CS100 Recitation 6

Contents

- The Beginning of C++
- About `<iostream>`
- C and C++
- About `std::string`
- About `std::vector`

The Beginning of C++

Preparations

- We adopt the **C++17** language standard.
- We may need **g++** instead of gcc.
- Use one IDE for developing, say VSCode:
  - In `settings.json`, under `code-runner.executorMap`, ensure that the `"cpp"` entry uses the compilation flag `-std=c++17` for the language standard.
  - In `c_cpp_properties.json`, set the `"cppStandard"` entry to `c++17`.
  - Debugging: The simplest and most direct method is to delete both `tasks.json` and `launch.json`. When you debug a C++ program, VSCode will automatically generate these files. And please remember to select `g++.exe`.

Your first C++ program

Please make sure that you can run the following code.

```cpp
#include <iostream>

int main() {
  std::cout << "Hello World!\n";
  return 0;
}
```

To compile it: `g++ fileName.cpp -o execName -std=c++17 -Wall -Wextra -Wpedantic` (replace `fileName.cpp` and `execName` with your desired choices).

About `<iostream>`

`<iostream>`

Example Code

```cpp
#include <iostream>

int main() {
  int a, b;
  std::cin >> a >> b;
  std::cout << a + b << std::endl;
  return 0;
}
```

`std::cin` and `std::cout` are two **objects** defined in `<iostream>`, representing the standard input stream and standard output stream, respectively.

`std::cin` and `std::cout`

- `std::cin >> x` inputs a value into `x`.
  - `x` can be any supported type, such as an integer, float, character, or string.
  - C++ is capable of recognizing the type of `x` and choosing the correct input method without the need for specifiers like `"%d"` or `"%c"`.
  - C++ can obtain a reference to `x`, so there's no need to take the address of `x`.
  - After the expression `std::cin >> x` is executed, `std::cin` is returned, allowing you to chain inputs: `std::cin >> x >> y >> z`.
- Output works in a similar manner: `std::cout << x << y << z`.

`std::endl`

- `std::endl` is a **manipulator**. The meaning of `std::cout << std::endl` is to output a newline character and **flush the output buffer**.
- If you don't manually flush the buffer, `std::cout` will automatically flush the buffer under certain conditions (this is also true for C's stdout).
- In fact, there are a lot more details for this, please refer to this link.

namespace `std`

- C++ has an extensive standard library, and to avoid name collisions, all names (functions, classes, type aliases, templates, global objects, etc.) are placed in a namespace called `std`.
- You can use `using std::cin;` to bring `cin` into the current scope, allowing you to omit the `std::` prefix when using `std::cin` within that scope.
- You can also use `using namespace std;` (not recommanded) to bring all names from `std` into the current scope, but this **makes the namespace effectively useless and reintroduces the risk of name collisions**.
- We **do not** allow the use of any form of `using` in the global scope of header files.

C and C++

Is C++ the more advanced version of C?

Not properly phrased, please recall what we have learnt in class: *C++ can be viewed as a federation of related languages.*

> - C.
> - Object-Oriented C++.
> - Template C++.
> - The STL.

Compatibility with the C Standard Library

The C++ Standard Library includes facilities from the C Standard Library, but they are not the same.

- Due to historical reasons (for backward compatibility), C has many inconsistencies, such as `strchr` accepting a `const char *` but returning a `char *`, and some entities that should be functions are implemented as macros.
- C lacks mechanisms like function overloading available in C++, making certain designs more cumbersome.
- C++ has significantly more powerful compile-time computation capabilities than C. For instance, starting from C++23, the mathematical functions in `<cmath>` can be computed at compile-time.

Compatibility with the C Standard Library

The C Standard Library headers `<xxx.h>` in C++ are available as `<cxxx>`, with all names placed in the `std` namespace.

```cpp
#include <cstdio>
int main() { std::printf("Hello world\n"); return 0;}
```

*** When using C Standard Library headers in C++, use `<cxxx>` instead of `<xxx.h>`.**

C in C++

More reasonable design choices

- `bool`, `true`, and `false` are built-in types and do not require additional headers.
- Logical and relational operators return `bool` rather than `int`.
- The type of a character literal `'a'` is `char` rather than `int`.
- The type of `"hello"` is `const char[6]` rather than `char[6]`.
- Implicit conversions with potential risks are not allowed, and they generate errors instead of warnings.
- A `const int maxn = 100;` declaration makes `maxn` a compile-time constant, which can be used as the size of an array.
- `int fun()` takes no arguments rather than accepting arbitrary arguments.

About `std::string`

`std::string`

- **Defined in standard library header `<string>`**
- When using `std::string`, focus on the content of the string itself, not its implementation details.
  - You no longer need to worry about how its memory is managed or whether it ends with `'\0'`.
  - Memory management of `std::string`: automatically managed, dynamically allocated, grows as needed, and automatically released.

