2024

C++

SIMPLIFIED JARGON

KIRAN VVN

Doc ID	Author	Version	Language	Page No
C++/Jargon	Kvvn	2.0	en	1

Contents

STL

1. Containers

Containers are objects that store collections of data. The major categories of containers are:

- **Sequence Containers**
- **Associative Containers**
- **Unordered Containers**
- **Container Adapters**

Sequence Containers

These store data in a linear sequence.

a. std::vector

Dynamic array that can resize itself.

```
#include <iostream>
#include <vector>
int main() {
  std::vector<int> vec = {1, 2, 3};
  vec.push_back(4);
  vec.pop_back();

  for (int val : vec) {
    std::cout << val << " ";
  }
  return 0;
}
```

b. std::deque

Double-ended queue allowing insertion and deletion at both ends.

```
#include <iostream>
#include <deque>

int main() {
   std::deque<int> dq = {1, 2, 3};
   dq.push_front(0);
   dq.push_back(4);

   for (int val : dq) {
      std::cout << val << " ";
   }
   return 0;
}
```

c. std::list

Doubly linked list for efficient insertion/deletion.

```
#include <iostream>
#include <list>
```

```
int main() {
   std::list<int> lst = {1, 2, 3};
   lst.push_back(4);
   lst.push_front(0);

   for (int val : lst) {
      std::cout << val << " ";
   }
   return 0;
}
```

Associative Containers

These store data in sorted order by key.

a. std::set

Stores unique elements in sorted order.

```
#include <iostream>
#include <set>

int main() {
   std::set<int> st = {3, 1, 4, 1};
   st.insert(2);

   for (int val : st) {
```

```
    std::cout << val << " ";
  }
  return 0;
}
```

b. std::map

Stores key-value pairs in sorted order by keys.

```
#include <iostream>
#include <map>

int main() {
  std::map<int, std::string> mp;
  mp[1] = "One";
  mp[2] = "Two";

  for (auto &pair : mp) {
    std::cout << pair.first << ": " << pair.second << "\n";
  }
  return 0;
}
```

c. std::multiset

Similar to set but allows duplicate elements.

```
#include <iostream>
#include <set>
```

```
int main() {
   std::multiset<int> ms = {1, 1, 2, 3};
   ms.insert(3);

   for (int val : ms) {
      std::cout << val << " ";
   }
   return 0;
}
```

d. std::multimap

Similar to map but allows duplicate keys.

```
#include <iostream>
#include <map>

int main() {
   std::multimap<int, std::string> mm;
   mm.insert({1, "One"});
   mm.insert({1, "Uno"});

   for (auto &pair : mm) {
      std::cout << pair.first << ": " << pair.second << "\n";
   }
   return 0;
}
```

Unordered Containers

These store elements in an unordered manner for faster access.

a. std::unordered_set

Stores unique elements in no particular order.

```
#include <iostream>

#include <unordered_set>


int main() {

  std::unordered_set<int> us = {3, 1, 4, 1};

  us.insert(2);


  for (int val : us) {

    std::cout << val << " ";

  }

  return 0;

}
```

b. std::unordered_map

Stores key-value pairs in no particular order.

```
#include <iostream>

#include <unordered_map>


int main() {

  std::unordered_map<int, std::string> um;

  um[1] = "One";

  um[2] = "Two";
```

```
  for (auto &pair : um) {
    std::cout << pair.first << ": " << pair.second << "\n";
  }
  return 0;
}
```

Container Adapters

These are wrappers around containers to provide specific functionalities.

a. std::stack

LIFO (Last In, First Out) structure.

```
#include <iostream>
#include <stack>

int main() {
  std::stack<int> stk;
  stk.push(1);
  stk.push(2);
  stk.push(3);

  while (!stk.empty()) {
    std::cout << stk.top() << " ";
    stk.pop();
```

```
    }
    return 0;
}
```

b. std::queue

FIFO (First In, First Out) structure.

```
#include <iostream>
#include <queue>

int main() {
    std::queue<int> q;
    q.push(1);
    q.push(2);
    q.push(3);

    while (!q.empty()) {
        std::cout << q.front() << " ";
        q.pop();
    }
    return 0;
}
```

c. std::priority_queue

Heap-based structure for priority ordering.

```
#include <iostream>
#include <queue>
```

```
int main() {
   std::priority_queue<int> pq;
   pq.push(1);
   pq.push(3);
   pq.push(2);

   while (!pq.empty()) {
      std::cout << pq.top() << " ";
      pq.pop();
   }
   return 0;
}
```

---

2. Algorithms

STL provides algorithms for searching, sorting, and manipulating data.

