. OOP is not just a programming concept; it is a way of thinking that helps us build more efficient, reusable, and maintainable systems. For those working in hardware verification, especially using the **Universal Verification Methodology (UVM)**, understanding OOP can make the job much easier and the results far more effective.

As a verification engineer, you are responsible for ensuring that a hardware design functions correctly, even under the most challenging scenarios. This is where **UVM**, a powerful verification framework, comes into play. UVM uses **SystemVerilog**, which heavily relies on OOP concepts like **classes**, **inheritance**, and **polymorphism**. If you understand OOP, you will be able to take full advantage of UVM's capabilities and write verification code that is both reusable and adaptable to different projects.

Why Learning OOP is Important for Verification

Object-Oriented Programming brings several benefits that are very useful in the world of verification:

1. **Abstraction:** OOP allows you to focus on the high-level goals of your verification without getting bogged down in unnecessary details. In UVM, this means you can create components like **drivers** and **monitors** that handle the details behind the scenes, letting you focus on the big picture.
2. **Encapsulation:** This concept helps keep different parts of your code separate so that changes in one part don't accidentally affect another. In UVM, this means you can create clean, self-contained verification components that work together smoothly.
3. **Inheritance:** One of the best features of OOP is that it lets you reuse and extend existing code. In UVM, you can create a **base test** that can be extended with new features without having to rewrite the entire test from scratch. This saves a lot of time and effort.
4. **Polymorphism:** This allows you to write flexible code that can work with different data types or components. In UVM, polymorphism is used to make sure your verification environment can adapt to different scenarios without needing major rewrites.

How OOP and UVM Work Together

UVM is designed to take full advantage of OOP. Everything in UVM is built around classes, and OOP concepts like **abstraction**, **encapsulation**, and **polymorphism** are used to create flexible, reusable testbenches. For example:

- **Sequences** in UVM rely on inheritance and polymorphism to create a structure where different test scenarios can be run easily.
- **Factory patterns** are used in UVM to dynamically create objects during simulation, giving you the flexibility to change your test environment without rewriting code.
- **Transaction-level modeling (TLM)** uses OOP concepts to simplify communication between different parts of the testbench, improving the performance and clarity of your verification code.

Purpose of These Notes

These notes are designed to help you learn **Object-Oriented Programming** in a way that is both simple and practical, especially in the context of **hardware verification** using **UVM**. By understanding the basics of OOP,

you will be better equipped to write **UVM-based testbenches** that are easier to maintain, reuse, and extend. Whether you are just getting started or already have some experience, this book will guide you step by step through the important concepts.

These notes embarks on the journey of Object-Oriented Programming (OOP) by introducing concepts through C++. C++ has been chosen as the starting point because it lays a strong foundation in OOP principles, which are essential for mastering complex verification methodologies like the Universal Verification Methodology (UVM). UVM and even SystemVerilog (SV) are deeply influenced by C++, both in their structure and implementation.

By building a thorough understanding of OOP in C++, readers will be better equipped to grasp the underlying concepts of UVM, making the transition to verification much smoother. This approach not only enhances comprehension but also establishes the vital connection between software development and hardware verification, both of which are essential in today's technology landscape.

CHAPTER 1 : Introduction to C++

Basic C++ Program Overview

Simple Program Example:

```
#include <iostream> // include header file
using namespace std;
int main () {
    cout << "C++ is better than C.\n"; // C++ statement
    return 0;
}
```

Key Features:

- **Main Function:** Every C++ program must contain a `main()` function where execution begins.
- **Free-Form Language:** C++ ignores spaces and carriage returns, but every statement must end with a semicolon.
- **Comments:**
 - Single-line comment: `//`
 - Multi-line comment: `/* ... */`

Output Operator:

- **cout:** Represents the standard output stream (usually the screen).
- **<<:** The insertion operator, which sends the output to `cout`.
- **Operator Overloading:** In C++, operators can perform different tasks depending on the context.

