C++ Efficiency Patterns

Chunting Gu, 2014/5

Construction & Destruction

C	C++
`{`	`{`
`  struct X x1;`	`  X x1;`
`  init(&x1);`	`  ...`
`  ...`	`}`
`  cleanup(&x1);`	
`}`	

C	C++
`{`	`{`
`  FILE* file = fopen( ... );`	`  std::fstream file( ... );`
`  ...`	`  ...`
`  fclose(file);`	`}`
`}`	

Implement Class String

```
class String {
  …
private:
  char* data_;
  size_t length_;
};
```

string a;

string b(“b”);

string c(a + b);

c = b;

```
class String {
public:
  String(const char* str = "") {
    Assign(str);
  }

  String(const String& rhs) {
    Assign(rhs.data_);
  }

  ~String() {
    delete[] data_;
  }

  String& operator=(const String& rhs) {
    if (&rhs != this) {
      delete[] data_;
      Assign(rhs.data());
    }
    return *this;
  }

private:
  void Assign(const char* str) {
    length_ = strlen(str);
    data_ = new char[length_ + 1];
    memcpy(data_, str, length_);
    data_[length_] = '\0';
  }
```

Temporaries

Postpone variable definitions as long as possible.

```
string encryptPassword(const string& password) {
  string encrypted;

  if (password.length() < MINIMUM_PASSWORD_LENGTH) {
    throw logic_error("Password is too short");
  }

  // do whatever is necessary to place an encrypted
  // version of password in encrypted
  return encrypted;
}
```

Temporaries

Initialize the variable when it's declared.

```
std::string str;
str = “...”;
```

```
std::string str = “...”;
```

Temporaries

The Return Value Optimization (demo)

Return constructor arguments instead of local objects.

```
Complex operator+(const Complex& lhs, const Complex& rhs) {
  Complex c;
  c.real_ = lhs.real_ + rhs.real_;
  c.imag_ = lhs.imag_ + rhs.imag_;
  return c;
}
```

```
Complex operator+(const Complex& lhs, const Complex& rhs) {
  return Complex(lhs.real_ + rhs.real_, lhs.imag_ + rhs.imag_);
}
```

Temporaries

Return const reference instead of value.

```
class Patient {
public:
  const string& name() const {
    return name_;
  }

private:
  string name_;
};
```

Temporaries

Assign to const reference instead of value.

```
string name = patient.name();
const string& name = patient.name();
```

```
map<string, string> patient_info;
string    name = patient_info[“name”];
string&   name = patient_info[“name”];
```

operator= returns reference instead of value.

Temporaries

Pass input parameters by const reference instead of value.

```
void f(string s);
```

```
char* s1 = malloc(strlen(s) + 1);
strcpy(s1, s);
f(s1); // void f(const char* s);
free(s1);
```

```
void f(const string& s);
```

Temporaries

Temp variable created on parameter mismatch.

```
void f(const string& name);
f("adam");
```

A temp string object is created.

```
string temp(“adam”);
f(temp);
```

Temporaries

Overload for const char* to avoid temp variable.

```
const string& subfamily = image.GetSubfamily();
if (subfamily == “PANO” || subfamily == “CRPANO”) {
    ...
}
```

Not only:

```
bool operator==(const string& lhs, const string& rhs);
```

But also:

```
bool operator==(const string& lhs, const char* rhs);
```

Temporaries

Key Points

- A temporary object could penalize performance twice in the form of constructor and destructor computations.
- Declaring a constructor explicit will prevent the compiler from using it for type conversion behind your back.
- A temporary object is often created by the compiler to fix a type mismatch. You can avoid it by function overloading.
- Avoid object copy if you can. Pass and return objects by reference.
- You can eliminate temporaries by using <op>= operators where <op> may be +, -, *, or /.

Reference

Why did C++ add references?

- In order to support operator overloaded, the existing type representations within C were inadequate.
- Bjarne Stroustrup felt that an additional type representation, that of a reference, needed to be introduced.
- A reference supports the syntax of object access while at the same time supporting the shallow intialization.

Reference

Example

Given a class named Matrix, to support the operator + for it, the solution without reference is:

```
Matrix operator+(Matrix m1, Matrix m2);
```

You know that it has performance issue, because it passes parameters “by value”.

Using pointers elliminates the performance issue:

```
Matrix operator+(Matrix* m1, Matrix* m2);
```

But the the pointer syntax is non-intuitive and error-prone as well.

