VG101 FA2021 RC #5

What's different in C++?

- **OOP (Object-oriented programming)**
- Namespace and scope operator
- Template
- **STL: standard template library**

In next two weeks, we will mainly focus on OOP and STL parts.

Special Notes

C++ may be one of the most complicated popular programming languages. Although you have only taken two courses about C++, I decided to cover a lot (almost everything I think you will encounter in the Hw and Lab) in this RC, since this is possibly the final regular RC this semester. In addition, I am neither a good teacher nor a master of C++. So probably you may feel hard to follow this RC. Don't worry about this, some concepts also took me a long time to understand when I prepared this RC slide. Feel free to ask us if you have any questions. **Finally, do not forget to practice.**

Namespace

- We could declare some scopes with names to store some variables or functions in C++, this scope is called namespace. Identifiers in different namespaces will not interfere with each other.
- Syntax:

  ```
  namespace scope {
      // ...
  }
  ```

- Variables or functions declared in a certain scope can be accessed by `scope::name`
- `using` keyword can integrate a namespace into global namespace.
- Example: `using namespace std;`
- Although the syntax is similar, `class` is quite different from `namespace`!

  - A **class** is a data type. If you have a class named Foo, you can create objects of class Foo and use them in many ways.
  - A **namespace** is simply an abstract way of grouping identifiers together.
- In VG101, I believe you only need to write `class`.

Reference

- Pointers are powerful, but they can also be devil if you do not use them properly.
- We can use `Type &ref = var;` to refer to a variable (notice that `var` cannot be a constant, and we cannot initialize a reference later).

-
  ```
  int a = 5;
  int &b = a;
  b = 10;
  std::cout << a << ' ' << b << std::endl;
  // Output: 10 10

  const int a = 5;
  int &b = a; // error
  int &c;     // error
  ```
- Once a reference is created, we can use it as the original `var` .
- Reference can avoid copy operation when we use big data structures, so it is useful with `class`.

-
  ```
  // Pass by value
  void LOAD(Database db) { // Copy an entire Database every time, may be very huge
      db.name = "vg101";   // And this line does not make sense
  }

  // C++ style (pass by reference)
  void LOAD(Database &db) {
      db.name = "vg101";
  }
  void SAVE(const Database &db);

  // C style (pass by pointer)
  void LOAD(Database *db) {
      strcpy(db->name, "vg101");
  }
  ```

Standard I/O

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

cout << "Hello " << "World!" << endl; // output: Hello World!
int a, b;
char c;
cin >> a >> b >> c;          // Read from stdin, separated by white
spaces (' ', '\t', '\n')
cout << a << b << c << endl; // Write to stdout (string, char, int. All
can be output by cout)
cerr << "Error:" << a << b;  // Write to stderr (no buffer)
```

- `cin` and `cout` free you from the workload of specifying the format; it can automatically transfer the format for you. We will come back to explain how it realize such functions after `class` and overloading
    - Honestly speaking, sometimes I miss the format string...
    - `printf("%d %d %d %d %d %d %d\n", ...);` will result in a quite long and ugly line using `cout`
    - **Although sometimes a bit ugly, please always use `cout` and `cin` instead of `printf` and `scanf` in C++**
    - C++20 introduces `std::format`, which allows you to use the format string in C++!
        - But you cannot use this feature since we only support C++17
- `endl` is similar to `\n`, plus it flushes the buffer. More specifically, `std::cout << std::endl;` is equivalent to `std::cout << "\n" << std::flush;`
- There are some other functions and libraries (e.g. `getline`, `fstream`) to implement I/O in different situations.

Stream I/O

`stream` is a commonly used concept in C++ with some special property (e.g. `iostream`, `fstream`, `stringstream`). You can consider `stream` as a pipe, and you can write it with some content from one end, and read the content from the other end of the pipe.

- `iostream` takes the keyboard (or other input device) as the one end of pipe
- `fstream` takes the file as one end of pipe
- `sstream` takes the `string` as one end of pipe

Their general syntax is similar, except for some special methods,