Sorting

```
#include <iostream>
#include <vector>
#include <algorithm>

int main() {
   std::vector<int> vec = {3, 1, 4, 2};
   std::sort(vec.begin(), vec.end());
```

```
    for (int val : vec) {
        std::cout << val << " ";
    }
    return 0;
}
```

Finding

```
#include <iostream>
#include <vector>
#include <algorithm>

int main() {
    std::vector<int> vec = {3, 1, 4, 2};
    auto it = std::find(vec.begin(), vec.end(), 4);

    if (it != vec.end()) {
        std::cout << "Found: " << *it << "\n";
    } else {
        std::cout << "Not Found\n";
    }
    return 0;
}
```

---

3. Iterators

Iterators provide a way to access elements in containers.

Basic Usage

```
#include <iostream>
#include <vector>

int main() {
   std::vector<int> vec = {1, 2, 3};
   for (std::vector<int>::iterator it = vec.begin(); it != vec.end(); ++it) {
      std::cout << *it << " ";
   }
   return 0;
}
```

---

4. Utilities

STL provides utility classes like pair and tuple.

std::pair

```
#include <iostream>
#include <utility>

int main() {
   std::pair<int, std::string> p = {1, "One"};
   std::cout << p.first << ": " << p.second << "\n";
   return 0;
```

}

std::tuple

```
#include <iostream>
#include <tuple>

int main() {
   std::tuple<int, std::string, char> t = {1, "One", 'A'};
   std::cout << std::get<0>(t) << ": " << std::get<1>(t) << " " << std::get<2>(t) << "\n";
   return 0;
}
```

Adaptors and Allocators in C++ STL

This section introduces **Adaptors** and **Allocators** in the C++ Standard Template Library (STL) with detailed explanations and examples.

Adaptors

1. Understanding Container Adaptors

Container Adaptors are specialized wrappers around existing containers in the STL that provide restricted interfaces. They modify the way a container is used without altering the underlying container's behavior. Common container adaptors include:

- **Stack**
- **Queue**
- **Priority Queue**

2. Adaptors in STL

a. Stack

A **stack** is a Last In First Out (LIFO) structure. It provides operations such as push, pop, and top.

```
#include <iostream>
#include <stack>
int main() {
  std::stack<int> stk;
  stk.push(10);
  stk.push(20);
  stk.push(30);

  while (!stk.empty()) {
    std::cout << "Top: " << stk.top() << std::endl; // Access top element
    stk.pop();                    // Remove top element
  }
  return 0;
}
```

Key Methods:

- push(): Adds an element.
- pop(): Removes the top element.
- top(): Accesses the top element.

---

b. Queue

A **queue** is a First In First Out (FIFO) structure. Elements are added at the back and removed from the front.

```
#include <iostream>
#include <queue>
int main() {
  std::queue<int> q;
  q.push(1);
  q.push(2);
  q.push(3);

  while (!q.empty()) {
    std::cout << "Front: " << q.front() << ", Back: " << q.back() << std::endl;
    q.pop(); // Remove the front element
  }
  return 0;
}
```

Key Methods:

- push(): Adds an element.
- pop(): Removes the front element.
- front(): Accesses the first element.
- back(): Accesses the last element.

