Final Review Notes

VG101-SU20 TA Group

Last modified: 20-08-04

C++ Basics

C vs C++

Similarity with C

Everything from C is valid:

- conditional statement (`if, else, for, while`)
- function declaration
- logical operators
- ...

New features

- New data type: `bool` (in c you need to define `bool` or `#include <stdbool.h>`)
- namespace
  - `using namespace std;`
- New libraries
  - iostream: for cin, cout
  - fstream: for file io
  - sstream: to parse strings
  - string: deal with real string instead of c-style char array
  - various containers
- Default arguments for function
- Function and operator overloading
- class, template

Compilation

- The following `g++` command will compile three c++ source files together, and name the output binary file as `main`:
  - `$ g++ --std=c++11 -pedantic -g -Wall -Werror ex1.cpp stats.cpp main.cpp -o main`
  - `g++` is the is the C++ compiler we are invoking.
  - `--std=c++17`: will tell compiler to compile according to C++ 17 standards.
  - `-pedantic`:tells the compiler to adhere strictly to the C++ standard. Without this flag, compilers often provide extensions or allow behavior that is not permitted by the standard.
  - `-Wall` argument asks the compiler to generate warnings about possible programming errors.
  - `-Werror` argument configures the compiler to treat warnings as errors, so that it does not compile code that has warnings.
  - `-o main.exe` tell the compiler to produce the output file main.exe.
- Notes for clion users:
  - You can set some of the `CXX_MAKE_FLAGS` in CMakelists if you are a clion a user. **Setting up them earlier in exams can help you save time, otherwise you may encounter the case that your code can wrong on JOJ computer but compile error on JOJ**)

I/O streams

> Streams: sequence of data that can be accessed sequentially.

Extraction operator **>>**: used to remove/get values from the input stream

Insertion operator **<<**: used to put values into the output stream

Stream in c++ is **uni-directional**: If you want to read and write data to the same file or device, you need two streams.

Standard streams

```
1 #include <iostream>
```

cin: an input stream binded with the standard input (something typed from the keyboard, etc)

cout: an output stream binded with standard output

```
// assume that a is of type int
cin >> a; // extract data from standard input stream to a cin
cout << 42; // Insert the number into the output stream.
```

formatting output

header

```
#include <iomanip>
```

useful functions

- `setw(width)`
- `setprecision(2)`
- `setfill(z)`

File streams

header

```
#include <fstream>
```

Declare input/output file streams

```
ifstream input(filename);
// connect input file stream `input` to a file `filename`
ofstream output(filename);
```

Read contents from ifstream

```
int a;
input >> a; // extract an int from the ifstream
string str;
getline(input,str); // obtain a whole line

```

Insert contents into the ofstream

```
output << contents;
```

Other functions

```
input.get(); // get a char from the ifstream
input.seekg(0);
/* move the position back to
the beginning of the file, like `rewind`
*/
```

Reset the state of the fstream: some operation may make the file stream into bad or eof state, which may preventing the normal operations like `seekg`, `tellg`, `getline`, etc.

```
if(input.fail())
    input.clear(); // reset the bad/error state flags
```

A comprehensive demo code is provided in the folder.

String Stream

`Stringstream` is a powerful tool that can make your life easier. More in the `string` section.

string

header

```
#include <string>
```

Note: sometimes you may compile your program without including the header `<string>`, however, devastating consequences may happen to your program.

basic properties

- by default, a string is initialized as `""` (empty)
- operators like `+`, `+=` are supported
- subscripting supported, i.e., you may visit the corresponding character using `[i]`, where i is the index
- get the size with `str.length()`

input/output

```
string str;
while(cin >> str) cout << str;

string line;
while(getline(cin, line)) cout << line;
```

string stream

header

```
#include <sstream>
```

Declare input/output string streams

```
istringstream iStream; // declare an input string stream
iStream.str(s); // initialize iStream with string `s`
ostringstream oStream;
```

Example:

```
istringstream iStream;
string s = "love vg101";
iStream.str(s);
string word1, word2;
iStream >> word1 >> word2;
// word1 is "love", word2 is "vg101"
cout << word1 << word2 << endl;
```

Note: you may also take advantage of string stream to convert between string and other data types.