```
istream >> a >> b >> c; // input stream
ostream << "This is the first line" << "\n"
    << a << "\n"
    << c << endl; // output stream

string str;
getline(istream, str); // read in the characters of each line until it
discovers a newline
```

- On Lecture 18, Page 11+, Jigang provides a sample usage of the `fstream`.
- The `draw()` function provided in Lab 7 is a sample usage of the output `stringstream`.

Procedural Programming vs. Object-Oriented Programming (OOP)

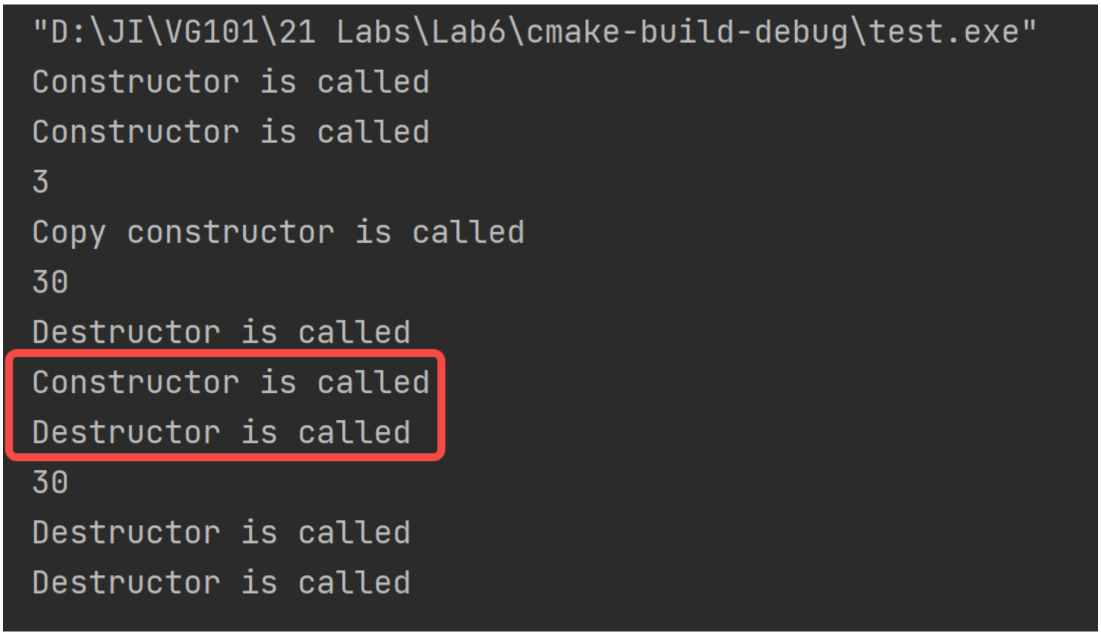
- Procedural Programming: Everything is considered as a procedure, which means a program is consists of several procedures (e.g. a function to implement some tasks). There is no ownership of data.
- Object-Oriented Programming: Everything is considered as an object. Data and methods all belong to some specific object (ownership of data).
- Example: Alice kisses Bob,
  - Procedural Programming: The core of this scenario is the procedure, "kiss". What happens is that the "kiss" procedure involves Alice and Bob this time. **Define function as `kiss(Alice, Bob)`.**
  - Object-Oriented Programming: The core of this scenario is the object, "Alice". Alice is the subject of the sentence, and "kiss" is just one of the methods that Alice can do. **Define function as `Alice.kiss(Bob)`.**
- OOP allow three major beneficial features,
  - Encapsulation (封装): Group data. Data are considered as the property of an object, and the object is responsible for maintaining the data. (e.g. Alice can use some `int` to store her `height`, `weight`, etc.)
  - Inheritance (继承): If an object `A` is an enhanced version of `a`, we can let `A` inherit from `a`. It promotes the code reuse.
  - Polymorphism (多态): If `A1`, `A2`, `A3` are all inherited from `a`, they work like `a` with different extensions. In a view from `a`, it is called polymorphism.
- An example from Lab 4.

Class

- `class` is a complex structure that is composed with multiple variables (called member variable or property) and functions (called member function or method).
- Compared with `struct` in C, the biggest difference is that `class` can contain functions.
- `struct` also appears in C++, and the only difference between `struct` and `class` in C++ is that `struct` is **public** by default, while `class` is **private** by default.
- However, for a good coding style, in C++, it's recommended:
  - use `struct` for plain-old-data structures without any class-like features
  - use `class` when you make use of features such as `private` or `protected` members, non-default constructors and operators, etc.
- A sample `Vector2` class from Lab 7 (without template),

