CS100 Lecture 11

C++ Introduction, Strings

Contents

- Brief history of C++
- Basic IO
- Standard library
- `std::string`

Brief history of C++

Cppcon 2021 The C++ Conference | October 24-29

Bjarne Stroustrup

C++20: Reaching for the Aims of C++

Video Sponsorship Provided By:

ansatz

Morgan Stanley

Bell Labs 1979

Dennis Ritchie

Innovation = invention + development

C
C++
awk
Unix
R
...

Transistors
CCDs
Fibers
...

Doug McIlroy

Al Aho

Brian Kernighan

Rich technical culture

Huge range of interesting technical problems

C with Classes

Back to 1979, the Bell Labs: *C with Classes* was invented by Bjarne Stroustrup.

- An object-oriented C, with the ideas of "class" from Simula (and several other programming languages).
- Based on C, with many improvements.

The birth of C++

After C with Classes was seen as a "medium success" by Stroustrup, he moved on to make a better new language - C++ (1983).

~~C++ is an object-oriented programming language.~~

C++ is a **multi-paradigm** programming language that

- is a better C,
- supports data abstraction, and
- supports object-oriented programming.

Standardization of C++

Standardization: **C++98**, **C++11**, C++14, C++17, **C++20**, C++23, C++26, ...

- C++98: The first ISO standard in 1998.
- C++11: Marks the beginning of modern C++.
- C++14/17: Some slight fixes and improvements of C++11.
- C++20: The first standard that delivers on virtually all the features that Bjarne Stroustrup dreamed of in *The Design and Evolution of C++* in 1994.
    - "D&E Complete"

CS100 is based on C++17.

Overview of C++

What do **embedded systems**, **game development**, **high frequency trading**, and **particle accelerators** have in common? - C++, of course!

Effective C++ Item 1 (by Scott Meyers): **View C++ as a federation of languages.**

> The easiest way is to view C++ not as a single language but as a federation of related languages ... Fortunately, there are only four:
>
> - C.
> - Object-Oriented C++.
> - Template C++.
> - The STL.

"Better C"

Safer & more reasonable designs.

- `bool`, `true` and `false` are built-in. No need to `#include <stdbool.h>`. `true` and `false` are of type `bool`, not `int`.
- The return type of logical operators `&&`, `||`, `!` and comparison operators `<`, `<=`, `>`, `>=`, `==`, `!=` is `bool`, not `int`.
- The type of string literals `"hello"` is `const char [N+1]`, not `char [N+1]`.
- The type of character literals `'a'` is `char`, not `int`.

"Better C"

Safer & more reasonable designs.

- Potentially dangerous type conversions are not allowed to happen implicitly. **They are errors, not just warnings.**
- `const` variables initialized with literals are compile-time constants. They can be used as the length of arrays.
- `int fun()` declares a function accpeting no arguments. It is not accepting unknown arguments.

Basic IO

Hello world

C: Use `printf`

```c
#include <stdio.h>

int main(void) {
  printf("Hello world\n");
  return 0;
}
```

C++: Use `std::cout`

```cpp
#include <iostream>

int main() { // just an empty `()`
  std::cout << "Hello world\n";
  return 0;
}
```

Note on the `main` function: In C++, a function declared with an empty parameter list **accepts no arguments**, so there is no need to put a `void` there.

A+B problem

C:

```c
#include <stdio.h>

int main(void) {
  int a, b;
  scanf("%d%d", &a, &b);
  printf("a + b = %d\n", a + b);
  return 0;
}
```

C++:

```cpp
#include <iostream>

int main() {
  int a, b;
  std::cin >> a >> b;
  std::cout << "a + b = " << a + b
            << '\n';
  return 0;
}
```

- For input: **There is no need to take the address of `a` and `b`!** C++ has a way to obtain the *reference* of the argument.
- **There is no need to write `%d`!** C++ has a way of identifying the type of the argument, and will select the correct way to handle that type.

IOStream: Input and Output Stream

```cpp
#include <iostream>

int main() {
  int a, b;
  std::cin >> a >> b;
  std::cout << "a + b = " << a + b << '\n';
  return 0;
}
```

`std::cin` and `std::cout`: two **objects** defined in the header file `<iostream>`.

