Table of Contents

Introduction

1 Header Files

1.1 The #define guard

All header files should have #define guards to prevent multiple inclusion. The format of the symbol name should be

```
__<HEADERNAME>_H__
```

To guarantee uniqueness, they should be based on the full name in a header file. For example, the file Seed.h in project VAT should have the following guard.

```
#ifndef __SEED_H__
#define __SEED_H__

#endif // __SEED_H__
```

1.2 Function Parameter Ordering

When defining a function, parameter order is: inputs, then outputs.

1.3 Names Order of Includes

Use standard order for readability and to avoid hidden dependencies: C library, C++ library, other libraries' .h, your project's .h. For example,

```
#include <sys/types.h>
#include <unistd.h>
#include <hash_map>
#include <vector>
#include "ben/project.h"
#include "zhong/czproject.h"
#include "myproject.h"
```

Reference

Google C++ Style Guide

Class

If a class has ten or more public methods, excluding constructor, destructor, access methods, and private/protected methods, we should split the class into smaller classes.

For example, in the Graph class, methods, like getNode(), getSize(), isEmpty() etc. should not be counted as those 10 public methods; while DFS(), removeNode() etc. should be counted.

For the Graph class, we only have two public methods, DFS() and removeNode(), although there are more than ten methods.

```
class Graph
{
   public:
   Graph(){};
   Graph(..,..){};
   Graph(..,..,..){};
   Graph(..,..,..,..){};
   DFS();
   removeNode();
   private:
   int getNode();
   int getSize();
   bool isEmpty();
   int max_vertex_
   int num_vertex_
   int num_edge_;
   char* vertex;
   int** edge;
};
```

Nested Classes

A class can define another class within it; this is also called a member class.

Normally, the number of nested classes cannot exceed two.

```
class Test
{
private:
// MyTest is a member class, nested within Test.
    class MyTest
    {
    ...
    };
};
```

Interface

A class is made abstract by declaring at least one of its functions as pure virtual function. A pure virtual function is specified by placing "= 0" in its declaration as follows:

```
class Box {
   public:
      // pure virtual function
      virtual double getVolume() = 0;
```

```
   private:
      double length;      // Length of a box
      double breadth;     // Breadth of a box
      double height;      // Height of a box
};
```

