CS100 Recitation 8

Contents

- C++ Standard Library
  - `std::string` & `std::vector`
- References
- Dynamic Memory Management: `new` and `delete`

C++ Standard Library

- **Namespace**
- `std::string`
- `std::vector`

Namespaces

- Defining a Namespace
- Scope Resolution Operator `::`
- Nested Namespaces
- `using`-Declarations
- `using`-Directives
- Namespace `std`

Namespaces

- **Purpose**: Mitigate identifier conflicts in complex software systems, especially in projects with multiple files.
- **Scope**:
  - Entities declared within a namespace occupy a distinct **namespace scope**, ensuring they do not conflict with identically named entities elsewhere.
  - Entities declared outside any namespace belong to the **global namespace**.

```
namespace MathOperations {
  int add(int a, int b) {
    return a + b;
  }
}

// Global namespace
int add(int a, int b) {
  return a - b;
}

int main() {
  int x = 10, y = 5;

  add(x, y);

  MathOperations::add(x, y);

  return 0;
}
```

Namespaces: Defining a Namespace

```
namespace namespace_name {
  int a;              // Variable declaration
  void add() {        // Function declaration
    // ...
  }
  class ClassName { // Class declaration
    // ...
  };
}
```

- No semicolon `;` is required after the closing brace of a namespace.
- **Extension:** A namespace can be re-opened to add further declarations.

```
namespace namespace_name{
  // Additional declarations add to the existing namespace.
}
```

Scope Resolution Operator `::`

Definition:

The operator `::` is used to explicitly specify the scope of an identifier, ensuring clarity in situations where multiple scopes define the same name.

Use Cases:

- **Global Variable Access:** When a local variable shadows a global one.

```
int x = 10; // Global variable
int main() {
  int x = 20; // Local variable
  std::cout << ::x; // Outputs the global 'x' (10)
}
```

Scope Resolution Operator `::` (Continued)

- **Accessing Namespace Members:** Explicitly refer to an entity within a namespace.

```
namespace Math {
  int add(int a, int b) { return a + b; }
}
int main(){
  std::cout << Math::add(5, 3); // Accesses 'add' within namespace Math
}
```

- **Defining Class Methods Outside the Class Body:** Clarify the method's context.

```
class MyClass {
  void display();
};
void MyClass::display() { // Definition outside the class scope
  std::cout << "Hello!";
}
```

Namespaces: Nested Namespaces

Namespaces can be **nested**, meaning you can define one namespace inside another. The syntax for this is:

```
namespace outer_namespace {
  namespace inner_namespace {
    // Code here
  }
}
```

This is equivalent to: (C++17 and later)

```
namespace outer_namespace::inner_namespace {
  // Code here
}
```

Exercise 1: What's the output?

```
#include <iostream>
namespace first_space {
  void func(){
    std::cout << "Inside first_space" << std::endl;
  }

  namespace second_space {
    void func(){
      std::cout << "Inside second_space" << std::endl;
    }
  }
}
int main(){
  first_space::func();
  first_space::second_space::func();
  return 0;
}
```

`using`-Declarations

A `using`-declaration introduces a specific name from another namespace into the current scope. This facilitates more concise code without sacrificing clarity.

1. In Namespace Scope

```
void f();

namespace A {
  void g();
}

namespace X {
  using ::f;  // Global 'f' is accessible as X::f.
  using A::g; // A::g is accessible as X::g.
}
```

`using`-Declarations (cont.)

2. In Function (Block) Scope.

```
using std::cout;
using std::endl;

int main() {
  std::string first_name;
  std::cin >> first_name;

  // No need to prefix with `std::`.
  cout << "Hello " << first_name << endl;

  return 0;
}
```

`using` -Declarations (cont.)

*3. In Class Definition

A `using` -declaration can introduce base class members into the derived class. This is useful to expose protected or private members of the base class as public members of the derived class.

```
class Base {
protected:
  int protected_member;
};

class Derived : public Base {
public:
  using Base::protected_member; // Exposes the protected member as public.
};
```

Exercise 2

Determine whether the following code will compile successfully:

```
#include <iostream>

void foo(){
  using std::cout;
  cout << "In foo()";
}

int main(){
  foo();
  cout << "In main()" << std::endl;
}
```

Hint: Consider the scope of the `using` declaration and its effect in different function bodies.

`using`-Directives

A *using directive* imports **all identifiers** from a specified namespace into the current scope, making them accessible as if they were declared there.