Construction

```
std::string str = "Hello world";
// equivalent: std::string str("Hello world");
// equivalent: std::string str{"Hello world"}; (modern)
std::cout << str << std::endl;

std::string s1(7, 'a');
std::cout << s1 << std::endl; // aaaaaaa

std::string s2 = s1; // s2 is a copy of s1
// equivalent: std::string s2(s1);
std::cout << s2 << std::endl; // aaaaaaa

std::string s; // "" (empty string)
```

See https://en.cppreference.com/w/cpp/string/basic_string/basic_string for more.

Input and Output

- `std::cin >> s` for **input** and `std::cout << s` for **output**.
  - **Exercise 1**

    Does `std::cin` ignore leading whitespaces? And what are its ending conditions? Please look it up.
- `std::getline(std::cin, s)`
  - reads a line starting from the current position, the newline character is consumed but not stored.

The length of a string

Motivation: we want to know the number of characters within a string.

- Member function **s.size()**

  Returns the number of `CharT` elements in the string with return type `size_type`.

```
std::string str{"Hello world"};
std::cout << str.size() << std::endl;
```

- Member function **s.empty()**

  Checks if the string has no characters. `true` if the string is empty, `false` otherwise.

Concatenation

Use operators like `+` and `+=`.

There's no need to worry about the memory management, thus no need to use functions like `strcat`.

```cpp
std::string s1 = "Hello";
std::string s2 = "world";
std::string s3 = s1 + ' ' + s2; // "Hello world"
s1 += s2; // s1 becomes "Helloworld"
s2 += "C++string"; // s2 becomes "worldC++string"
```

Operator +

At least one side of `+` must be an object of `std::string`.

```
const char *old_bad_ugly_c_style_string = "hello";
std::string s = old_bad_ugly_c_style_string + "aaaaa"; // Error
```

Exercise 2

check if the following code is legal and give your reasons.

```
std::string hello{"hello"};
std::string s = hello + "world" + "C++";
```

Operator +=

- `s1 = s1 + s2` first creates a temporary object for `s1 + s2`, which necessarily involves copying the contents of `s1`.
- On the other hand, `s1 += s2` directly appends `s2` to `s1`.
- As a result, we may prefer `+=` in certain cases (but not all cases).

Comparation

- Comparations between a pair of strings are based on lexicographic order.
  - e.g. (1, 10, 11, 12, 2, 3, 4, 5, 6, 7, 8, 9) is a lexicographic sort on integers from 1 to 12.
- In `std::string`, we shall use `<`, `<=`, `>`, `>=`, `==` and `!=`.

Iterating over a string: Range-based `for` loop

- Example: Output all uppercase letters (`std::isupper` is in `<cctype>`)

```cpp
for (char c : s)
  if (std::isupper(c))
    std::cout << c;
std::cout << std::endl;
```

```cpp
for (std::size_t i = 0; i != s.size(); ++i)
  if (std::isupper(s[i]))
    std::cout << s[i];
std::cout << std::endl;
```

- Range-based `for` loops are clearer, more concise, more general, more modern, and **highly recommended**.
- Your intent is to "iterate over the string," not to "create an integer and increment it from 0 to `s.size()` ."

About `std::vector`

- **Defined in standard library** `<vector>`
- Reference: https://en.cppreference.com/w/cpp/container/vector.

Basic

`std::vector` is a class template that becomes an actual type only after providing template parameters.

`std::vector` do have a abstract algorithm prototype, which we will not cover in CS100.

```cpp
std::vector<int> vi;         // An empty vector of ints
std::vector<std::string> vs; // An empty vector of strings
std::vector<double> vd;      // An empty vector of doubles
std::vector<std::vector<int>> v; // 2D empty vector of vectors
```

- For two different types `T` and `U`, `std::vector<T>` and `std::vector<U>` are distinct types.
- `std::vector` is not a type itself.

Construction

```
std::vector<int> v{2, 3, 5, 7}; // A vector of ints,
                                // whose elements are {2, 3, 5, 7}.
std::vector<int> v2 = {2, 3, 5, 7}; // Equivalent to ↑

std::vector<std::string> vs{"hello", "world"}; // A vector of strings,
                          // whose elements are {"hello", "world"}.
std::vector<std::string> vs2 = {"hello", "world"}; // Equivalent to ↑

std::vector<int> v3(10); // A vector of ten ints, all initialized to 0.
std::vector<int> v4(10, 42); // A vector of ten ints, all initialized to 42.
```

The construction `vector<T> v(n)` will **value-initialize** the `n` elements (similar to "zero initialization" in C) rather than producing indeterminate values.