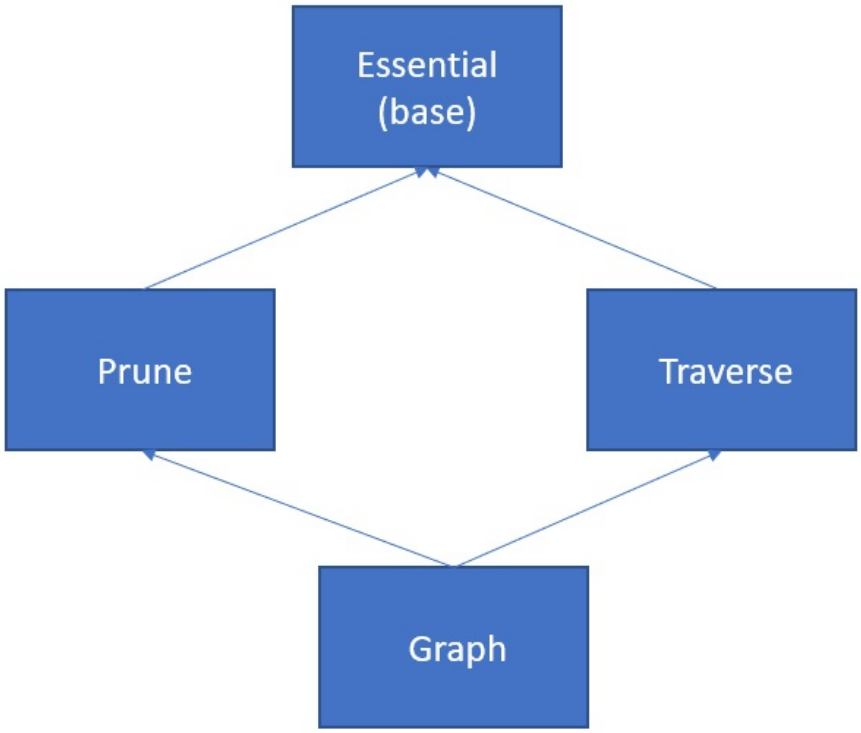
Abstract Class Example

Consider the following example where the parent class provides an interface to the base class to implement a function called getArea() −

```
#include <iostream>

using namespace std;

// Base class
class Shape {
   public:
      // pure virtual function providing interface framework.
      virtual int getArea() = 0;
      void setWidth(int w) {
         width = w;
      }

      void setHeight(int h) {
         height = h;
      }

   protected:
      int width;
      int height;
};

// Derived classes
class Rectangle: public Shape {
   public:
      int getArea() {
         return (width * height);
      }
};

class Triangle: public Shape {
   public:
      int getArea() {
         return (width * height)/2;
      }
};

int main(void)
{
   Rectangle Rect;
   Triangle  Tri;

   Rect.setWidth(5);
   Rect.setHeight(7);

   // Print the area of the object.
   cout << "Total Rectangle area: " << Rect.getArea() << endl;

   Tri.setWidth(5);
   Tri.setHeight(7);

   // Print the area of the object.
   cout << "Total Triangle area: " << Tri.getArea() << endl;
```

```
   return 0;
}
```

Other Design




```
class Essential {
   public:
     double getVolume() = 0;

   private:
      double length;      // Length of a box
      double breadth;     // Breadth of a box
      double height;      // Height of a box
};

class Prune : public Essential{
   public:
     FunctionA();

   private:
      double l;

};

class Traverse : public Essential{
   public:
      FunctionB();

   private:
      double h;
};
```

```
class Graph {
   public:
      Prune getPrune()
      {
         return *p;
      }

      Traverse getTraverse()
      {
         return *t;
      }

   private:
      Prune *p;
      Traverse *t;
};
```

Reference

Interfaces in C++ (Abstract Classes)

Google C++ Style Guide

Naming

General Naming Rules

Function names, variable names, and filenames should be descriptive; eschew abbreviation.

Types and variables should be nouns, while functions should be "command" verbs.

For example:

```
int num_errors; // Good.
int num_completed_connections; // Good.
```

File Names

Filenames should start with a capital letter and have a capital letter for each new word.

Examples of acceptable file names:

```
MyUsefulClass.cpp
MyClass.cpp
MyUsefulClass.h
MyClass.h
```

C++ files should end in .cpp and header files should end in .h.

Do not use filenames that already exist in /usr/include, such as db.h.

Type Names

Type names start with a capital letter and have a capital letter for each new word, with no underscores: MyClass, MyEnum.

```
// classes and structs
class MyTable { ...
class MyClass { ...
struct MyTestClass { ...
// typedefs
typedef hash_map<UrlTableProperties *, string> MyHash;
typedef long long int DataType_AL;
// enums
enum MyEnums { ...
```

Variable Names

Common Variable names

```
string table_name; // OK uses underscore.
string tablename; // OK all lowercase.
```

Class and Struct Data Members

```
string table_name_; // OK underscore at end.
string tablename_; // OK.
```

Function Names

Functions should start with a lowercase letter and have a capital letter for each new word. No underscores.

```
myFunction(MyTable& stu_name)
addTableEntry(int& stu_name)
deleteUrl(MyClass& stu_name)
```

Reference

Google C++ Style Guide

Scoping

Namespaces

Unnamed namespaces in .cpp files are encouraged. With named namespaces, choose the name based on the project, and possibly its path.

Unnamed Namespaces

Unnamed namespaces are allowed and even encouraged in .cpp files, to avoid runtime naming conflicts:

```
myspace
{
    // This is in TestFile.cpp file.
    // The content of a namespace is not indented
    enum { kUnused, kEOF, kError }; // Commonly used tokens.
    bool atEof() { return pos_ == kEOF; } // Uses our namespace's EOF.
} // myspace
```

Do not use unnamed namespaces in .h files.