Header Files:

- **#include <iostream>:** Includes predefined functionalities, such as `cout` and `<<`.
- **Old vs. New Header Files:**
 - Old: `<iostream.h>`
 - New: `<iostream>`

Namespace:

- **using namespace std;:** This brings all identifiers from the standard library into the current scope.

Return Type of main():

- **int main():** Returns an integer to the operating system. Typically ends with `return 0;`.

Input/Output Operators Example:

```
#include <iostream>
using namespace std;
int main() {
    float number1, number2, sum, average;
    cout << "Enter two numbers:";
    cin >> number1 >> number2;
    sum = number1 + number2;
    average = sum / 2;
    cout << "Sum = " << sum << "\n";
    cout << "Average = " << average << "\n";
    return 0;
}
```

- **Input Operator:**
 - **cin:** Represents the standard input stream (usually the keyboard).
 - **>>:** The extraction operator, which reads input.
- **Cascading Operators:** Multiple input/output operations can be chained together:
 - Example: `cin >> number1 >> number2;`

Example with Classes:

```
#include <iostream>
using namespace std;
class Person {
    char name[30];
    int age;
public:
    void getData();
    void display();
};
void Person::getData() {
    cout << "Enter name: ";
    cin >> name;
    cout << "Enter age: ";
    cin >> age;
}
void Person::display() {
    cout << "Name: " << name << "\n";
    cout << "Age: " << age << "\n";
}
int main() {
    Person p;
    p.getData();
    p.display();
    return 0;
}
```

- **Class:** A user-defined data type that binds data and functions.
- **Member Functions:** Functions inside a class, such as `getData()` and `display()`.
- **Object:** An instance of a class (e.g., `p` is an object of class `Person`).

Structure of a C++ Program:

A typical C++ program has:

1. **Include Files:** To bring in external functions and definitions.
2. **Main Function:** The starting point.
3. **Class Definitions:** Define data types and associated operations.
4. **Member Function Definitions:** Implement class functionality.

Compiling and Linking:

- **Source Files:** C++ source files usually have extensions like `.cpp` or `.cxx`.
- **Compilation:** Converts source code into object files (`.o`), which are then linked into an executable.
 - Example command in UNIX: `CC -o example.o example.C`

LET US DO WARM UP:

PROGRAMMING EXERCISES

Write a program to display the following output using a single cout statement:

Maths = 90

Physics = 77

Chemistry = 69

```
main.cpp
1  #include <iostream>
2  using namespace std;
3  int main()
4  {
5      cout << "Maths = 90\nPhysics = 77\nChemistry = 69\n";
6      return 0;
7  }
8
```

Maths = 90
Physics = 77
Chemistry = 69

Write a program to read two numbers from the keyboard and display the larger value on the screen.

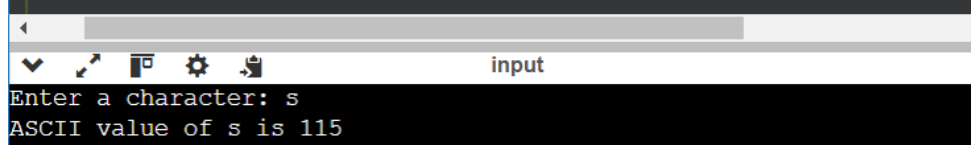
```
1  #include <iostream>
2  using namespace std;
3  int main()
4  {
5      int a, b;
6      cout << "Enter two numbers: ";
7      cin >> a >> b;
8      if (a > b)
9          cout << "Larger value is: " << a;
10     else
11         cout << "Larger value is: " << b;
12     return 0;
13 }
14
15
```

input

Enter two numbers: 1
2
Larger value is: 2

Write a program that inputs a character from the keyboard and displays its corresponding ASCII value.

```
#include <iostream>
using namespace std;
int main()
{
    char ch;
    cout << "Enter a character: ";
    cin >> ch;
    cout << "ASCII value of " << ch << " is " << int(ch) << endl;
    return 0;
}
```