- They are not **functions**.
- The input and output "functions" are actually the operators `>>` and `<<`, which are **overloaded** to do something different from bit shifting.
  - We will learn about operator overloading in later lectures.

IOStream: Input and Output Stream

```cpp
#include <iostream>

int main() {
  int a, b;
  std::cin >> a >> b;
  std::cout << "a + b = " << a + b << '\n';
  return 0;
}
```

`std::cin` and `std::cout`: two **objects** defined in the header file `<iostream>`.

- `std::cin` stands for standard input stream. `std::cout` stands for the standard output stream.

IOStream: Input and Output Stream

`std::cin >> x`: Read something and stores it in the variable `x`.

- `x` can be of any supported type: integers, floating-points, characters, strings, ...
- **C++ has a way of identifying the type of `x` and selecting the correct way to read the value for `x`.** We don't need the annoying `"%d"`, `"%f"`, ... anymore.
- **C++ functions have a way of obtaining the *reference* of `x`.** We don't need to take the address of `x`.

IOStream: Input and Output Stream

- `std::cin >> x` **returns** `std::cin`, so we can read several inputs in a chained way:
  - `std::cin >> x >> y >> z` is equivalent to `((std::cin >> x) >> y) >> z`, which is equivalent to

```
std::cin >> x;
std::cin >> y;
std::cin >> z;
```

  - Similarly, outputs can also be chained: `std::cout << x << y << z` is equivalent to

```
std::cout << x;
std::cout << y;
std::cout << z;
```

Standard library

Standard library header file names

The names of C++ standard library header files **have no extensions**: `<iostream>` instead of `<iostream.h>`, `<string>` instead of `<string.h>`.

Namespace `std`

A **namespace** is a colleciton of names (of types, objects, functions, etc.).

C++ has a large standard library with a lot of names declared.

To avoid **name collision**, all the names from the standard library (such as `cin` and `cout`) are placed in a **namespace** named `std`.

- Example of name collision in C: Suppose we want to write our own quick sort:

```c
#include <stdlib.h> // include all the names from `stdlib` into the program.
void qsort(int *a, int n) { // Ooops! `stdlib` already has one named `qsort`.
  // ...
}
// ...
qsort(a, n); // which version of `qsort` is referred to?
```

- Use `std::qsort` to refer to `qsort` from `stdlib`, if it is in `std`.

Namespace `std`

A **namespace** is a colleciton of names (of types, objects, functions, etc.).

C++ has a large standard library with a lot of names declared.

To avoid **name collision**, all the names from the standard library (such as `cin` and `cout`) are placed in a **namespace** named `std`.

- Use `std::cin;` to refer to `cin` in `std`, where `::` is a scope resolution operator.
- Use `using std::cin;` to introduce `cin` in `std` into **the current scope**, so that `cin` can be used without `std::`.
- Use `using namespace std;` to introduce **all the names in `std`** into the current scope, but **you will be at the risk of name collision again.**

Compatibility with C standard library

The C++ standard library has everything from the C standard library.

- The C++ version of a C standard library header file `<xxx.h>` is `<cxxx>`, with all the names also introduced into `namespace std`.

```cpp
#include <cstdio>
int main() {
  int a, b; std::scanf("%d%d", &a, &b);
  std::printf("%d\n", a + b);
}
```

[Best practice] Use `<cxxx>` instead of `<xxx.h>` when you need the C standard library in C++.

`std::string`

Defined in the C++ standard library header file `<string>` (**not** `<string.h>`, **not** `<cstring>`)

Define and initialize a string

```
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
std::cout << s2 << std::endl; // aaaaaaa

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

Default-initialization of a `std::string` will produce **an empty string**, not indeterminate value and has no undefined behaviors!

Strings

- The memory of `std::string` is **allocated and deallocated automatically**.
- We can insert or erase characters in a `std::string`. **The memory of storage will be adjusted automatically.**
- `std::string` **does not need an explicit** `'\0'` **at the end**. It has its way of recognizing the end.
- When you use `std::string`, **pay attention to its contents** instead of the implementation details.

Length of a string

Member function `s.size()`

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

Not `strlen`, not `sizeof`!!

Member function `s.empty()`

```
if (str.empty()) {
  // ...
}
```

Concatenation of strings

Use `+` and `+=` directly!

- No need to care about the memory allocation.
- No awkward functions like `strcat`.

