9 Inline Functions

Hu Sikang
skhu@163.com

School of Computer
Beijing Institute of Technology

- How to improve the efficiency ?
- In C, one of the ways to preserve efficiency is through the use of ***preprocessor macros***. The preprocessor replaces all macro calls directly with the macro code.
- In C++, there are two problems with preprocessor macros:
  - A macro can bury difficult-to-find bugs.
  - The preprocessor macros cannot be used as class member functions.
- To retain the efficiency of the preprocessor macro, but to add the safety and class scoping of true functions, C++ has the ***inline function***.

9.1 Preprocessor pitfalls

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

#define  f(x)  x*x
void main()
{
    int x(2);
    cout << f(x) << endl;
    cout << f(x+1) << endl;
}
```

Output:
4
5

9.2 Inline Functions

- When a function has several lines code but may be called frequently, we can use *inline* to save time and improve efficiency.
- *An* ***inline function*** is a true function, which is expanded in place, like a preprocessor macro, so the overhead of the function call is eliminated.
- You should (almost) never use macros, only inline functions.

9.2.1 Inline Functions

```
inline int add(int x, int y)
{
    return x + y;
}
void main()
{
    int a(1), b(2), c(3), sum;
    // Suggest compiler: replaced by sum=a+(b+c)
    sum = add(a, b+c);
}
```

9.2.2 Inline Functions

- Inline function definition must be appeared before its called.
- The body of inline function don’t include exception handling.
- The body of inline function don’t be recursive.

9.2.3 Inlines inside classes

- The “**inline**” keyword is not necessary inside a class definition.
- Any function you define inside a class definition is automatically an inline.

```
// Inlines inside classes
#include <iostream>
#include <string>
using namespace std;
class Point  {
    int i, j, k;
 public:
    Point( ): i(0), j(0), k(0)  {  }
    Point(int ii, int jj, int kk): i(a), j(b), k(c)  {   }
    void print(const string& msg = "") const    {
        if(msg.size() != 0)  cout << msg << endl;
        cout << i << ", " << j << ", " << k << endl;
    }
};
```

```
// Inlines outside classes
#include <iostream>
#include <string>
using namespace std;
class Point {
    int i, j, k;
 public:
    Point( );
    Point(int ii, int jj, int kk);
    void print(const string msg = "") const;
};

inline Point::Point( ): i(0), j(0), k(0) {  }
inline Point::Point(int ii, int jj, int kk): i(a), j(b), k(c) {   }
inline void  Point::print(const string& msg) const {
    if(msg.size() != 0) cout << msg << endl;
     cout << i << ", " << j << ", " << k << endl;
}
```

The definition of an **inline** is placed outside the class to keep the interface clean, using the **inline** keyword.

9.3 Hidden activities in constructors & destructors

```
class Member {
    int i, j, k;
public:
    Member(int x = 0) : i(x), j(x), k(x) { }
    ~Member() { cout << "~Member" << endl; }
};
class WithMembers {
    Member q, r, s;         // Have constructors?
    int i;
public:
    WithMembers(int a) :  q(a) { i = a; }     // Trivial?
    ~WithMembers() { cout << "~WithMembers" << endl;  }
};
```

```
void main()
{
    WithMembers  wm(1);
}
```

Summary

- Inline Function
- Inline function in the class
- Inline function VS. #define
- Inline & compiler
- Limitation when using inline function