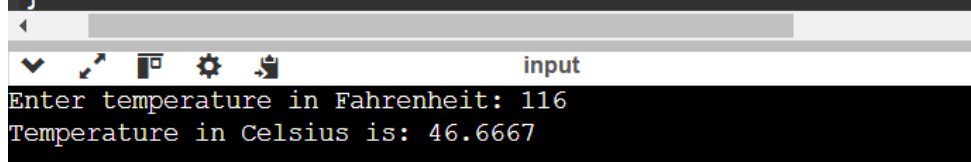


input

Enter a character: s
ASCII value of s is 115

Write a C++ program that will ask for a temperature in Fahrenheit and display it in Celsius.

```
#include <iostream>
using namespace std;
int main()
{
    float fahrenheit, celsius;
    cout << "Enter temperature in Fahrenheit: ";
    cin >> fahrenheit;
    celsius = (fahrenheit - 32) * 5 / 9;
    cout << "Temperature in Celsius is: " << celsius << endl;
    return 0;
}
```



input

Enter temperature in Fahrenheit: 116
Temperature in Celsius is: 46.6667

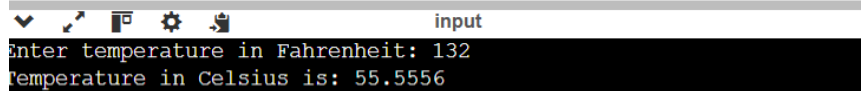
Redo above Ex. using a class called temp and member functions.

```
#include <iostream>
using namespace std;
class Temp
{
public:
    float fahrenheit;

    void getFahrenheit()
    {
        cout << "Enter temperature in Fahrenheit: ";
        cin >> fahrenheit;
    }

    void convertToCelsius()
    {
        float celsius = (fahrenheit - 32) * 5 / 9;
        cout << "Temperature in Celsius is: " << celsius << endl;
    }
};

int main()
{
    Temp t;
    t.getFahrenheit();
    t.convertToCelsius();
    return 0;
}
```



```
input
Enter temperature in Fahrenheit: 132
Temperature in Celsius is: 55.5556
```

CHAPTER 2 : Tokens, Expressions and Control Structures

Introduction to C++ Tokens and Control Structures

As mentioned earlier, C++ is an extended version of the C programming language. Most of the rules and features in C still apply to C++, but there are some differences and additional features that make C++ more powerful, especially for object-oriented programming. In this chapter, we'll talk about those differences and additions, focusing on the building blocks of C++ programs: **tokens** and control structures.

What are Tokens?

In programming, **tokens** are the smallest individual elements or building blocks of a program. Just like in English where words make up a sentence, tokens make up a C++ program. C++ tokens include:

- Keywords (like `int`, `if`, `return`)
- Identifiers (names of variables, functions, etc.)
- Constants (fixed values like `5`, `3.14`, `'A'`)
- Strings (like `"Hello"`)
- Operators (like `+`, `-`, `=`)

In C++, you write programs using these tokens along with spaces and the rules (syntax) of the language. Many of these tokens are the same as in C, but C++ has some extras and minor changes.

Keywords

| | | | |
|---------------------------|------------------------|-------------------------------|-----------------------|
| <code>asm</code> | <code>double</code> | <code>new</code> | <code>switch</code> |
| <code>auto</code> | <code>else</code> | <code>operator</code> | <code>template</code> |
| <code>break</code> | <code>enum</code> | <code>private</code> | <code>this</code> |
| <code>case</code> | <code>extern</code> | <code>protected</code> | <code>throw</code> |
| <code>catch</code> | <code>float</code> | <code>public</code> | <code>try</code> |
| <code>char</code> | <code>for</code> | <code>register</code> | <code>typedef</code> |
| <code>class</code> | <code>friend</code> | <code>return</code> | <code>union</code> |
| <code>const</code> | <code>goto</code> | <code>short</code> | <code>unsigned</code> |
| <code>continue</code> | <code>if</code> | <code>signed</code> | <code>virtual</code> |
| <code>default</code> | <code>inline</code> | <code>sizeof</code> | <code>void</code> |
| <code>delete</code> | <code>int</code> | <code>static</code> | <code>volatile</code> |
| <code>do</code> | <code>long</code> | <code>struct</code> | <code>while</code> |
| <i>Added by ANSI C++</i> | | | |
| <code>bool</code> | <code>export</code> | <code>reinterpret_cast</code> | <code>typename</code> |
| <code>const_cast</code> | <code>false</code> | <code>static_cast</code> | <code>using</code> |
| <code>dynamic_cast</code> | <code>mutable</code> | <code>true</code> | <code>wchar_t</code> |
| <code>explicit</code> | <code>namespace</code> | <code>typeid</code> | |