Named namespaces should be used as follows:

Namespaces wrap the entire source file after includes, gflags definitions/declarations, and forward declarations of classes from other namespaces:

```
// In the .h file
namespace mynamespace
{
// All declarations are within the namespace scope.
// Notice the lack of indentation.
    class MyClass
    {
    public:
    ...
        void addTest();
    };
} // namespace mynamespace
```

```
// In the .cpp file
namespace mynamespace {
// Definition of functions is within scope of the namespace.
void MyClass::addTest()
{
...
}
} // namespace mynamespace
```

Do not declare anything in namespace std, not even forward declarations of standard library classes. Declaring entities in namespace std is undefined behavior, i.e., not portable.

To declare entities from the standard library, include the appropriate header file. You may not use a usingdirective to make all names from a namespace available.

```
// Forbidden This pollutes the namespace.
```

```
using namespace foo;
```

Reference

Google C++ Style Guide

Other Features

Prefer consts, enums, and inlines to #define

If you define the aspect_ratio as follows:

```
#define aspect_ratio 1.653
```

We recommend that you use the following code to replace it:

```
const double aspect_ratio = 1.653
```

When you need a constant value during the compilation phase, const is powerless. At this time you need enum.

For example, if you define an array:

```
int scores[numTurns];
```

The compiler insists on knowing the size of numTurns. You can use enum at this time:

```
enum {numTurns = 5};
```

Using #define to implement a macro, the function is the same as a function, but there is no overhead when the function is called, but the macro definition check causes unnecessary errors. The classic one is the comparison of two numbers:

```
#define CALL_WITH_MAX(a,b)  f((a)>(b)?(a):(b))
```

Even if you add parentheses to each variable, there is still an error, that is, when you pass in ++ a, b, when a> b, a plus two times will happen, and you all understand it. You can use the following statement instead:

```
template <typename T>
inline void callWithMax(const T& a,const T& b)
{
    f(a>b ? a : b);
}
```

Use const whatever possible

If you define the myVariable as follows:

```
int myVariable = 0;
myVariable = ComputeFactor(params...);
```

We recommend that you use the following code to replace it:

```
const int myVariable = ComputeFactor(params...);
```

If you define the str as follows:

```
string str;
```

```
str = "Hello World";
```

We recommend that you use the following code to replace it:

```
const string str = "Hello World";
```

Casting

Use C++ casts like static_cast<>(). Do not use other cast formats like int y = (int)x; or int y = int(x);. The problem with C casts is the ambiguity of the operation; sometimes you are doing a conversion and sometimes you are doing a cast (e.g., (int)"hello"); C++ casts avoid this. Additionally C++ casts are more visible when searching for them.

Preincrement and Predecrement

Use prefix form (++i) of the increment and decrement operators with iterators and other template objects. When the return value is ignored, the "pre" form (++i) is never less efficient than the "post" form (i++), and is often more efficient. This is because postincrement (or decrement) requires a copy of i to be made, which is the value of the expression. If i is an iterator or other nonscalar type, copying i could be expensive. Since the two types of increment behave the same when the value is ignored, why not just always preincrement?

Integer Types

Of the builtin C++ integer types, the only one used is int. If a program needs a variable of a different size, use a precisewidth integer type from , such as int16_t.

The sizes of integral types in C++ can vary based on compiler and architecture.

0 and NULL

Use 0 for integers, 0.0 for reals, NULL for pointers, and '\0' for chars.

For pointers (address values), there is a choice between 0 and NULL. Bjarne Stroustrup prefers an unadorned 0. We prefer NULL because it looks like a pointer. In fact, some C++ compilers, such as gcc 4.1.0, provide special definitions of NULL which enable them to give useful warnings, particularly in situations where sizeof(NULL) is not equal to sizeof(0). Use '\0' for chars. This is the correct type and also makes code more readable.

C++11

Use only approved libraries and language extensions from C++11 (formerly known as C++0x). Currently, none are approved.

C++11 has become the official standard, and eventually will be supported by most C++ compilers. It standardizes some common C++ extensions that we use already, allows shorthands for some operations, and has some performance and safety improvements.

Reference

Google C++ Style Guide