---

c. Priority Queue

A **priority queue** is a special type of queue where elements are ordered by their priority. By default, it works as a max-heap.

```
#include <iostream>
#include <queue>
int main() {
   std::priority_queue<int> pq;
   pq.push(10);
   pq.push(20);
   pq.push(5);

   while (!pq.empty()) {
      std::cout << "Top: " << pq.top() << std::endl; // Access highest priority element
      pq.pop();                          // Remove top element
   }
   return 0;
}
```

Custom Comparator for Min-Heap:

```
#include <iostream>
#include <queue>
int main() {
   std::priority_queue<int, std::vector<int>, std::greater<int>> min_heap;
   min_heap.push(10);
```

```
    min_heap.push(20);
    min_heap.push(5);

    while (!min_heap.empty()) {
        std::cout << "Top: " << min_heap.top() << std::endl;
        min_heap.pop();
    }
    return 0;
}
```

---

3. Custom Adaptor Implementation

You can create your own adaptor by encapsulating a container and providing specific operations.

```
#include <iostream>
#include <deque>
template <typename T>
class CustomStack {
    std::deque<T> container;

public:
    void push(const T& value) { container.push_back(value); }
    void pop() { container.pop_back(); }
    T& top() { return container.back(); }
    bool empty() const { return container.empty(); }
```

```
};


int main() {
    CustomStack<int> stk;
    stk.push(10);
    stk.push(20);
    std::cout << "Top: " << stk.top() << std::endl; // Output: 20
    stk.pop();
    std::cout << "Top: " << stk.top() << std::endl; // Output: 10
    return 0;
}
```

Allocators

1. The Role of Allocators in STL

Allocators abstract memory management in STL. They manage memory allocation, deallocation, and object construction/destruction for STL containers.

Key Methods:

- allocate(): Allocates memory.
- deallocate(): Deallocates memory.
- construct(): Constructs an object.
- destroy(): Destroys an object.

2. Custom Allocator Implementation

Here's how you can define a custom allocator:

```
#include <iostream>
```

```
#include <memory>
#include <vector>

template <typename T>
class CustomAllocator {
public:
  using value_type = T;

  CustomAllocator() = default;

  T* allocate(std::size_t n) {
    std::cout << "Allocating " << n << " elements.\n";
    return static_cast<T*>(std::malloc(n * sizeof(T)));
  }

  void deallocate(T* p, std::size_t n) {
    std::cout << "Deallocating " << n << " elements.\n";
    std::free(p);
  }

  template <typename U, typename... Args>
  void construct(U* p, Args&&... args) {
    new (p) U(std::forward<Args>(args)...);
  }
```

```
  template <typename U>
  void destroy(U* p) {
    p->~U();
  }
};

int main() {
  std::vector<int, CustomAllocator<int>> vec;
  vec.push_back(10);
  vec.push_back(20);

  for (int val : vec) {
    std::cout << val << " ";
  }
  return 0;
}
```

---

3. Memory Management in STL Containers

STL containers rely on allocators for memory management. Custom allocators can be used to optimize memory usage, debug allocations, or implement specialized memory models.

Example: Using a custom allocator with std::map.

```
#include <iostream>
#include <map>
int main() {
```

```
    std::map<int, int, std::less<>, CustomAllocator<std::pair<const int, int>>> custom_map;

    custom_map[1] = 100;

    custom_map[2] = 200;


    for (const auto& [key, value] : custom_map) {

        std::cout << key << " -> " << value << "\n";

    }

    return 0;

}
```

Compilation Steps

1. **Compile using g++**:
   Use the following command to compile:

```
g++ -std=c++17 adaptor_allocator_example. -o adaptor_allocator
```

2. **Run the executable**:

```
./adaptor_allocator
```

3. **Debugging Memory**:
   Use tools like Valgrind to check memory usage:

```
valgrind ./adaptor_allocator
```

- **Adaptors** modify the behavior of existing containers for specific use cases like LIFO (stack) or FIFO (queue).
- **Allocators** handle memory management for STL containers and allow custom implementations for efficiency or debugging.
- C++ STL's flexibility with these components enables efficient and customizable data structures.

Interfaces

In C++, interfaces are implemented using **abstract classes**, which are classes with at least one pure virtual function. An abstract class provides a blueprint for derived classes to implement specific functionality. Abstract classes cannot be instantiated.

An **interface** defines a set of methods that derived classes must implement. It focuses on the *behavior* that a class must provide rather than how it implements it. This is useful when multiple derived classes need to follow the same contract but implement it differently.

