C++ Q&A

Yangang Cao

June 25, 2019

Contents

1 Basic Type

Type	Size in Xcode
bool	1
char	1
wchar_t	4
char16_t	2
char32_t	4
short	2
int	4
long	8
long long	8
float	4
double	8
long double	16

and the code

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

int main()
{
cout << "bool:" << sizeof(bool) << endl;
cout << "char:" << sizeof(char) << endl;
cout << "wchar_t:" << sizeof(wchar_t) << endl;
cout << "char16_t:" << sizeof(char16_t) << endl;
cout << "char32_t:" << sizeof(char32_t) << endl;
cout << "short:" << sizeof(short) << endl;
cout << "int:" << sizeof(int) << endl;
cout << "long:" << sizeof(long) <<  endl;
cout << "long long:" << sizeof(long long) << endl;
cout << "float:" << sizeof(float) << endl;
cout << "double:" << sizeof(double) << endl;
cout << "long double:" << sizeof(long double) << endl;

return 0;
}
```

2 Pointer

1. Why do we need pointer?
    - 8 byte for any type in 64-bit system, 4 byte in 32-bit system
    - CallByPtr can change the parameters or not
    - Using “new” to allocate memory for dynamic memory
    - Faster than callByVal for comlpex type
2. How to allocate and free memory using “new” and “delete”?

```
typeName * pointerName = new typeName;

delete pointerName;
```

"delete" will free the memory, but not delete the pointer.

```
int *p = new int; // return memory address
*p = 10; // assignment directly
delete p; // free memory
int a =3;
*p = &a; // OK
```

3 Overload

1. When do we use overload?
   Funtion with different type parameters

2. What's the key of overload?
   Function signature, contains three points: number, type and order of parameters, complier realize overload by name mangling

3. How about reference in overload?
   Type and type reference share one function signature in complier, error will be raised if both of them exist

4. What happen if no funtion signature matches?
   Compiler attempt to cast it to funtion signature already exists, if more than one ways it can cast, error will be raised

5. How about keyword "const" in function signature ?
   Matching the corresponding function signature, if no match, cast will happen, "const" signature can match modifiable and const parameter, however, signature without "const" only match modifiable parameter.

4 Function Template

1. How to understand function template?
   Function with same algrithm for different type parameters

```
// function template prototype
template <typename T>
void swap(T &a, T &b)

// function template definition
template <typename T>
void swap(T &a, T &b)
{
  T temp;
  temp = a;
```

```
  a = b;
  b = temp;
}
```

2. How to understand overloaded function template?
   Function with different algrithms for different type parameters

```
// original template prototype
template <typename T>
void swap(T &a, T &b);

// new template prototype
template <typename T>
void swap(T *a, T *b, int n);
```

3. How to understand explicit specialization?
   Provides specific template definitions for specific types, prior to normal templates

```
// explicit specialization for the job type
template <>
void swap(job &a, job &b);
```

4. How to understand explicit instantiattion?
   Generate a function from template directly, less detection, more efficient

```
// explicit instantiattion for the job type in main
   ()
template
void swap(char &a, char &b);
```

5 Copy Contructor

1. How to use copy contructor?
   If there is a pointer variable initialized by "new" in class, **deep copy** should be set in copy constructor and "=" operator overload.

```
StringBad::StringBad(const StringBad & st)
{
num_string++;
len = st.len;
str = new char [len + 1];
std::strcpy(str, st.str);
}
```

```
StringBad & StringBad::operator=(const StringBad &
   st) //  e.g. S0 = "fox", tip: = operator overload
    don't change num_string
{
```

```
if (this == &st) // in case S0 = S0
return *this; // directly return
delete [] str; // free S0 str
len = str.len;
str = new char [len + 1]; // get space for new
    string
std::strcpy(str, st.str);
return *this;
}
```

4 statements will call copy constructor.

```
StringBad ditto(motto);
StringBad ditto = motto;
StringBad ditto = Stringbad(motto);
StringBad * ditto = new StringBad(motto);
```

Some tips:

- CallByVal and ReturnObj will call copy constructor(they both creates temporaty variable), we'd better to use reference.
- If there is a static data member in class, its value will change when new object is created, a explicit copy constructor should be set.

```
String::StringBad(const StringBad &s)
{
  num_strings++;
  ...// other code
}
```

- Default constructor in dynamic memoty:

```
String::String()
{
  len = 0;
  str = new char[1]; // Compatible with
      destructor "delete [] str"
  str[0] = '\0' ;
}
```

- Static member functions can't use "this" pointer, non-static data member.

```
static int HowMant() {return num_strings;} //
    declaration

int count = String::HowMany(); // how to use
```

6 Return Object

Containing 4 ways of returning object.

1. when we return "const ClassName &"
   Return the object in parameters list

```
const Vector & Vector::Max(const Vector & v1, const
    Vector & v2)
{
  if (v1.magval() > v2.magval())
    return v1;
  else
    return v2;
}
```

   Three points:

   - Returning object will call copy constructor, but object reference won't.
   - The object should exists when we return the object reference.
   - If all the parameters is "const ClassName &", one of them should be returned, we must return "const ClassName &".

2. when we return "ClassName &"?
   2 conditions:

   - overload assignment operator "=", more efficient

```
String & operator =(const String &);
String & operator =(const char *);

String s1("good stuff");
String s2, s3;
s1 = s2 = s3; // s2=s3 will change s2
```

   - overload output operator "<<", must obey

```
ostream & operator <<(ostream & os, const String
    & s1);
```

3. When we return "ClassName"?
   Return local variables, such as overload "+", "-", "*"

```
Vector Vector::operator+(const Vector &b) const
{
  return Vector(x+b.x, y+b.y);
}
```

4. When we return "const ClassName"?
   If we have overloaded operator "+" using 3, a strange property appears:

```
net = force1 + force2; // OK
force1 + force2 = net; // also OK
```

When we want to forbid the second statement, “const ClassName” should be used

```
const Vector Vector::operator+(const Vector &b)
    const
{
return Vector(x+b.x, y+b.y);
}
```

7 Class Inheritance(is-a)

This is a base class.