Keywords are special words that have a specific meaning in the C++ language. You can't use them as names for variables or anything else in your program. Some keywords, like `int`, `if`, and `for`, are the same in both C and C++. But C++ also introduces new keywords to support object-oriented features, like `class`, `virtual`, and `public`.

Check out Table above for a complete list of C++ keywords. Those marked in **bold** are the ones that are shared with C, while the others are unique to C++.

Identifiers and Constants

Identifiers are the names you give to things like variables, functions, arrays, or classes in your program. These are the rules for naming identifiers (same in both C and C++):

1. You can only use letters (uppercase or lowercase), digits, and underscores (`_`).
2. The name cannot start with a digit.
3. Uppercase and lowercase letters are treated as different.
4. You can't use a keyword as a name.

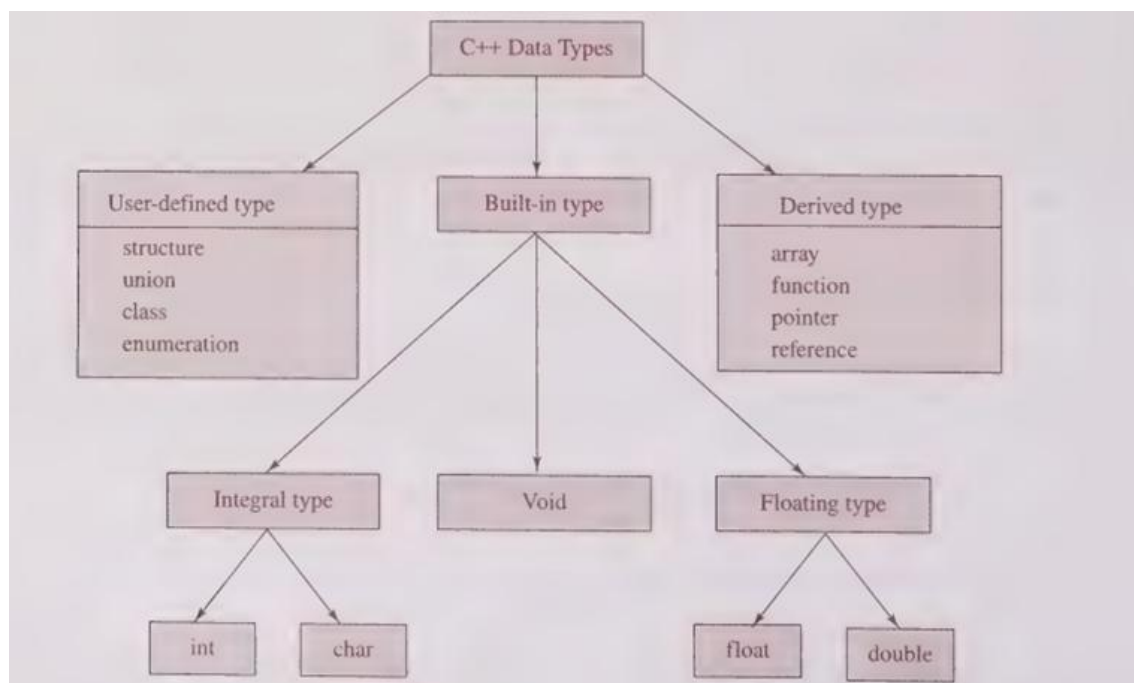
A key difference between C and C++ is how long an identifier name can be. In C, only the first 32 characters matter, but in C++, there's no limit on name length—every character counts.