```cpp
#include <iostream>

class Vector2 {
private:
    int x, y; // private member variables

public:
    Vector2(int _x = 0, int _y = 0) : x(_x), y(_y) { // constructor
using initializer lists, with default arguments (default arguments
should always be put at the end)
        std::cout << "Constructor is called" << std::endl;
    }

    Vector2(const Vector2 &another) { // copy constructor
        this->x = another.x;
        this->y = another.y;
        std::cout << "Copy constructor is called" << std::endl;
    }

    ~Vector2() { // destructor
        std::cout << "Destructor is called" << std::endl;
    }

    // Some function declartions
    void setX(int _x);

    int getX() const; // const member function

    // Various operator overloading
    Vector2 operator+(const Vector2 &another) const { return {x +
another.x, y + another.y}; }
```

```
    Vector2 operator-(const Vector2 &another) const { return {x - another.x, y - another.y}; }

    Vector2 operator*(int multiplier) const { return {x * multiplier, y * multiplier}; }

    Vector2 operator/(int multiplier) const { return {x / multiplier, y / multiplier}; }

    // Why we return a reference instead of value in the following functions?
    Vector2 &operator+=(const Vector2 &another) {
        x += another.x;
        y += another.y;
        return *this; // `this` is a pointer to the instance itself
    }

    Vector2 &operator-=(const Vector2 &another) {
        x -= another.x;
        y -= another.y;
        return *this;
    }

    Vector2 &operator*=(int multiplier) {
        x *= multiplier;
        y *= multiplier;
        return *this;
    }

    Vector2 &operator/=(int multiplier) {
        x /= multiplier;
        y /= multiplier;
        return *this;
    }

    bool operator==(const Vector2 &another) const { return x == another.x && y == another.y; }
};

// For function definitions outside the class, you need add Vector2::
void Vector2::setX(int _x) {
    this->x = _x; // `this` is a pointer!
}

int Vector2::getX() const {
    return x;
}

int main() {
    Vector2 tankA{1, 2}, tankB{3, 2}; // call the first constructor
    // tankA.x = 3; // error, access a private member
    tankA.setX(3); // How to call a class member function?
    std::cout << tankA.getX() << std::endl; // Output: 3
    if (tankA == tankB) {
        Vector2 tankC{tankB}; // call the copy constructor
        tankC += tankA;
        tankC *= 5;
        std::cout << tankC.getX() << std::endl; // Output: 30
        // tankC ends its life cycle, the destructor will be called
    }
    tankA = tankB * 10;
    std::cout << tankA.getX() << std::endl; // Output: 30
    // tankA and tankB end their life cycles, destructors will be called
}
```

- The real running output:
- 

```
"D:\JI\VG101\21 Labs\Lab6\cmake-build-debug\test.exe"
Constructor is called
Constructor is called
3
Copy constructor is called
30
Destructor is called
Constructor is called
Destructor is called
30
Destructor is called
Destructor is called
```

- We only declare three `Vector2` objects, but constructor and destructor are both called four times, why?
- `tankA = tankB * 10;` **Refer to the `operator*`, think about what happens in this line.**
- The concepts appear above:
    - **How to call a `class` member function?**
        - First declare a `class` object, e.g. `Vector2 tankA;`
        - Then call the member function through the object, e.g. `tankA.setX(3);`
    - Constructor: Called when declaring/initializing a new `class` object. If there is more than one constructors, the compiler will choose one according to the passing arguments.
    - Destructor: Called when the `class` object ends its life cycle.

  - `const` member function: Any member variables of the `class` cannot be changed in this function.
  - `private` members: Can only be accessed by the members of this `class` (or some `friend class`).
  - `public` members: Can be accessed by anyone.
  - `this`: A **pointer** to the instance itself.
  - Operator overloading: Customizes the C++ operators for operands of user-defined types.
- A sample usage of constructor and destructor:
-
```
class DynamicArray {
private:
    int *arr;
    size_t size;