Concept: Interface with a Car Example

Let's consider a car manufacturing scenario. Each car type (e.g., ElectricCar, GasCar) may have different ways of starting, accelerating, or stopping. However, all cars must implement these behaviors.

Defining the Interface

We define a car interface as an abstract class with pure virtual functions:

```
#include <iostream>

#include <string>


// Abstract class (Interface)

class ICar {

public:

  // Pure virtual functions

  virtual void start() = 0;

  virtual void accelerate() = 0;

  virtual void stop() = 0;


  // Virtual destructor
```

```
  virtual ~ICar() {}
};
```

In the above code:

1. The ICar class defines the interface with pure virtual functions start, accelerate, and stop.
2. A virtual destructor is added to ensure proper cleanup of derived objects.

Implementing the Interface

ElectricCar Class

```
class ElectricCar : public ICar {
public:
  void start() override {
    std::cout << "ElectricCar: Starting silently with the battery.\n";
  }

  void accelerate() override {
    std::cout << "ElectricCar: Accelerating with instant torque.\n";
  }

  void stop() override {
    std::cout << "ElectricCar: Regenerative braking activated.\n";
  }
};
```

GasCar Class

```
class GasCar : public ICar {
public:
  void start() override {
    std::cout << "GasCar: Ignition system activated.\n";
  }

  void accelerate() override {
    std::cout << "GasCar: Accelerating with engine power.\n";
  }

  void stop() override {
    std::cout << "GasCar: Hydraulic braking applied.\n";
  }
};
```

Using the Interface

We can use pointers to the ICar interface to call methods on different car types polymorphically.

```
int main() {
  ICar* car1 = new ElectricCar();
  ICar* car2 = new GasCar();
```

```
    std::cout << "Electric Car Actions:\n";
    car1->start();
    car1->accelerate();
    car1->stop();

    std::cout << "\nGas Car Actions:\n";
    car2->start();
    car2->accelerate();
    car2->stop();

    // Clean up
    delete car1;
    delete car2;

    return 0;
}
```

Output

```
Electric Car Actions:
ElectricCar: Starting silently with the battery.
ElectricCar: Accelerating with instant torque.
ElectricCar: Regenerative braking activated.
```

Gas Car Actions:

GasCar: Ignition system activated.

GasCar: Accelerating with engine power.

GasCar: Hydraulic braking applied.

Key Points

1. **Pure Virtual Functions:** Declared with = 0 and must be overridden in derived classes.
2. **Polymorphism:** Enables calling methods on different car types through a common interface (ICar).
3. **Virtual Destructor:** Ensures proper destruction of derived class objects when deleted via a base class pointer.

Advanced Usage: Factory Design with Interface

An interface can also be used in combination with the **Factory Design Pattern** to dynamically create objects of specific car types at runtime.

Constructors

Copy Constructor

A **copy constructor** is a special constructor in C++ used to create a new object as a copy of an existing object. It initializes an object using another object of the same class.

Syntax:

ClassName(const ClassName &obj);

Example:

```
#include <iostream>

using namespace std;


class Car {
   string brand;
public:
   Car(string b) : brand(b) {}


   // Copy Constructor
   Car(const Car &c) {
      brand = c.brand;
      cout << "Copy Constructor Called" << endl;
   }


   void display() {
      cout << "Brand: " << brand << endl;
   }
};
```

```
int main() {
  Car car1("Tesla");
  Car car2 = car1; // Copy constructor invoked
  car2.display();
  return 0;
}
```

Shallow Copy

A **shallow copy** copies all member variables as-is, including pointers. It does not duplicate the actual memory pointed to by the pointers, leading to shared references.