Constants are values that don't change while the program runs. Like in C, C++ supports many types of constants, such as integers (`5`), characters (`'A'`), floating-point numbers (`3.14`), and strings (`"Hello"`).

Basic Data Types

C++ includes several **data types** that specify the kind of data a variable can hold. These can be divided into categories, like built-in types (e.g., integers and floats), user-defined types (e.g., classes and structures), and derived types (e.g., arrays and pointers).

The basic types, such as `int`, `float`, and `char`, are mostly the same in C and C++. However, C++ allows you to use **modifiers** (like `signed`, `unsigned`, `long`, and `short`) to alter the range or behavior of these types. For example, a `short int` might save memory by using less space, while an `unsigned int` can only store non-negative values.



| Type | Bytes | Range |
|--------------------|-------|---------------------------|
| char | 1 | -128 to 127 |
| unsigned char | 1 | 0 to 255 |
| signed char | 1 | -128 to 127 |
| int | 4 | -2147483648 to 2147483647 |
| unsigned int | 4 | 0 to 4294967295 |
| signed int | 4 | -2147483648 to 2147483647 |
| short int | 2 | -32768 to 32767 |
| unsigned short int | 2 | 0 to 65535 |
| signed short int | 2 | -32768 to 32767 |
| long int | 4 | -2147483648 to 2147483647 |
| signed long int | 4 | -2147483648 to 2147483647 |
| unsigned long int | 4 | 0 to 4294967295 |
| float | 4 | 3.4E-38 to 3.4E+38 |
| double | 8 | 1.7E-308 to 1.7E+308 |
| long double | 10 | 3.4E-4932 to 1.1E+4932 |

The `void` Type in C++ and Its Uses

The `void` type was introduced in ANSI C and continues to be an important part of C++. It serves two main purposes:

1. **Indicating that a function doesn't return any value:** When you want a function to perform a task but not return any data, you use `void` as its return type. For example:

```
void myFunction(void);
```

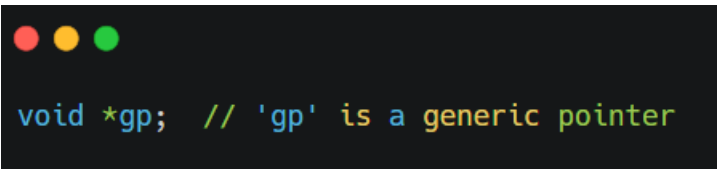
This function, `myFunction`, doesn't return anything. It simply performs its task and ends.

2. **Indicating an empty argument list:** In C++, you can specify that a function takes no arguments by using `void` in its parameter list. The same example applies here:

```
void myFunction(void); // This function has no arguments
```

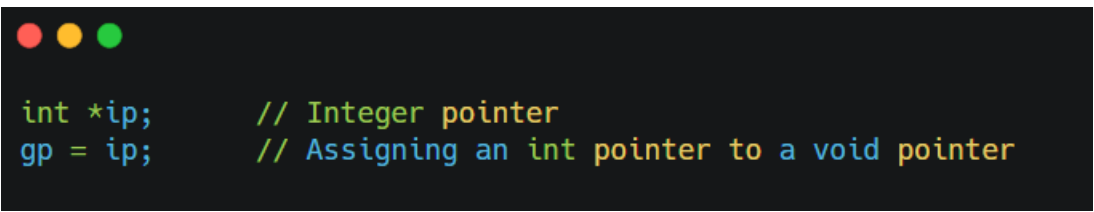
`void` Pointers – Generic Pointers

One of the more interesting uses of `void` in C++ is with **generic pointers**. A generic pointer is a pointer that doesn't have a specific type attached to it. This allows you to use it to point to data of any type. For example:



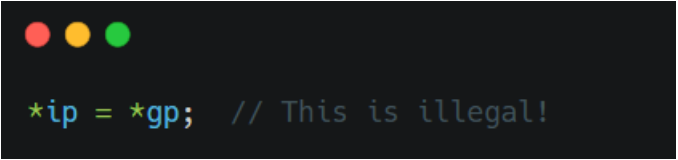
```
void *gp; // 'gp' is a generic pointer
```

You can assign any type of pointer (like an integer or float pointer) to a `void` pointer:



```
int *ip;        // Integer pointer
gp = ip;        // Assigning an int pointer to a void pointer
```

This is perfectly valid. However, there's a catch: you **cannot directly use** (or dereference) a `void` pointer. For instance, trying to do this:



```
*ip = *gp; // This is illegal!
```

won't work. It doesn't make sense to dereference a pointer when the type of data it's pointing to is not defined.