```
#include <string> // tabtenn0.h
using std::string;

class TableTennisPlayer
{
private:
    string firstname;
    string lastname;
    bool hasTable;
public:
    TableTennisPlayer(const string &fn = "none", const
        string &ln = "none", bool ht = false);
    void Name() const;
    bool HasTable() const {return hasTable;};
    void ResetTable(bool v) {hasTable = v;};
}
```

```
#include "tabtenn0.h" // cpp file
#include <iostream>

class TableTennisPlayer
{
    TableTennisPlayer::TableTennisPlayer(const string
       &fn, const string &ln, bool ht): firstname(fn),
        lastname(ln), hasTable(ht) {} // efficient

    void TableTennisPlayer::Name() const
    {
        std::cout << lastname << "," << firstname;
    }
}
```

RatedPalyer class derives from the TableTennisPlayer base class.

```
class RatedPlayer: public TableTennisPlayer //
    tabtenn1.h
{
private:
    unsigned int rating; // add a new data member
public:
    RatedPlayer(unsigned int r = 0, const string &fn =
        firstname, const string &ln = lastname, bool
       ht = false); // new constructor
    RatedPlayer(unsigned int r, const
       TableTennisPlayer & tp); // new constructor
    unsigned int Rating() const {return rating;} //
       new member function
    void ResetRating (unsigned int r) {rating = r;} //
       new member function
};
```

The first constructor:

```
RatedPlayer::RatedPlayer(unsigned int r, const string
    &fn, const string &ln, bool ht): TableTennisPlayer(
    fn, ln, ht)
{
    rating = r;
}
```

or

```
RatedPlayer::RatedPlayer(unsigned int r, const string
    &fn, const string &ln, bool ht): TableTennisPlayer(
    fn, ln, ht), rating(r)
{
}
```

The second constructor:

```
RatedPlayer::RatedPlayer(unsigned int r, const
    TableTennisPlayer &tp): TableTennisPlayer(tp)
{
    rating = r;
}
```

or

```
 RatedPlayer::RatedPlayer(unsigned int r, const
    TableTennisPlayer &tp): TableTennisPlayer(tp),
    rating(r)
 {
 }
```

Above all, 3 points can be obtained about constructor of derived class:

- Complier creates base class object before creating derived class object, the constructor of base class will called firstly as well
- Initialization list should be used to deliver the parameters from derived class to base in derived class construtors
- Derived class constructors should initial new data member

Special relationship between base and drived class:

- Derived class objects can call public base class methods

```
RatedPlayer rplayer(1140,"Mallory", "Duck", true);
rplayer.Name(); // OK
```

- Base class pointer and reference can point to derived class object, but not vice versa

```
RatedPlayer rplayer(1140,"Mallory", "Duck", true);
TableTennisPlayer &rt = rplayer; // OK
TableTennisPlayer *pt = &rplayer; //OK

TableTennisPlay player("Besty", "Bloop", true);
RatedPlayer &rr = player; // not OK
RatedPlayer *pr = &player; // not OK
```

- The base class pointer and reference which point to derived class can't invoke the derived class methods

```
RatedPlayer rplayer(1140,"Mallory", "Duck", true);
TableTennisPlayer &rt = rplayer;
TableTennisPlayer *pt = &rplayer;
rt.Name(); // OK
pt->Name(); // OK
rt.Rating(); // not OK
pt->Rating(); // not OK
```

- If we set a base class pointer or reference on parameter, both base and derived class object pointer or reference can be appected as argument

```
void Show(const TableTennisPlayer &rt)

TableTennisPlayer Player("Tara","Boomdea", flase);
RatedPlayer rplayer(1140,"Mallory", "Duck", true);
Show(player); // OK
Show(rplayer); // OK
```

- We can initial base class by derive class and assign a deirve class to base class

```
RatedPlayer olaf1(1140,"Mallory", "Duck", true);
TableTennisPlayer olaf2(olaf1); // initialization
```

```
RatedPlayer olaf1(1140,"Mallory", "Duck", true);
TableTennisPlayer winner
winner = olaf1; // assignment
```

8 Public Inheritance Polymorphism

In this section, keyword “vitual” is introduced.

```
class Brass // base class
{
...
public:
    void ViewAcct() const;
    virtual void Withdraw(double amt);
    virtual ~Brass() {};
...
};

class BrassPlus: public Brass // derived class
{
...
public:
        void ViewAcct() const;
    virtual void Withdraw(double amt);
...
}
```

Two kinds of Polymorphism Keyword “**vitual**” means two different in base and derived class.

```
Brass::ViewAcct();
BrassPlus::ViewAcct();
```