Example:

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

class Car {
  int *mileage;
public:
  Car(int m) {
    mileage = new int(m);
  }

  void display() {
    cout << "Mileage: " << *mileage << endl;
  }

  ~Car() {
    delete mileage;
```

```
    }
};

int main() {
    Car car1(15000);
    Car car2 = car1; // Shallow copy (default)
    car1.display();
    car2.display();
    return 0;
}
```

This leads to issues as both objects share the same memory.

Deep Copy

A **deep copy** duplicates all members, including dynamically allocated memory, ensuring that objects manage their own copies of resources.

Example:

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

class Car {
    int *mileage;
public:
    Car(int m) {
        mileage = new int(m);
    }

    // Deep Copy Constructor
```

```
  Car(const Car &c) {
    mileage = new int(*c.mileage);
  }

  void display() {
    cout << "Mileage: " << *mileage << endl;
  }

  ~Car() {
    delete mileage;
  }
};

int main() {
  Car car1(15000);
  Car car2 = car1; // Deep copy
  car1.display();
  car2.display();
  return 0;
}
```

Operator Overloading

Operator overloading allows defining custom behavior for operators when used with objects of a class.

Overloading +:

```
#include <iostream>
```

```
using namespace std;


class Point {
  int x, y;
public:
  Point(int a, int b) : x(a), y(b) {}

  // Overload + operator
  Point operator+(const Point &p) {
    return Point(x + p.x, y + p.y);
  }

  void display() {
    cout << "(" << x << ", " << y << ")" << endl;
  }
};

int main() {
  Point p1(2, 3), p2(4, 5);
  Point p3 = p1 + p2;
  p3.display();
  return 0;
}
```

Overloading =:

```
class MyClass {
```

```
  int value;
public:
  MyClass(int v = 0) : value(v) {}

  // Overload assignment operator
  MyClass& operator=(const MyClass &obj) {
    if (this != &obj) {
      value = obj.value;
    }
    return *this;
  }
};
```

Exception Handling

Basics of Exception Handling

Exception handling in C++ manages runtime errors using try, catch, and throw keywords.

Example:

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

int main() {
  try {
    throw "An error occurred";
  } catch (const char* msg) {
    cout << "Caught exception: " << msg << endl;
```

```
    }
    return 0;
}
```

Namespaces

Understanding Namespaces

Namespaces avoid naming conflicts by grouping related classes, functions, or variables.

Example:

```
#include <iostream>

namespace A {
    void display() {
        std::cout << "Namespace A" << std::endl;
    }
}

namespace B {
    void display() {
        std::cout << "Namespace B" << std::endl;
    }
}

int main() {
    A::display();
    B::display();
```

```
    return 0;
}
```

Smart Pointers

Smart pointers manage dynamic memory automatically.

Types of Smart Pointers

1. **unique_ptr** - Ownership of an object cannot be shared.
2. **shared_ptr** - Shared ownership.
3. **weak_ptr** - Observes shared ownership without increasing reference count.

Example:

```
#include <iostream>
#include <memory>

int main() {
    std::unique_ptr<int> p1 = std::make_unique<int>(42);
    std::cout << "Value: " << *p1 << std::endl;

    std::shared_ptr<int> p2 = std::make_shared<int>(10);
    std::cout << "Value: " << *p2 << std::endl;

    return 0;
}
```

Templates

Templates provide a way to create generic classes and functions.

Function Templates

```
#include <iostream>


// Function Template
template <typename T>
T add(T a, T b) {
  return a + b;
}


int main() {
  std::cout << add(2, 3) << std::endl;
  std::cout << add(2.5, 3.5) << std::endl;
  return 0;
}
```

Class Templates

```
#include <iostream>


// Class Template
template <typename T>
class Box {
  T value;
public:
  Box(T v) : value(v) {}
  void display() {
    std::cout << "Value: " << value << std::endl;
```

```
    }
};

int main() {
    Box<int> intBox(10);
    Box<double> doubleBox(20.5);

    intBox.display();
    doubleBox.display();
    return 0;
}
```

Modern C++ Features: Overview

This section explores the features introduced in C++11, C++14, and C++17, including examples and steps to compile them.

C++11 Features

1. auto Keyword

Automatically deduces the type of a variable at compile time.

```
#include <iostream>
int main() {
    auto x = 42;     // int
    auto y = 3.14;   // double
    auto str = "C++"; // const char*
    std::cout << x << " " << y << " " << str << std::endl;
```

```
    return 0;
}
```

Compilation:

```
g++ -std=c++11 auto_example. -o auto_example
```

---

2. nullptr

A new keyword to represent a null pointer.