- For class types, "value initialization" is almost equivalent to calling the default constructor.
- Like us, the standard library despises uninitialized values!

Copy

```
std::vector<int> v{2, 3, 5, 7};
std::vector<int> v2 = v; // v2 is a copy of v
std::vector<int> v3(v);  // Equivalent
std::vector<int> v4{v};  // Equivalent
```

C++17 CTAD

Class Template Argument Deduction: As long as you provide enough information, the compiler can deduce the element type automatically.

```
std::vector v{2, 3, 5, 7};  // vector<int>
std::vector v2{3.14, 6.28}; // vector<double>
std::vector v3(10, 42);     // vector<int>
std::vector v4(10);         // Error: cannot deduce template argument type
```

Check number of elements in `std::vector`

- `v.size()` and `v.empty()`

```cpp
std::vector v{2, 3, 5, 7};
std::cout << v.size() << std::endl;
if (v.empty()) {
  // ...
}
```

Accessing elements by index

You can use `v[i]` to access the i-th element.

- The valid range for `i` is `$[0, N)$`, where `N = v.size()`.
- Out-of-bounds access is undefined behavior and often results in severe runtime errors.
- The subscript operator `v[i]` does not check for out-of-bounds access, in order to ensure efficiency.

One way to both check bounds and access elements is to use `v.at(i)`, which throws a `std::out_of_range` exception when out of bounds.

Access the first and last element

`v.back()` and `v.front()`

- These functions return **references** to the last and first elements, respectively.
- "Reference" means you can modify them through these member functions:

```
v.front() = 42;
++v.back();
```

When `v` is empty, invoking `back` and `front` are **undefined behaviours**. That is, they do not check bounds!

Add elements at the end

`v.push_back(x)`

add elementes `x` at the end of `v`.

```cpp
int n;
std::cin >> n;
std::vector<int> v;
for (int i = 0; i != n; ++i) {
  int x;
  std::cin >> x;
  v.push_back(x);
}
```

Remove Elements

`v.pop_back()`

Removes the last element of the vector. **Calling pop_back on an empty container results in undefined behavior.**

`v.erase(args)`

To remove a specific or range of elements

`v.clear()`

Erases all elements from the container. After this call, `size()` returns zero.

Exercise 3

Given a container `std::vector<int>`, remove the even elements from the back until the last element is odd.

Range-based `for` loops

You can use a range-based `for` loop to iterate over a `std::vector`:

```
std::vector<int> vi = some_values();
for (int x : vi)
  std::cout << x << std::endl;
std::vector<std::string> vs = some_strings();
for (const std::string &s : vs) // use reference-to-const to avoid copying
  std::cout << s << std::endl;
```

Exercise 4

Use a range-based `for` loop to print all the uppercase letters from each string in a `vector<string>`.

Growth strategy of `std::vector`

Consider the following code that builds a vector by repeatedly calling `push_back` n times:

```cpp
std::vector<int> v;
for (int i = 0; i != n; ++i)
  v.push_back(i);
```

How does vector manage to grow efficiently?

Growth strategy of `std::vector`

Assume there is a dynamically allocated memory block of length `i`.

When the `i+1`th element arrives, a naive approach would be:

1. Allocate a block of memory with length `i+1`.
2. Copy the `i` existing elements over.
3. Place the new element at the end.
4. Free the original memory block.

But this requires copying `i` elements. Performing n `push_back` operations would result in $\sum_{i=0}^{n-1} i = O\left(n^2\right)$ copies!

Growth strategy of `std::vector`

Assume there is a dynamically allocated memory block of length `i`.

When the `i+1`th element arrives,

1. Allocate a block of memory with length `2*i`.
2. Copy the `i` existing elements over.
3. Place the new element at the end.
4. Free the original memory block.

For the next `i+2, i+3, ..., 2*i` elements, no new memory is allocated, and no objects are copied!

Growth strategy of `std::vector`

$0 \to 1 \to 2 \to 4 \to 8 \to 16 \to \cdots$

- Assuming $n = 2^m$, the total number of copies is $\displaystyle\sum_{i=0}^{m-1} 2^i = O(n)$, so the average time complexity per `push_back` operation is $O(1)$ (constant), which is acceptable.
- We can use `v.capacity()` to check the current memory capacity allocated for v and verify if this is really happening.
  - *Note: This is just one possible strategy, and the standard does not mandate it.*

The impact of dynamic growth on `std::vector`

- As we've seen, changing the size of a vector can cause its elements to be "moved," which invalidates all pointers, references, and iterators pointing to those elements.
- The most direct consequence: the following code results in undefined behavior because a range-based for loop fundamentally relies on iterators.

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

Do not modify the size of a container while iterating over it using a range-based for loop!