- Unlike *using declarations*, a using directive does **NOT** add names to the declarative region, and thus does not prevent re-declaration of identifiers.

```
using namespace std;

int main(){
  cout << "Hello World!" << endl;
}
```

Important: Avoid using a directive in header files or global scopes where name collisions may arise.

`using`-Directives

```
#include <iostream>
namespace A{
  namespace B {
    int k = 9;  // A::B::k
  }
  using namespace B;
  int k = 10; // A::k
  int l = k;  // error
}
int main(){
  std::cout << A::l;
}
```

A reference to `k` in namespace `A` is ambiguous because `A::k` and `A::B::k` are both visible.

```
#include <iostream>
namespace B {
  int k = 9;  // B::k
}
namespace A{
  using namespace B;
  int k = 10; // A::k
  int l = k;  // OK
}
int main(){
    std::cout << A::l;
}
```

- A name introduced in the current scope always takes precedence over names from an outer scope.
- In this case, `A::k` hides `B::k` within namespace `A`.

Namespace `std`

The C++ standard library comprises numerous identifiers. To prevent name collisions, all these identifiers are encapsulated within the namespace `std`.

- Using `using std::cin;` brings only `std::cin` into the current scope, allowing you to refer to it simply as `cin`.
- Using `using namespace std;` imports all identifiers from `std` into the current scope, but this increases the risk of name conflicts.

Recommendation:

For clarity and to avoid potential conflicts, explicitly prefix standard library objects and functions with `std::` (e.g., `std::cout`, `std::cin`, `std::endl`).

`std::string`

- Overview
- Library Interfaces
- String I/O

`std::string`: Overview

- `std::string` is a type alias for `std::basic_string<char>`. The template `std::basic_string` can manage sequences of various character types such as `char`, `char8_t`, `char16_t`, and `char32_t`.
- Memory for `std::string` is managed automatically. Allocation and deallocation occur as objects are created and go out of scope, and operations like concatenation and insertion are handled transparently.
- When using `std::string` or other STL containers (e.g., `std::vector<T>`), focus on their interfaces and functions instead of their internal implementation details.

Library Interfaces of `std::string`

- **Constructors**
- **Element Access:** e.g., `operator[]`
- **Concatenation:** `operator+`
- **Comparison Operators:** `operator==`, `operator!=`, `operator<`, etc.

Constructor

Examples:

- **Default Constructor:**

```
std::string s;    // Creates an empty string.
```

- **Fill Constructor:**

```
std::string s(n, ch);  // e.g., std::string s(5, 'a'); creates "aaaaa"
```

- **C-Style String Constructor:**

```
std::string s("Hello");
```

For further constructors, refer to cppreference.

`operator[]`

Non-const Version:

```
char& operator[](size_t pos);
```

Const Version:

```
const char& operator[](size_t pos);
```

The subscript operator for `std::string` is ***overloaded*** to support both modifiable and non-modifiable access:

```
std::string s("Exampl ");
s[s.size() - 1] = 'e';
std::cout << s << '\n';   // outputs: "Example"
```

```
const std::string e("Example");
for (unsigned i = e.length() - 1; i != 0; i /= 2)
  std::cout << e[i];
std::cout << '\n';  // outputs: "emx"
```

`operator+`

The concatenation operator (`operator+`) is overloaded for `std::string` . At least one operand must be of type `std::string` to invoke the appropriate overload.

```
std::string s1{"hello"};
std::string s2 = "world";
std::string s3 = s1 + "world" + "C++"; // s1 is used to initiate concatenation
s1 += s2;
```

Note: String literals (of type `const char[N]`) cannot be concatenated directly with another literal using `+` , as both operands must be either converted to or be `std::string` . For instance, the following results in a compile-time error:

```
std::string str = "Hello " + "World"; // error: no matching operator+
```

Comparison Operators for `std::string`

Supported operators include:

- **Equality and Inequality:** `operator==`, `operator!=`
- **Relational Operators:** `operator<`, `operator<=`, `operator>`, `operator>=`

Lexicographical Comparison:

Based on element-wise comparison (see std::lexicographical_compare):

- An empty sequence is considered the smallest.
- Comparison stops at the first mismatching element.
- If one string is a prefix of another, the shorter is considered smaller.