```
#include <iostream>
void func(int* ptr) {
    if (ptr == nullptr) {
        std::cout << "Pointer is null.\n";
    }
}
int main() {
    func(nullptr); // Better than using NULL or 0
    return 0;
}
```

Compilation:

```
g++ -std=c++11 nullptr_example. -o nullptr_example
```

---

3. Strongly-Typed Enums

Enums now have scope and type safety.

```
#include <iostream>

enum class Color { Red, Green, Blue }; // Scoped enum

int main() {
   Color c = Color::Red;
   if (c == Color::Red) {
      std::cout << "Color is Red.\n";
   }
   return 0;
}
```

Compilation:

```
g++ -std=c++11 enum_example. -o enum_example
```

---

4. Move Semantics

Efficiently transfer ownership of resources.

```
#include <iostream>
#include <vector>

int main() {
   std::vector<int> v1 = {1, 2, 3};
   std::vector<int> v2 = std::move(v1); // v1 is now empty
   std::cout << "v2 size: " << v2.size() << "\n";
   return 0;
}
```

Compilation:

g++ -std=c++11 move_example. -o move_example

---

5. Range-Based Loops

Simplifies iteration over containers.

```
#include <iostream>
#include <vector>
int main() {
   std::vector<int> vec = {1, 2, 3, 4};
   for (int num : vec) {
      std::cout << num << " ";
   }
   return 0;
}
```

Compilation:

g++ -std=c++11 range_loop. -o range_loop

---

6. Lambdas

Compact, inline functions.

```
#include <iostream>
#include <vector>
#include <algorithm>
int main() {
   std::vector<int> vec = {3, 1, 4, 1, 5};
```

```
    std::sort(vec.begin(), vec.end(), [](int a, int b) { return a < b; });
    for (int n : vec) {
        std::cout << n << " ";
    }
    return 0;
}
```

Compilation:

```
g++ -std=c++11 lambda_example. -o lambda_example
```

C++14 Features

1. Generalized Lambdas

Allows lambda captures of std::unique_ptr or any object.

```
#include <iostream>
#include <memory>
int main() {
    auto ptr = std::make_unique<int>(42);
    auto lambda = [value = std::move(ptr)]() {
        std::cout << *value << std::endl;
    };
    lambda();
    return 0;
```

```
}
```

Compilation:

```
g++ -std=c++14 generalized_lambda. -o generalized_lambda
```

---

2. constexpr Enhancements

constexpr functions can have loops and branches.

```
#include <iostream>
constexpr int factorial(int n) {
   return n <= 1 ? 1 : n * factorial(n - 1);
}
int main() {
   constexpr int result = factorial(5);
   std::cout << "Factorial: " << result << std::endl;
   return 0;
}
```

Compilation:

```
g++ -std=c++14 constexpr_example. -o constexpr_example
```

C++17 Features

1. Fold Expressions

Simplifies variadic template operations.

```
#include <iostream>
template<typename... Args>
void print_sum(Args... args) {
   std::cout << (args + ...) << std::endl; // Right fold
}
int main() {
   print_sum(1, 2, 3, 4); // Outputs 10
   return 0;
}
```

Compilation:

```
g++ -std=c++17 fold_expression. -o fold_expression
```

---

2. Class Template Argument Deduction

No need to specify template arguments explicitly.

```
#include <iostream>
#include <vector>
int main() {
   std::vector vec = {1, 2, 3, 4}; // Template arguments deduced
   for (int n : vec) {
      std::cout << n << " ";
```

```
  }
  return 0;
}
```

Compilation:

```
g++ -std=c++17 ctad_example. -o ctad_example
```

---

3. Reader-Writer Locks

Provides shared mutex for multiple readers or a single writer.