public:
    DynamicArray(size_t _size) : size(_size) {
        arr = new int[_size];
    }

    ~DynamicArray() {
        delete[] arr;
    }
};
```

std::vector

STL vector realizes a dynamic array container so that we could use it as normal arrays, plus more functions like `insert()` and `push_back()` to add elements, and `erase()` to remove elements.

- Pros:
  - fast random access (e.g. `vec[101]`) $O(1)$
  - fast insert/delete at the back (`push_back`, `pop_back`) $O(1)$
- Cons:
  - inserting / deleting at other position is slow (`insert`, `erase`) $O(n)$

Believe me, at least in VG101, `vector` can meet all your expectations for an array. So please get familiar with `vector` and frequently use it in your homework/lab.

- How to use it?

-
  ```
  #include <vector>
  using namespace std;

  vector<int> vec1;     // holds int
  vector<Vector2> vec2; // holds Vector2
  vector<string> vec3;  // holds string
  ```
- Initialization:

-
  ```
  vector<T> v1;       // empty vector v1
  vector<T> v2(v1);   // copy constructor, v2 == v1
  vector<T> v3(n, t); // construct v3 that has n elements with value t
  ```
- Size:
  - `v.size()` returns a value of `size_type` corresponding to the vector type.
  - Example: `vector<int>::size_type`
    - a companion type of vector (to make the type machine-independent).
    - essentially an **unsigned** type, so it can be directly converted to `unsigned int` but not `int`.
    - `unsigned int s = v.size();`
  - Check whether the `vector` is empty: `v.empty()`.
- Add/Remove:
  - `vec.push_back(t)`: add element `t` to the end of `vec`.
  - Elements are copies: no relationship between the element in the container and the value from which it was copied.
  - `vec.pop_back()`: remove the last element in `vec`. `vec` must be non-empty.
- Other useful operations:

-
  ```
  v1 = v2;    // copy assignment
  v.clear(); // clear all elements, size = 0
  v.front(); // The first element of v, must be non-empty
  v.back();  // The last element of v, must be non-empty
  ```

Iterator

All STL containers define iterator types:

- Declaration: `vector<int>::iterator it;`
  - `` `v.begin() `` returns an iterator pointing to the first element of vector
  - `v.end()` returns an iterator positioning to **one-past-the-end** of the vector
    - usually used to indicate when we have processed all the elements in the vector
  - If the vector is empty, then `v.begin() == v.end()`

- Operations:
  - Dereference: can read/write through `*iter` (cannot dereference the iterator `v.end()`)
  - `++iter, iter++`: next iterator (cannot increment the iterator `v.end()`)
  - `--iter, iter--`: go back to the previous iterator
  - `iter == iter1` and `iter != iter1`: check whether two iterators point to the same data item

```
vector<int>::iterator begin = ivec.begin();
auto end = ivec.end(); // Thanks to C++11.
while (begin != end) {
  cout << *begin++ << " ";
  // 1. get the value of *begin
  // 2. cout << *begin << " ";
  // 3. begin++;
}
```

- Iterator Arithmetic
  - `iter + n, iter - n`, where `n` is an integer

```
// Example 1: Go to the middle
auto mid = v.begin() + v.size()/2;

// Example 2: Random access through iterator
auto begin = v.begin();
cout << *(begin + 7) << endl;
cout << v[7] << endl; // Same
```

- Relational Operation: `>, >=, <, <=, ==, !=`
  - To compare, iterators must refer to elements **in the same container**.