Casting with `void` Pointers

In both C and C++, you can assign any specific pointer type (like an `int*` or `float*`) to a `void*` without needing a cast. But when you want to go the other way (assign a `void*` to a specific pointer type), things are different.

- In ANSI C, you can directly assign a `void*` pointer to any other pointer type without a cast. For example:

```
int *ptr1;
char *ptr2;
ptr2 = ptr1; // This works in ANSI C
```

- In C++, however, you can't do this. You must explicitly cast the `void*` pointer to the correct type. For example:

```
int *ptr1;
char *ptr2;
ptr2 = (char*)ptr1; // Casting is required
in C++
```

Without this cast, C++ will throw an error because it needs to know the specific type before making such assignments.

User-Defined Data Types: Structures, Unions, and Enumerations

In programming, **user-defined data types** allow you to create custom types to better represent real-world objects and problems. Here's how they work:

Structures

- A **structure** groups together data of different types, making it easier to represent a complex object.
- Example: A book has different attributes like title (text), author (text), number of pages (integer), and price (decimal). Instead of handling these separately, you can group them into a structure.

Syntax:

```
struct book {  
    char title[25];  
    char author[25];  
    int pages;  
    float price;  
};  
  
struct book book1, book2; // Declares two  
book objects
```

- Access members using the **dot operator**:

```
book1.pages = 550;  
book2.price = 225.75;
```

Unions

- A **union** also groups data of different types, but it saves memory by using the same memory space for all its members.
- This means only one member can be accessed at a time.

Example:

- In this example:

```
union result {  
    int marks;  
    char grade;  
    float percent;  
};
```

- A union allocates memory equal to its largest member (4 bytes for `float` in a 32-bit system).
- If this was a structure, it would occupy 12 bytes (sum of all member sizes).

| Feature | Structure | Union |
|-------------------|---------------------------------------|--|
| Keyword | <code>struct</code> | <code>union</code> |
| Memory Usage | Separate memory for each member | Shared memory for all members |
| Access | All members accessible simultaneously | Only one member accessible at a time |
| Memory Efficiency | Less efficient | More efficient when simultaneous access is unnecessary |

Enumerated Data Types (Enums)

- **Enums** assign names to a list of integer values, improving code readability.
- Example:

```
enum colour { red, blue, green };
enum shape { circle, square, triangle };
```

- By default:
 - `red = 0, blue = 1, green = 2.`
 - You can also assign custom values:

```
enum colour { red = 5, blue, green = 10 }; // blue = 6,
green = 10
```

- Enums are like constants, and their values can be used directly:

```
int c = red; // Assigns 5 to c
```


- **Anonymous enums** allow defining constants without a name:

```
enum { off, on }; // off = 0, on = 1
```

Storage Classes in C++

Storage classes in C

| Storage Specifier | Storage | Initial value | Scope | Life |
|-------------------|--------------|---------------|--------------------------|---------------------|
| auto | stack | Garbage | Within block | End of block |
| extern | Data segment | Zero | global Multiple files | Till end of program |
| static | Data segment | Zero | Within block | Till end of program |
| register | CPU Register | Garbage | Within block | End of block |

Storage Classes in C++ with Examples

Storage classes in C++ describe the **lifetime, visibility, default value, and storage location** of variables or functions. They dictate how a variable behaves during program execution.

Here's an overview of the six storage classes available in C++:

1. `auto` Storage Class

The `auto` keyword indicates that a variable has automatic storage duration. It is the **default** storage class for variables declared inside a block.