Being non-intuitive is unacceptable since the idea behind operator overloading is to make the use of the class object intuitive.

The pointer syntax cannot stop programmers to write the code like this:

```
Matrix d = &a + &b + &c; // oops!
```

It cannot compile!

Inline

Inlining replaces method calls with a macro-like expansion of the called method within the calling method.

C:

```c
#define MAX(a, b) ((a) > (b) ? (a) : (b))
```

C++:

```cpp
template <typename T>
inline T max(const T& a, const T& b) {
  return (a > b ? a : b);
}
```

How to inline?

- Prefix the method definition with the reserved word inline.
- Define the method within the declaration header.

```
class Patient {
  int age_;
public:
  int age() const { return age_; }
  void set_age(int age) { age_ = age; }
};
```

Or

```
inline int Patient::age() const {
  return age_;
}
inline void Patient::set_age(int age) {
  age_ = age;
}
```

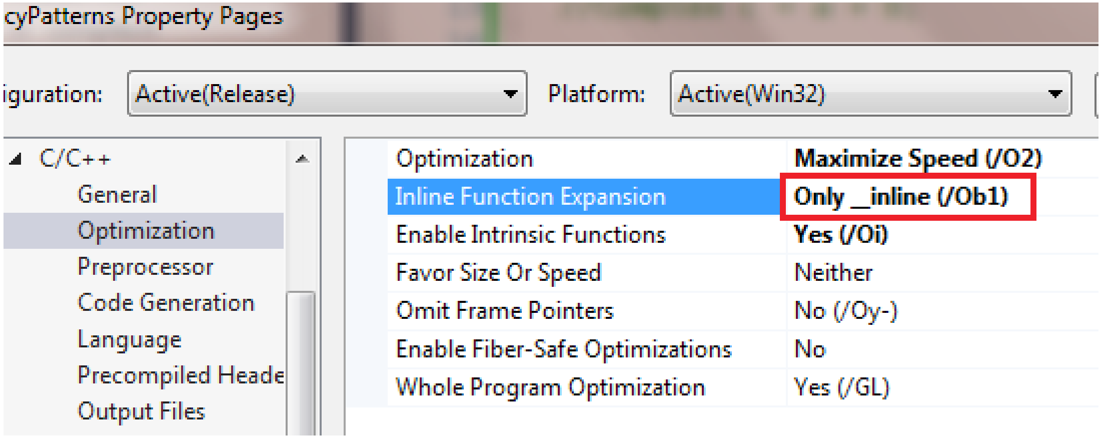
Inline

Example (demo)

```
//inline
int calc(int a, int b) {
  return a + b;
}

int main() {
  int x[1000];
  int y[1000];
  int z[1000];
  for (int i = 0; i < 1000; ++i) {
    for (int j = 0; j < 1000; ++j) {
      for (int k = 0; k < 1000; ++k) {
        z[i] = calc(y[j], x[k]);
      }
    }
    }
}
```

VC++ configuration:



cyPatterns Property Pages

iguration: Active(Release) Platform: Active(Win32)

Optimization	Maximize Speed (/O2)
Inline Function Expansion	Only __inline (/Ob1)
Enable Intrinsic Functions	Yes (/Oi)
Favor Size Or Speed	Neither
Omit Frame Pointers	No (/Oy-)
Enable Fiber-Safe Optimizations	No
Whole Program Optimization	Yes (/GL)

STL

C++ Standard Template Library.

Vector

A sequence container that encapsulates dynamic size arrays.

The complexity (efficiency) of common operations on vectors is as follows:

- Random access – constant O(1)
- Insertion or removal of elements at the end - amortized constant O(1)
- Insertion or removal of elements - linear in distance to the end of the vector O(n)

List

A double-linked list.

- Constant time insertion and removal of elements from anywhere in the container.
- Fast random access is not supported.
- It is usually implemented as double-linked list.

Map

A sorted associative container that contains key-value pairs with unique keys.

- Search, removal, and insertion operations have logarithmic complexity.
- Maps are usually implemented as red-black trees.

Set

An associative container that contains a sorted set of unique objects.

- Search, removal, and insertion operations have logarithmic complexity.
- Sets are usually implemented as red-black trees.

STL

Well defined interfaces.