```
#include <iostream>
#include <shared_mutex>
#include <thread>
#include <vector>

std::shared_mutex rw_lock;
int shared_resource = 0;

void read_data() {
  std::shared_lock<std::shared_mutex> lock(rw_lock);
  std::cout << "Read: " << shared_resource << std::endl;
}

void write_data(int value) {
  std::unique_lock<std::shared_mutex> lock(rw_lock);
```

```
    shared_resource = value;
    std::cout << "Written: " << value << std::endl;
}

int main() {
    std::thread writer1(write_data, 10);
    std::thread writer2(write_data, 20);
    std::thread reader1(read_data);
    std::thread reader2(read_data);

    writer1.join();
    writer2.join();
    reader1.join();
    reader2.join();
    return 0;
}
```

Compilation:

```
g++ -std=c++17 rw_lock_example. -o rw_lock_example -lpthread
```

---

4. Transactional Memory (Experimental)

Handles atomic operations. Use with caution as it depends on compiler support.

Build Tool Chain

What is a Build Tool Chain?

Imagine you're building a toy car. You have parts like wheels, a body, and an engine. To make the toy car work, you need to assemble all the parts step by step. A **build tool chain** does the same thing for software. It helps take all the parts of a program (like pieces of code) and assembles them into something that works, like an app or game.

Why Do We Use Build Tools?

Software is like a puzzle made of many pieces. Build tools help us:

- **Assemble the pieces**: Combine code files into one program.
- **Fix errors**: Show where we made mistakes.
- **Optimize**: Make the program faster or smaller.

Let's Meet g++

What is g++?

g++ is like a robot that helps you build your software car! It reads your C++ code and turns it into a program your computer can run.

Steps in the Build Process:

When we use g++, it goes through these steps:

1. **Preprocessing**: Cleans and prepares your code.
2. **Compiling**: Translates your code into an intermediate language.
3. **Assembling**: Turns the intermediate language into machine code (a language your computer understands).
4. **Linking**: Combines different parts and libraries into one executable file.

Example: Building a Simple Calculator Program

Let’s make a program that adds two numbers together.

Step 1: Write the Code

Create two files:

main.cpp

```
#include <iostream>

#include "calculator.h"


int main() {

   int a = 5, b = 10;

   std::cout << "The sum is: " << add(a, b) << std::endl;

   return 0;

}
```

calculator.h

```
int add(int a, int b);
```

calculator.cpp

```
#include "calculator.h"


int add(int a, int b) {

   return a + b;

}
```

Step 2: Compile the Code

Open a terminal and run:

Compile each file into object files:

g++ -c main. -o main.o

g++ -c calculator. -o calculator.o

This creates main.o and calculator.o.

Link the object files into an executable:

g++ main.o calculator.o -o calculator

Step 3: Run the Program

Run the program with:

./calculator

You should see:

The sum is: 15

Cool Tricks with g++

Compiler Flags:

- -Wall: Show all warnings to catch bugs early.
  Example: g++ -Wall main. -o main
- -O2: Optimize the program for better performance.
  Example: g++ -O2 main. -o main

Debugging:

- Use -g to add debugging information.
  Example: g++ -g main. -o main
- Use a debugger like gdb to find bugs.

Static vs. Dynamic Linking:

- **Static Linking**: Include everything the program needs in one file.
  Example: g++ main.o calculator.o -o calculator

- **Dynamic Linking**: Share libraries to save space.
  Example: g++ main.o -L. -lcalculator -o calculator

In Short

- A build tool chain assembles your code step by step.
- g++ is a powerful tool for building C++ programs.
- With g++, you can compile, optimize, and debug your code.
- Practice by building simple programs like our calculator!

Multithreading in C++

Multi-threading in C++

What is Multi-threading?

Imagine you are building a toy train set with a friend. Instead of you doing everything alone, your friend helps by doing part of the work. Multi-threading is when a program works on different parts of a task at the same time, like teamwork for your computer!

Step-by-Step Guide to Multi-threading in C++

1. Introduction to Multi-threading

- **Parallelism**: Doing many things at once.
- **Concurrency**: Switching between tasks quickly to appear like doing many things at once.

---

2. Basics of Threads in C++

A thread is like a helper who works on a small job while the main program works on another. In C++, threads are in the std::thread library.

---

3. Working with Threads

Example 1: Creating Threads
Let's make a program where one thread calculates the sum of numbers while the main thread prints "Hello!"