- **Scope:** Local to the block where it's declared.
- **Default Value:** Garbage value.
- **Memory Location:** RAM.
- **Lifetime:** Until the end of the block.

Example:

Note: After C++11, the `auto` keyword was redefined for type deduction and is no longer used for storage class specification.

```
#include <iostream>
using namespace std;

int main() {
    int a = 32; // auto variable
    float b = 3.2;
    string c = "vlsitechwithanoushka";

    cout << a << "\n" << b << "\n" << c <<
    "\n";
    return 0;
}
```

2. extern Storage Class

The `extern` keyword is used for variables declared in one file but defined elsewhere. It ensures external linkage, making the variable accessible across multiple files.

- **Scope:** Global.
- **Default Value:** Zero.
- **Memory Location:** RAM.
- **Lifetime:** Till the end of the program.

Example:

```
File: def.cpp
int var = 10; // Global variable
File: main.cpp
#include <iostream>
using namespace std;

extern int var; // Declaration of extern
variable

int main() {
    cout << var; // Accessing extern
variable
    return 0;
}
```

3. `static` Storage Class

The `static` keyword ensures that a variable retains its value between function calls. A `static` variable is initialized only once and exists throughout the program's lifetime.

- **Scope:** Local (but retains value across function calls).
- **Default Value:** Zero.
- **Memory Location:** RAM.
- **Lifetime:** Till the program ends.

Example:



```
#include <iostream>
using namespace std;

void counter() {
    static int count = 0; // Static variable
    count++;
    cout << count << "\n";
}

int main() {
    counter(); // Outputs: 1
    counter(); // Outputs: 2
    return 0;
}
```

4. `register` Storage Class

The `register` keyword is used to suggest that a variable be stored in a CPU register (if available) for faster access.

- **Scope:** Local.
- **Default Value:** Garbage value.
- **Memory Location:** CPU register or RAM (if no registers are available).
- **Lifetime:** Until the end of the block.

Example:

```

#include <iostream>
using namespace std;

int main() {
    register char letter = 'G'; // Register
    variable
    cout << letter;
    return 0;
}

```

Note: The `register` keyword is deprecated in C++17 and later.

5. mutable Storage Class

The `mutable` keyword allows modification of a data member in a `const` object or function.

- **Scope:** Defined by the containing object.
- **Lifetime:** Same as the object.

Example:

```

#include <iostream>
using namespace std;

class MyClass {
public:
    int x;
    mutable int y;

    MyClass() : x(10), y(20) {}
};

int main() {
    const MyClass obj; // Constant object
    obj.y = 50;        // Modifying mutable
    member
    cout << obj.y;
    return 0;
}

```

6. `thread_local` Storage Class

The `thread_local` keyword declares variables that are local to a thread. Each thread gets its own copy of the variable.

- **Scope:** Thread-specific.
- **Default Value:** Zero for built-in types.
- **Memory Location:** RAM.
- **Lifetime:** Till the thread ends.

Example:

```
#include <iostream>
#include <thread>
using namespace std;

thread_local int val = 10; // Thread-local variable

void threadFunc(int id) {
    val += id;
    cout << "Thread " << id << " value: " <<
val << '\n';
}

int main() {
    thread t1(threadFunc, 1);
    thread t2(threadFunc, 2);
    t1.join();
    t2.join();

    cout << "Main thread value: " << val <<
'\n';
    return 0;
}
```

Summary Table

| Storage Class | Scope | Lifetime | Default Value | Memory Location |
|---------------|-----------------|-------------------------|------------------------|--------------------|
| auto | Local | Block execution | Garbage value | RAM |
| extern | Global | Entire program | Zero | RAM |
| static | Local | Entire program | Zero | RAM |
| register | Local | Block execution | Garbage value | CPU Register / RAM |
| mutable | Object-specific | Same as object lifetime | Depends on declaration | Same as object |
| thread_local | Thread-specific | Thread execution | Depends on type | RAM |

Key Takeaway

Storage classes in C++ provide fine-grained control over variable behavior, enabling efficient memory management, optimized performance, and modular programming.