`vector`

clear, insert, erase, push_back, pop_back, resize, swap

`list`

clear, insert, erase, push_back, pop_back, push_front, pop_front, resize, swap, remove, remove_if, sort

`map`

..., find, ...

STL

Key Points

- The STL is an uncommon combination of abstraction, flexibility, and efficiency.
- Depending on your application, some containers are more efficient than others for a particular usage pattern.
- Unless you know something about the problem domain that the STL doesn't, it is unlikely that you will beat the performance of an STL implementation by a wide enough margin to justify the effort.
- It is possible, however, to exceed the performance of an STL implementation in some specific scenarios.

Coding Optimizations

Redundant Computations

For example,

```
for (i = 0; i < 100; i++) {
  a[i] = m * n;
}
```

The value of m*n is computed every iteration. Lifting invariants outside the loop optimizes the performance.

```
int k = m * n;
for (i = 0; i < 100; i++) {
  a[i] = k;
}
```

You really don't have to understand the overall program design. Understanding the for loop is sufficient.

Caching

Evaluating constant expressions inside a loop is a well known performance inefficiency.

```
For (... ;!done;... ) {
  done = patternMatch(pat1, pat2, isCaseSensitive());
}
```

Compute case-sensitivity once, cache it in a local variable, and reuse it.

```
int isSensitive = isCaseSensitive();
for (... ;!done;... ) {
  done = patternMatch(pat1, pat2, isSensitive);
}
```

Precompute

Lookup tables for Fast Fourier Transform.

```
FFT::FFT(int n) {
  int m = n / 2;
  cos_table_ = new double[m];
  sin_table_ = new double[m];

  // Precompute lookup tables.
  double f = -2 * PI / n;
  double d;
  for (int i = 0; i < m; i++) {
    d = f * i;
    cos_table_[i] = cos(d);
    sin_table_[i] = sin(d);
  }
}
```

```
FFT::~FFT() {
  delete[] cos_table_;
  delete[] sin_table_;
}
```

```
void FFT::Transform(double* x, double* y) {
  ...
}
```

Reduce Flexibility

Make simplifying assumptions that boost performance.

A web server allocates heap storage to hold the client IP address for every request, and deallocates at the end of the request handling.

Calling new() and delete() for heap memory is expensive.

An IP address cannot be longer than 15 bytes.

```
XXX.XXX.XXX.XXX
```

IPv6) will be longer, but still bounded. It is more efficient to store the IP address on the stack as a local variable:

```
char clientIPAddr[256];
```

Lazy Evaluation

```
int route(Message *msg) {
  ExpensiveClass upstream(msg);
  if (goingUpstream) {
    ... // do something with the expensive object
  }
  // upstream object not used here.
  return SUCCESS;
}
```

Defer object creation to the correct scope.

```
  if (goingUpstream) {
    ExpensiveClass upstream(msg);
    ...
  }
```

Delay the object construction until you have all the necessary ingredients for an efficient construction.

```
void f() {
  string s; // 1
  ...
  char* p = "Message"; // 2
  ...
  s = p; // 3
  ...
}
```

```
void f() {
  ...
  char* p = "Message";
  string s(p);
  ...
```

Useless Computations

```
class Student {
public:
  Student(const char* name);
private:
  string name_;
};
```

```
Student::Student(const char* name) {
  name_ = name;
}
```

```
Student::Student(const char* name) : name_(name) {
}
```

Initialization list is prefered.

Memory Management

Allocating and deallocating heap memory on the fly is expensive.

```
void f() {
  X* x = new X;
  ...
  delete x;
}
```

```
void f() {
  X x;
  ...
} // no need to delete x
```

Using stack object is prefered.

Library And System Calls

Take the case of concatenating two strings.

```
char s0;
char* s1 = "Hello";
char * s2 = "World";
s0 = (char*)malloc(strlen(s1) + strlen(s2) + 1);
strcpy(s0, s1); // Copy first string
strcat(s0, s2); // Concatenate the second
```

In C++, all these details are encapsulated in a string class implementation. The overloaded "+" operator for string would simplify our client code to this:

```
string s1 = "Hello";
string s2 = "World";
string s0 = s1 + s2;
```

Be aware of hidden costs as they may have a profound impact on system performance.

The Final Example: Regex

SLRE: http://slre.sourceforge.net

Parse a 15647 lines C++ file.

Modes	**boost::regex**	**slre**
Debug	8949 ms	1408 ms
Release	257 ms	391 ms

Trust the data!