A comprehensive demo code is provided in the folder.

keyword

many new keywords are preserved / introduced in c++:

- `const`
- `static`
- `virtual`
- ...

const

basics

- only types with values may be declared as `const`.
- array can hold const values, but cannot be declared as const
- high level function can not be declared as const
  - member function in class can be declared as const
- reference cannot be declared as const
  - can declare reference to const object

conversion

General rule: permit that const object cannot be modified

```
int x = 3;
int *ptr1 = &x; // OK -- does not permit const object to be modified

const int *ptr2 = ptr1; // OK -- does not permit const object to be modified

ptr1 = ptr2; // ERROR -- compiler sees that ptr2 is pointing to a const
object, but ptr1 is not a pointer to const
```

const pointers and pointers to const (optional, more in VE280)

```
int x = 3;

int *ptr1 = &x;
int * const ptr2 = &x; // const pointer pointing to a non-const int
const int *ptr3 = &x; // non-const pointer pointing to a const int
const int * const ptr4 = &x; // const pointer pointing to a const int
```

Dynamic Array Allocation

Similar to malloc/calloc/free, but much more simplier.

- Single object:

```
int* a = new int;
delete a;
```

- Array:

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

- 2D Array:

```
int** A=new int*[n];
for(int i=0;i<n;++i)
    A[i]=new int[n];

for(int i=0;i<n;++i)
    delete[] A[i];
delete[] A;
```

Function default argument

Default argument: default value provided for a function parameter

```
int add(int x=5, int y=6);

int main()
{
    // case 1
    add(); // perform 5+6, return value is 11
    // case 2
    add(1); // perform 1+6
    // case 3: normal case
    add(1,2); //perform 1+2, return value is 3
    return 0;
}

int add(int x, int y)
{
    return x+y;
}
```

Function Overloading

Define two different functions with exactly the same name, but different argument count and/or argument types. Example:

```cpp
int add(int x, int y);  // func1
int add(int x, int y, int z);  // func2
double add(double x, double y);  //func3

add(2, 3); // it will call func1
add(1, 2, 3); // it will call func2
add(1.0, 2.5); // it will call func3
```

Compiler tells which function to call based on the actual argument count and types.

If you would like to dig a little deeper, you may refer to function signature in C++. The next part is optional:

Factors that determine the function signature:

- function name
- type of the parameter
- (if the function is a class member) **const** / volatile qualifier
- (for function template) the type of its template arguments & the **return type**
    - Note: the c++ standard does forbid a function declaration that only differs in the return type.

Class

struct vs. class

`struct` is reserved in c++; however, c++ provides us with a more powerful tool to manage data: `class`. Their major differences include (but not limit to):

Category	struct	class
security	Structure members can be accessed from anywhere	class has access control, so that it can hide its implementation details
function	c structures do not permit functions inside structure	class permits member function
instance declaration	the keyword `struct` is required	the keyword `class` is not required
special features	-	constructor, destructor, operator overloading, template, ...
OOP	-	Polymorphism, Inheritance

A c-style triangle structure:

```c
typedef struct _Triangle{
    double a;
    double b;
    double c;
} Triangle;

void triangleScale(Triangle *tri, double s) {
    tri->a *= s;
    tri->b *= s;
    tri->c *= s;
}
// ...
```

You may compare this definition with the Triangle class in the next section.

Components

components in class

- Attributes
  - data
  - struct
- Methods
  - constructor / destructor
  - member function
  - operator overloading
- Instance
  - a realization of a class

Note: the compiler will generate a default constructor/destructor for your class. However, if you want to manage dynamic memory / operator on files /..., you need to define your own constructor/destructor.

access control

Access	Same Class	Derived Class	Others
private	√	×	×
protected	√	√	×
public	√	√	√

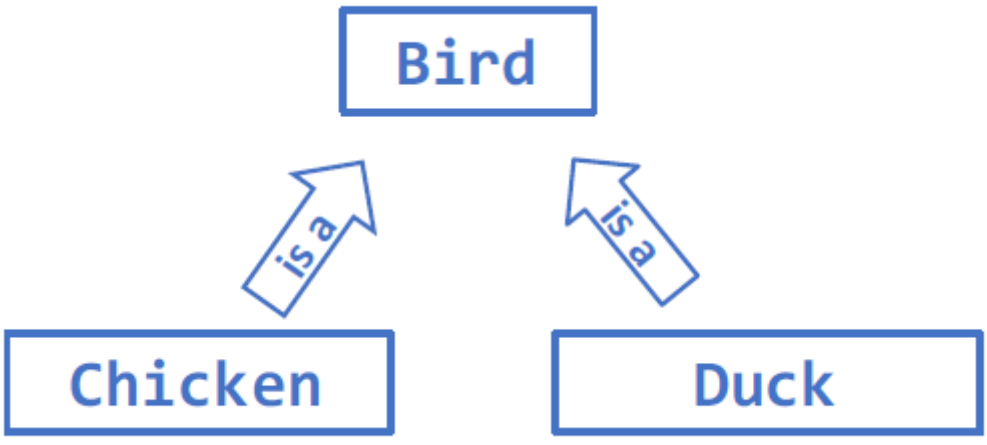
demo

```c
class Triangle {
 private:
    /* Private Attributes */
    double a;
    double b;
    double c;

    /* Private Methods */
    void printEdge(){
        cout << this->a << " " << this->b << " " << this->c << endl;
```

```
    }
 public:
    /* constructor */
    Triangle(double a_in, double b_in, double c_in){
        a=a_in; b=b_in; c=c_in;
    }
    /* Public Methods */

    double perimeter() const {
        return this->a + this->b + this->c;
    }
    void scale(double s) {
        cout << "original edges: ";
        printEdge();
        this->a *= s;
        this->b *= s;
        this->c *= s;
        cout << "new edges: ";
        printEdge();
    }
    Triangle operator+ (const Triangle him){ // operator overloading
        this->a += him.a;
        this->b += him.b;
        this->c += him.c;
        return *this;
    }
};
// ...
int main(){
    Triangle a(1,1,1), b(2,2,2); // two instances of class Triangle
    cout << a.perimeter() << endl;
    a = a + b; // operator overloading
    cout << a.perimeter() << endl;
    a.scale(0.5);
    // a.printEdge(); // Wrong: member function printEdge is private
    return 0;
}
```