String I/O

- `std::cin >> s` : Skips leading whitespaces and stops at the first whitespace.
- `std::cout << s` : Does not append a newline automatically.

```
#include <iostream>
#include <string>
int main() {
  std::string str; std::cin >> str;
  std::cout << str << str;
}
```

Example:

If the input is `"abcd efg hij"` , the output will be:

```
abcdabcd
```

- `std::getline` : Reads a string starting from the current input position until the first occurrence of `'\n'` . The newline character is consumed but not stored. If the previous input left the stream positioned at `'\n'` , an empty string will be read.

`std::vector`

- `std::vector` & Containers Library
- Overview
- Growth Strategy
- Library Interfaces

`std::vector` & Containers Library

- `std::vector` is the first container of the STL we learn and will be reviewed in upcoming lectures on the STL.
    - Note that although `std::string` is not formally regarded as a container, it behaves similarly and is sometimes referred to as a "pseudo container."
- For further reference, consult the table of all C++ standard library containers summarized by cppreference: cppreference Containers. This table shows that the interfaces introduced for `std::vector` are uniformly supported by other containers. ***This reflects the C++ standard library's design philosophy of a consistent interface across containers.***
    - Each cell in the table is a link to more detailed information.

`std::vector`: Overview

- `std::vector` is a sequence container that encapsulates dynamic arrays whose size can change at runtime.
- Its elements are stored contiguously, allowing access via iterators as well as through **pointer arithmetic**. This contiguous storage ensures that a pointer to an element can be passed to functions that expect a pointer to an array.

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

void printArray(int* arr, size_t size){
  for (size_t i = 0; i < size; i++){
    std::cout << arr[i] << " ";
  }
  std::cout << "\n";
}
```

```cpp
int main(){
  std::vector<int> v =
        {10, 20, 30, 40, 50};

  int* p = &v[0];
  printArray(p, v.size());

  return 0;
}
```

`std::vector` Growth Strategy

`std::vector` is a dynamic array in C++ that manages its own memory automatically. Its growth strategy is designed to optimize performance by minimizing the frequency of memory reallocations.

- As elements are inserted, `std::vector` expands its storage capacity **dynamically**.
- It typically allocates extra space beyond the current number of elements, anticipating future insertions and reducing the need for frequent reallocations.
- The `capacity()` function returns the total amount of memory currently allocated, which may exceed the number of stored elements.
- To release any unused memory, the `shrink_to_fit()` function can be called, which requests the reduction of the vector's capacity to match its size.

Naïve Growth Strategy

Assume that the vector currently has allocated memory for i elements. A naïve approach for inserting the $(i+1)$th element would be to:

1. Allocate a new memory block of size $i+1$.
2. Copy the existing i elements to the new block.
3. Append the new element.
4. Release the previous memory block.

This method requires copying i elements for each insertion, leading to a total of

$$\sum_{i=0}^{n-1} i = O(n^2)$$

copy operations, which is computationally inefficient.

Optimized Growth Strategy

A common optimization technique adopts **exponential growth** to reduce the frequency of memory reallocations. When inserting the $(i+1)$-th element into a dynamic array:

1. Allocate new memory with capacity approximately $2 \cdot i$ (i.e., double the current capacity).
2. Copy the existing i elements into the newly allocated memory.
3. Insert the new element.
4. Deallocate the old memory block.

This strategy ensures that the remaining capacity can accommodate future insertions—from the $(i+2)$-th up to the $2 \cdot i$-th element—**without further reallocations or copying**. As a result, the `push_back` operation achieves **amortized constant time complexity**, i.e., each insertion runs in $O(1)$ time on average.

Optimized Growth Strategy

Note: This doubling strategy is commonly used but is not mandated by the C++ standard. You can inspect the actual growth behavior using the `capacity()` method of `std::vector`.

Risks of Modifying Containers During Iteration

Dynamic growth in `std::vector` may **reallocate internal storage**, which **invalidates all pointers, references, and iterators** to its elements.

- A critical consequence: modifying the container (e.g., using `push_back`) during iteration leads to **undefined behavior**. The following example illustrates this issue:

```
for (int i : vec)
  if (i % 2 == 0)
    vec.push_back(i + 1);
```

  This loop is unsafe because it relies on iterators that may be invalidated when the vector resizes during `push_back`.

Never alter the size of a container while iterating over it using a range-based for loop.

Library Interfaces

The C++ Standard Template Library (STL) defines a standardized set of interfaces for container types. `std::vector`, as a representative container, supports a wide range of operations.

Here, we will take a **quick look** at a few selected categories:

- **Element Access**
- **Capacity Queries**
- **Modifiers**

Element Access

Member Function	Description
`at`	Accesses a specified element with bounds checking
`operator[]`	Accesses a specified element without bounds checking
`front`	Accesses the first element
`back`	Accesses the last element
`data`	Provides direct access to the underlying contiguous storage

Capacity

Member Function	Description
`empty`	Checks whether the container is empty
`size`	Returns the number of elements currently stored
`max_size`	Returns the maximum number of elements the container can hold
`reserve`	Reserves memory to accommodate at least the specified number of elements
`capacity`	Returns the number of elements that can be held in currently allocated storage
`shrink_to_fit`	Requests the reduction of capacity to fit the current size

Modifiers

Member Function	Description
`clear`	Removes all elements from the container
`insert`	Inserts elements at a specified position
`emplace`	Constructs an element in-place at a specified position
`erase`	Removes elements at a specified position or range
`push_back`	Adds an element to the end
`emplace_back`	Constructs an element in-place at the end
`pop_back`	Removes the last element
`resize`	Changes the number of elements stored

Exercise: Implement Python's `rstrip`

Implement a function similar to Python's `rstrip` which accepts a `std::string` and returns the string after removing trailing contiguous whitespace.

```
std::string rstrip(std::string str) {
  /* Your implementation here */
}
```

Exercise: Implementing `rstrip`

```
std::string rstrip(std::string str) {
  while (!str.empty() && std::isspace(str.back()))
    str.pop_back();
  return str;
}
```

Note:

In C++17, copy elision optimizes the return of `str`, preventing unnecessary copying.

Reference

Motivation

Exercise: Use a range-based `for` loop to iterate over a string and convert all uppercase letters to lowercase.

Consider the following code:

```
for (char c : str)
  c = std::tolower(c);
```

This approach is **ineffective** because `c` is a **copy** of each character in `str`. Modifying `c` does **not** affect the original string.

> To modify the original elements, you must use a reference in the loop.

Motivation (Correct Approach)

Corrected Implementation:

```
for (char &c : str)
  c = std::tolower(c);
```

- Declaring `c` as a **reference** (`char &c`) ensures that you modify the actual character in `str` .
- This is equivalent to the indexed access approach:

```
for (std::size_t i = 0; i != str.size(); ++i) {
  char &c = str[i];
  c = std::tolower(c);
}
```

References as Aliases

```
std::string s;
std::string &r = s;
r.push_back('a');         // equivalent to s.push_back('a');
r = "hello";              // equivalent to s = "hello";
std::cout << r << '\n'; // equivalent to std::cout << s << '\n';
std::string &r2 = r;      // equivalent to std::string &r2 = s;
```

- A **reference must be initialized** upon declaration, and its binding is **immutable** thereafter. **Assignment** through a reference modifies the **bound object**, not the binding itself.
- You may write `Type &r` or `Type& r`; both declare a reference.
  In `Type& r1, r2;`, only `r1` is a reference—write `Type &r1, &r2;` to declare both as references.

References Are Not Objects

Although references must be bound to **objects** (i.e., lvalues), they themselves are **not objects** and do **not necessarily occupy storage**.

As a result, the following are **not allowed** in C++:

- **Arrays of references**
- **Pointers to references**
- **References to references**

```
int& a[3]; // error: array of references
int&* p;   // error: pointer to reference
int& &r;   // error: reference to reference
```

Passing Reference to Functions

Passing by Reference vs. by Pointer

```
void print_array(const int (&array)[10]) { // Read-only: const-qualified
  for (int x : array)    // Using a range-based for loop to iterate
    std::cout << x << ' ';
  std::cout << std::endl;
}
int a[10], ival = 42, b[20];
print_array(a);     // Correct: a is int[10]
print_array(&ival); // Error: &ival is not an array
print_array(b);     // Error: b is int[20]
```

Advantages over pointers:

- Unlike pointers, which require explicitly passing the address using `&`, references allow us to pass variables directly.
- No array-to-pointer decay occurs; the reference binds to the entire array. Therefore, arrays of incorrect size are rejected at compile time.

Passing by Reference vs. by Value

> When parameters are passed **by value**, the function receives a copy of the argument.
>
> As a result:
>
> - The function cannot modify the original argument.
> - Copying large objects may incur performance overhead.

```
void print_upper(std::string str) {
  for (char c : str)
    if (std::isupper(c))
      std::cout << c;
  std::cout << std::endl;
}
```

Using references addresses both issues: it enables modification (when non-const) and avoids the cost of copying large objects.

Passing by Reference vs. by Value (cont.)