```
// Example: Traverse a vector through iterator
for (auto it = v.begin(); it != v.end(); ++it) {
    cout << *it << endl;
}

// C++11 style: for-range based loop
for (auto &item : v) {
    cout << item << endl;
}
```

- More about initialization of `vector`
  - `vector<T> v(b,e)`: create vector `v` with a copy of the elements from the range denoted by iterators `b` and `e`.

```
vector<int> v1(10, 5);
vector<int> v2(v1);
vector<int> v3(v1.begin(), v1.end());
vector<int> v4(5, 5);
vector<int> v5(v1.begin(), v1.begin() + v1.size()/2);
// v1, v2, v3 are the same
// v4, v5 are the same
```

  - You can even use array to initialize vector:

    ```
    int a[] = {1, 2, 3, 4};
    unsigned int sz = sizeof(a) / sizeof(int);
    vector<int> vi(a, a + sz); // pointer
    ```

- More about add/remove:
  - `v.insert(it, t)`, `it` **is an iterator**
    - Insert an element with value `t` **right before** the element referred to by iterator `it`.
    - Return an iterator referring to the element that was added.
  - `v.erase(it)`, `it` **is an iterator**
    - Remove the element that iterator `it` refers to.
    - Return an iterator referring to the element after the deleted one, or `v.end()` if `it` refers to the last element.

`std::string`

Besides `std::vector`, C++ also provides an useful string library `<string>`. Also, at least in VG101, I think you can always use `string` rather than C-style string `char[]`.

- Initialization:

-
  ```
  #include <string>
  using namespace std;

  string str1 = “blablabla”; // Overload assignment operator
  string str2(“blablabla”);  // Copy constructor
  ```
- The `string` can automatically store infinite numbers of characters without worrying about memory leak. `cin` and `cout` can also take string as parameters.
- Other useful and straight-forward operations (No longer anti-human like `strcmp`!)
  - Assignment:
    - C++ `string`: `str1 = str2;`
    - C-style string: `strcpy(str1, str2);`

    - Concatenate:
        - C++ `string`: `str3 = str1 + str2;`
        - C-style string: `strcpy(str3, str1); strcat(str3, str2);`
    - Compare:
        - C++ `string`: `str1 == str2`, `str1 > str2`, ...
        - C-style string: `strcmp(str1, str2) == 0`, ...
    - Get length:
        - C++ `string`: `str.length();` or `str.size();` they are the same
        - C-style string: `strlen(str);`
        - This is also an example of the OOP style.
    - Convert to a C-style string (so it’s compatible with C library): `str.c_str();`
- Actually, `string` is also a STL container like `vector`, so it has the iterator, and some methods similar to the `vector`
    - `opeartor[]`: access string as a char array, e.g. `str[10]`
    - Check whether the `string` is empty: `str.empty()`.
    - Methods quite similar to `vector`: `str.front()`, `str.back()`,`str.push_back`, `str.pop_back()`, ...
    - iterator-based: `str.insert()`, `str.erase()`, ...
    - Special and useful methods: `str.append()`, `str.substr()`, `str.find()`, ...
- **There are a lot of methods, you can check them in https://en.cppreference.com/w/cpp/string/basic_string.**
- Useful non-member functions:
    - `stoi()`: convert string to number
    - `getline()`: taking a `istream` and a `string` as argument, read a line from `istream` and store it into `string`
- The best way to get familiar with them: 1. Read the documentation, 2. Try it by yourself!

Reference

1. RC-week12-Checklist.md, Ye Chenhao, VG101 FA2018 TA
2. RC-week13-Checklist.md, Ye Chenhao, VG101 FA2018 TA
3. VG101 Jigang RC10.pptx, Wang Kaibin, VG101 FA2019/2020 TA
4. VG101 Jigang RC 8 - C++.pdf, Wang Kaibin, VG101 FA2019/2020 TA
5. final_notes_21_26.md, Ma ZiQiao, VE280 SU2020 TA
6. Many kind programmers on Stack Overflow