Inheritance

> Inheritance is the ability for a class to reuse the interface or functionality of another class.



Inheritance field type:

- public
- protected
- private

member in Base Class	public Inheritance	protected Inheritance	private Inheritance
public	public	protected	private
protected	protected	protected	private
private	×	×	×

Notes:

- Private members are never inherited
- the visibility of a member in a derived class is never higher than the inheritance type

Polymorphism

> Polymorphism is providing a single interface to entities of different types. -- Bjarne Stroustrup

We use inheritance so that an instance of a derived class is automatically an instance of the base class.

demo

```
class Bird {
 private:
    int age;
    string name;

 public:
    Bird(const string &nameIn){
        age = 0;
        name = nameIn;
    }
    string getName() const {
        return name;
    }
    int getAge() const {
        return age;
    }
    void haveBirthday() {
        ++age;
    }
    virtual void talk() const {
        cout << "tweet" << endl; // we use a virtual keyword to override member functions even there is no compile-time information about the actual type of the class
    }
};

class Chicken : public Bird {
 private:
    int roadsCrossed;
```

```
 public:
 // since name is a private method of the bird, we use a member-initializer list to implement the constructor
    Chicken(const string &nameIn) : Bird(nameIn), roadsCrossed(0) { }
    void crossRoad() {
        cout << "Chicken " << this->getName() << " crosses a road." << endl;
    }
    void talk() const {
        cout << "chee" << endl;
    }
};

class Duck : public Bird {
 private:
    int numDucklings;
 public:
    Duck(const string &nameIn) : Bird(nameIn), numDucklings(0) { }
    void haveBabies() {
        numDucklings += 7;
    }
    void talk() const {
        cout << "quack" << endl;
    }
};
```

Notes for `virtual`:

- We define a virtual function in the base class and use base class pointers to access the function, so that all the grouped types share the interface.
- The virtual property of a member function is inherited.
- `virtual` keyword cannot be applied to the constructor. Indeed, when creating an instance of a derived class, the base class constructor will also be called (right before the derived class constructor is called).
- When we use a pointer of class A to invoke a non-virtual function foo, A::foo would always be invoked.
- When we use a pointer of class A to invoke a virtual function bar, which function is invoked depends on the type of the object the pointer is pointing to.

```
// we use the class defined in the previous section
Bird b("1");
b.talk(); // tweet
Chicken c("2");
c.talk(); // chee
Duck d("3");
d.talk(); // quack

Bird *ptrB = &b;
Bird *ptrC = &c;
Bird *ptrD = &d;
ptrB->talk();
ptrC->talk();
ptrD->talk();
// if the virtual keyword is included, it will print tweet chee quack
// if not, it will print tweet tweet tweet
```

Notes for constructor:

- In `Triangle`, we just assign the member data with corresponding value; in `Bird` and its inherited classes, we use a member-initializer list to implement the constructor. These are two different ways of implementing a constructor, and the second way is recommended.
- For the call of constructor/destructor with inheritance, see the demo code below:

```
class A{
 public:
    int data;
    A(){cout << "con.A\n";}
    A(int d){data=d;cout<<"con.Data.A\n";}
    virtual ~A(){cout << "des.A\n";}
};

class B : public A{
 public:
    int data2;
    // B(int d1, int d2) : A (d1) {data2=d2;cout << "con.Data.B\n";}
    B(int d2){data2=d2;cout << "con.Data.B\n";}
    B() { cout << "con.B\n"; }
    ~B(){ cout << "des.B\n"; }
};
// ...
B b(5);
```

Operator overloading

> Operator overloading: customizes the C++ operators for operands of user-defined types.