```
void print_upper(std::string &str) {
  for (char c : str)
    if (std::isupper(c))
      std::cout << c;
  std::cout << std::endl;
}
std::string s = some_very_long_string();
print_upper(s); // Binds str directly to s, avoiding a copy
```

However, consider the following case:

```
const std::string hello = "Hello";
print_upper(hello); // Error: cannot bind non-const reference to const object
```

Caveat: Non-const references can only bind to modifiable (non-const) objects. This makes them unsuitable for read-only operations on const arguments.

Passing by Constant Reference

For read-only access, the preferred approach is to pass parameters by **const reference**.

This provides the efficiency of reference passing while maintaining safety.

```
void print_upper(const std::string &str) {
  for (char c : str)
    if (std::isupper(c))
      std::cout << c;
  std::cout << std::endl;
}
```

Benefits:

- **Efficiency:** Avoids unnecessary copying.
- **Flexibility:** Accepts both const and non-const arguments.
- **Safety:** Prevents unintended modifications within the function.

`new` and `delete`

The `new` Expression

The `new` expression dynamically allocates memory **and constructs an object**.

```
int *pi1 = new int;        // Default-initialized (undetermined value).
int *pi2 = new int();      // Value-initialized (zero for built-in types).
int *pi3 = new int{};      // Modern syntax for value initialization.
int *pi4 = new int(42);    // Initializes to 42.
int *pi5 = new int{42};    // Equivalent modern syntax.
```

For built-in types:

- **Default Initialization:** Leaves the memory uninitialized.
- **Value Initialization:** Ensures that the memory is zero-initialized.

The `new[]` Expression

The `new[]` expression dynamically allocates an array and **constructs its objects**.

```
int *pai1 = new int[n];
// Dynamically creates an array of n ints, default-initialized
int *pai2 = new int[n]();
// Dynamically creates an array of n ints, value-initialized
int *pai3 = new int[n]{};
// Dynamically creates an array of n ints, value-initialized
```

```
int *pai4 = new int[n]{2, 3, 5};
```

Initializes the first three elements to 2, 3, and 5; remaining elements (if any) are value-initialized. If `n < 3`, throws `std::bad_array_new_length`.

The `delete` and `delete[]` Expressions

Used to destroy dynamically allocated objects and release memory:

```
int *p = new int{42};
delete p;

int *a = new int[n];
delete[] a;
```

- Use `delete` for memory allocated with `new`.
- Use `delete[]` for memory allocated with `new[]`.
- The pointer passed to `delete` must exactly match one returned by `new`; mismatches result in **undefined behavior**.
- Omitting `delete` causes **memory leaks**.

`new`/`delete` vs `malloc`/`free`

Feature	`new` / `delete`	`malloc` / `free`
Memory Allocation	Yes	Yes
Object Construction	Yes (calls constructor)	No
Object Destruction	Yes (calls destructor)	No
Type Safety	Yes	No (requires casting)

In C++, Avoid Unnecessary Manual Memory Management

Modern C++ encourages the use of high-level abstractions to promote code safety, maintainability, and reliability. By leveraging the standard library and smart pointers, developers can minimize errors associated with manual memory management and focus on building robust applications.

Standard Practices for Memory Management

- **Prefer Standard Library Containers:**
  Use containers (e.g., `std::string`, `std::vector`, `std::deque`, `std::list` / `std::forward_list`, `std::map` / `std::set`, `std::unordered_map` / `std::unordered_set`) for managing sequences, collections, tables, or sets to improve safety and maintainability.

- **Favor Smart Pointers for Dynamic Allocation:**
  For a single dynamically allocated object, use smart pointers (e.g., `std::shared_ptr`, `std::unique_ptr`, `std::weak_ptr`).
  Note: The RAII (Resource Acquisition Is Initialization) paradigm, which underlies `std::unique_ptr`, will be introduced in recitations.

When to Use Manual Memory Management

- **Reserving Manual Memory Management:**
  Use direct memory management with `new` / `delete` only in exceptional cases, such as when implementing custom data structures with stringent performance requirements.
- **Specialized Memory Allocation:**
  When precise control over memory allocation is needed, you might resort to the C memory allocation/free functions. Even then, these functions are often employed to customize `new` and `delete` (please refer to GKxx's video for more information).

CS100 Recitation 8