```
#include <iostream>
#include <thread>

// Function for the thread
```

```
void calculateSum() {
   int sum = 0;
   for (int i = 1; i <= 5; ++i) {
      sum += i;
   }
   std::cout << "Sum: " << sum << std::endl;
}


int main() {
   // Create a thread to calculate sum
   std::thread t(calculateSum);

   // Main thread prints a message
   std::cout << "Hello from the main thread!" << std::endl;

   // Wait for the thread to finish
   t.join();

   return 0;
}
```

How to Run:

1. Save this as threads.cpp.
2. Compile:

g++ -std=c++11 threads.cpp -o threads -pthread

3. Run:

./threads

Output:

css

Hello from the main thread!

Sum: 15

---

4. Thread Management

- Use .join() to wait for a thread to finish.
- Use .detach() to let a thread run independently.

---

5. Using Lambda Expressions with Threads

You can define tasks inline using a lambda expression.

Example 2: Threads with Lambdas

```
#include <iostream>
#include <thread>

int main() {
  std::thread t([]() {
```

```
    std::cout << "Hello from Lambda Thread!" << std::endl;
  });

  t.join();
  return 0;
}
```

---

6. Synchronization and Advanced Concepts

Problem: Race Conditions

When two threads access shared data at the same time, bad things can happen!

Solution: Mutexes
A mutex locks shared data so only one thread can access it at a time.

Example 3: Using Mutex

```
#include <iostream>
#include <thread>
#include <mutex>

std::mutex mtx;

void printMessage(const std::string &message) {
  mtx.lock();
  std::cout << message << std::endl;
  mtx.unlock();
}
```

```
int main() {
   std::thread t1(printMessage, "Hello from Thread 1");
   std::thread t2(printMessage, "Hello from Thread 2");

   t1.join();
   t2.join();

   return 0;
}
```

---

7. Advanced Multi-threading Techniques

Condition Variables

Condition variables allow threads to wait for a signal to continue.

Example 4: Using Condition Variables

```
#include <iostream>
#include <thread>
#include <condition_variable>
#include <mutex>

std::mutex mtx;
std::condition_variable cv;
bool ready = false;
```

```
void worker() {
   std::unique_lock<std::mutex> lock(mtx);
   cv.wait(lock, [] { return ready; });
   std::cout << "Worker is running!" << std::endl;
}

int main() {
   std::thread t(worker);

   {
      std::lock_guard<std::mutex> lock(mtx);
      ready = true;
   }
   cv.notify_one();

   t.join();
   return 0;
}
```

Thread Pools

A thread pool is like a team of workers who pick up jobs from a queue.

Example 5: Simple Thread Pool

```
#include <iostream>
```

```
#include <thread>
#include <vector>
#include <queue>
#include <functional>
#include <mutex>
#include <condition_variable>

class ThreadPool {
  std::vector<std::thread> workers;
  std::queue<std::function<void()>> tasks;
  std::mutex mtx;
  std::condition_variable cv;
  bool stop = false;

public:
  ThreadPool(size_t numThreads) {
    for (size_t i = 0; i < numThreads; ++i) {
      workers.emplace_back([this] {
        while (true) {
          std::function<void()> task;
          {
            std::unique_lock<std::mutex> lock(mtx);
            cv.wait(lock, [this] { return !tasks.empty() || stop; });
            if (stop && tasks.empty()) return;
            task = std::move(tasks.front());
```

```
            tasks.pop();
        }
        task();
      }
    });
  }
}

void enqueue(std::function<void()> task) {
  {
    std::lock_guard<std::mutex> lock(mtx);
    tasks.push(task);
  }
  cv.notify_one();
}

~ThreadPool() {
  {
    std::lock_guard<std::mutex> lock(mtx);
    stop = true;
  }
  cv.notify_all();
  for (auto &worker : workers) {
    worker.join();
  }
```

```
    }
};

void sayHello(int id) {
    std::cout << "Hello from Task " << id << std::endl;
}

int main() {
    ThreadPool pool(3);

    for (int i = 0; i < 5; ++i) {
        pool.enqueue([i] { sayHello(i); });
    }

    return 0;
}
```

In Short

1. Use std::thread to create and manage threads.
2. Use mutexes and locks to avoid race conditions.
3. Advanced techniques like condition variables and thread pools help manage complex tasks.
   Now, you're ready to make your programs faster with multi-threading!