We redefine certain operators to apply to new data types.

Most of the operators can be overloaded, but not all. We list a few common operators here:

```
A& A::operator= (const A& rhs);
A operator+ (const A& lhs, const A& rhs);
istream& operator>> (istream& is, A& rhs);
ostream& operator<< (ostream& os, const A& rhs);
bool operator== (const A& lhs, const A& rhs);
```

demo

```
class Complex{
private:
    double real,comp;
public:
    Complex(double r, double c):real(r),comp(c) {};

    Complex operator+(Complex oprand2)
    {
        // return *this + oprand2
        Complex result(0,0);
        result.real=this->real+oprand2.real;
        // *this is oprand1
        result.comp=this->comp+oprand2.comp;
        return result;
    }
// .....
```

Template

> A template is a model for producing code. We write a generic version, parameterized by one or more template parameters. The compiler then instantiates a specific version of the code by substituting arguments for the parameters and compiling the resulting code. We specify a template and its parameters by placing a template header before the entity that we are defining.

Template function

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

template <class T>
void outputArray (const T array, int count){
    for (int i = 0; i < count; i++)
        cout<< array[i]<< "\t";
    cout<<endl;
}
int main(){
    int a[2] = {9, 4};
    outputArray(a, 2);
    char b[2] = {'a', 'z'};
    outputArray(b, 2);
    return 0;
}
```

In this way, we define a templated function that can be applied to any data type.

Template class

```
template <class T>
class List{
 public:
    bool isEmpty();
    void insert(T v); // insert an element of type T in the back
    T remove(); // remove the last element and return it
    // ...
}
```

A glimpse at STL

STL stands for Standard Template Library, which includes a lot of useful containers, iterators, and algorithms. In VG101, you're only required to get familiar with a simple STL container `vector`, and some remaining stuff will be introduced in VE280, VE281.

std::vector

initialization

```
vector<T> v1; // empty vector v1
vector<T> v2(v1); // copy constructor
vector<T> v3(n,t); // construct v3 that has n elements with value t

vector<int> front(v.begin(), mid); // init with another vector

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

size `vector<int>::size_type sz = v.size()`

subscripting `v[i]`

```
vector<int>::size_type n;
v[n] = 0; //must ensure that the v.size() > n
for (n = 0; n != v.size(); ++n) {
    v[n] = 0;
}
```

iterator

iterator is a generalization of subscripts (i.e., index). It saves you from managing the index or data address by yourself.

Declaration: `vector<int>::iterator itr;`

Usage:

```
// assume vec is of type vector<int>
vector<int>::iterator begin = vec.begin();
vector<int>::iterator end = vec.end();
while (begin != end) {
    cout << *begin++ << " ";
    // 1. get the value of *begin
    // 2. cout << *begin << " ";
    // 3. begin++;
}

// assume vec is of type vector<int>
int sum = 0;
vector<int>::iterator itr;
for(itr=vec.begin(); itr!=vec.end(); ++itr){
    sum += *itr; // retrieving its value at position itr
}
```

supported operators on the vector iterator:

- `*` `++` `--` `==` `!=`
- `+`, `-`
- `>`,`>=`, `<`, `<=`

with iterator, we can introduce the fourth way of initilizing the vector:

```
vector<T> v4(begin,end); // create vector v4 with a copy of the elements
from the range denoted by iterators begin and end.
```

Notes:

- end() represents the position after the last element in the vector
- special operation supported by vector iterator:
  - `iter+n`, `iter-n`
  - `>`,`>=`, `<`, `<=`

add/remove

```
v.push_back(t); // copy the element t and store it at the end of the vector
```

```
v.pop_back(); // remove the last element in the vector
```

insert/erase

```
v.insert(p, t) // insert element t at position p, where p is an iterator
```

- inserts element with value `t` right before the element referred to by iterator `p`.
- returns an **iterator** referring to the element that was added

```
v.erase(p)
```

- removes element refered to by iterator p.
- returns an iterator referring to the elemtn after the one deleted, or an off-the-end iterator if p referred to the last element.

Useful Tools

> Congratulation! You've already digested every piece of knowledge required in Vg101 for c++. Here we provide you with some convenient tools for debugging.

I/O Redirection

- `<` **input** redirect
- `>` **output** redirect *More in Lab11 Manual.*

diff

> a command line tool that compares two text files.

Basic usage: `diff FILE1 FILE2`

- `c` - change
- `a` - addition
- `d` - deletion
- `n1 c n2` -- line n1 in FILE1 should change into line n2 in FILE 2
- `<` delete from FILE1
- `---` separation
- `>` added in FILE2

valgrind

> a powerful tool that checks whether your program has memory-related issues Basic usage: `valgrind --leak-check=full ./ex1`

Note: not available in windows