```
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"
```

Concatenation of strings

At least one operand of `+` should be `std::string`.

```
const char *old_bad_ugly_C_style_string = "hello";
std::string good_beautiful_Cpp_string = "hello";

std::string s1 = good_beautiful_Cpp_string + "aaaaa"; // OK.
std::string s2 = "aaaaa" + good_beautiful_Cpp_string; // OK.
std::string s3 = old_bad_ugly_C_style_string + "aaaaa"; // Error
```

Is this ok?

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

Concatenation of strings

At least one operand of `+` should be `std::string`.

```
const char *old_bad_ugly_C_style_string = "hello";
std::string good_beautiful_Cpp_string = "hello";

std::string s1 = good_beautiful_Cpp_string + "aaaaa"; // OK.
std::string s2 = "aaaaa" + good_beautiful_Cpp_string; // OK.
std::string s3 = old_bad_ugly_C_style_string + "aaaaa"; // Error
```

Is this ok?

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

Yes! `+` is **left-associated**. `(hello + "world")` is of type `std::string`.

Use `+=`

In C, `a = a + b` is equivalent to `a += b`. **This is not always true in C++.**

For two `std::string`s `s1` and `s2`, `s1 = s1 + s2` **is different from** `s1 += s2`.

- `s1 = s1 + s2` constructs a temporary object `s1 + s2` (so that the contents of `s1` are copied), and then assigns it to `s1`.
- `s1 += s2` appends `s2` directly to the end of `s1`, without copying `s1`.

Try these with `n = 1000000`:

```
std::string result;
for (int i = 0; i != n; ++i)
  result += 'a'; // Fast
```

```
std::string result;
for (int i = 0; i != n; ++i)
  result = result + 'a'; // Very slow
```

Lexicographical comparison of strings

Just use `<`, `<=`, `>`, `>=`, `==`, `!=`.

- No loops. No weird functions like `strcmp`.

Copying a string

Just use `=`.

```
std::string s1{"Hello"};
std::string s2{"world"};
s2 = s1; // s2 is a copy of s1
s1 += 'a'; // s2 is still "Hello"
```

String IO

Use `std::cin >> s` and `std::cout << s`, as simple as handling an integer.

- Does `std::cin >> s` ignore leading whitespaces? Does it read an entire line or just a sequence of non-whitespace characters? Do some experiments on it.

`std::getline(std::cin, s)`: Read a string starting from the current character, and stop at the first `'\n'`.

- Is the ending `'\n'` consumed? Is it stored? Do some experiments.

Traversing a string: Use range-based `for` loops.

Example: Print all the uppercase letters in a string.

```
for (char c : s) // The range-based for loops
  if (std::isupper(c)) // in <cctype>
    std::cout << c;
std::cout << std::endl;
```

Equivalent way: Use subscripts, which is verbose and inconvenient.

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

[Best practice] Use range-based `for` loops. They are modern, clear, simple.

⇒ More about range-based `for` loops in later lectures.

Conversion between strings and arithmetic numbers

For a number `x` of any arithmetic type, `std::to_string(x)` returns a string representing it.

```
int ival = 42;
double dval = 3.14;
std::string s = std::to_string(ival) + std::to_string(dval);
std::cout << s << '\n'; // output: 423.140000
```

`std::stoi(s)`, `std::stol(s)`, ...: Extract the arithmetic value represented by `s`.

See this list.

Summary

- C++ IO: Use `std::cin >> x >> y`, `std::cout << x << y`.
- C++ standard library header file names have no extensions.
- Namespace `std`.
- Can use the C standard library in C++, but use `<cxxx>` instead of `<xxx.h>`.

Summary

`std::string`

- No need for a terminating `'\0'`.
- Automatic memory management.
- `s.size()` returns the length. `s.empty()` returns whether `s` is empty.
- Use `+` and `+=` for concatenation. Use `<`, `<=`, `>`, `>=`, `==`, `!=` for comparison. Use `=` for copying.

Summary

`std::string`

- Use `std::cin` and `std::cout`, as well as `std::getline` for IO.
- Use `s[i]` to access the elements.
- Use range-based `for` loops to traverse a string.
- Use `std::to_string` and `std::stoi`, `std::stol`, ... for numeric conversions.
- Full list of functions related to `std::string`: https://en.cppreference.com/w/cpp/string/basic_string