

Efficient C++

An overview for High Performance Programming

Berenger Bramas - November 2011

Table of contents

Classes

[Avoid virtual](#)

[Copy only if needed](#)

[Using explicit](#)

[Be careful with memcpy](#)

[Initialize attributes early](#)

Functions and methods

[Functions/methods types](#)

[Value, reference or pointer](#)

[RVO \(Return Value Optimization\)](#)

Stream

[Do not flush](#)

Misceallineous

[Using C libraries](#)

[Using restrict](#)

[Using const](#)

[Using inline](#)

[Pre/Post increment](#)

[Using metaprog](#)

[Do not use exceptions](#)

[MPI](#)

[Compilation](#)

[Debug](#)

[No macro functions or define values](#)

[Prefer stack to heap](#)

[Using register](#)

[Prefetch](#)

[Declare variable when needed](#)

Stl

[Using the Stl containers](#)

[Iterators](#)

Annexes - Tools

[ITAC - Intel Trace Collector](#)

[VTune - Study and improve your threaded application](#)

[DDT - Debug your parallel application](#)

References

Classes

Avoid virtual

Calling a virtual function is slow for several reasons:

- It cannot be optimized at compile time, and cannot be inlined
- It requires to read the pointer, then the “vtable” and finally access the method address
- Accessing the “vtable” can cause a cache miss since the “vtable” exists for a class and you are working on an object
- A “vtable” pointer is initialized for each object in the constructor

You can define an interface but keep in mind that calling a virtual function costs a lot especially if this function is called many times.

```
class SuperClass {
public:
    virtual ~SuperClass(){}
    virtual void print(){}
};

class SubClass : public SuperClass {
public:
    void print(){}
};

SuperClass* const pt = new SubClass;
pt->print(); // Use the vtable!

SubClass* const pt = new SubClass;
pt->print(); // Do not use the vtable!
```

Use RTTI the less as possible

Then what are the solutions when we really need inheritance. It depends on the case. If you are working with an extremely small number of items and an extremely small number of calls you can continue to use it. If you are working with an array of elements which are all from the same class you can use the RTTI to test the first element and then hardly cast all the other items using a conditional statement or use a template if the information is known at compile time.

Template

Template is an efficient feature. Use it every time you can move computation and work from the run time to the compile time. It may help you to avoid inheritance if you know object type at compile time.

```
class SuperClass {
```

```

public:
    virtual ~SuperClass(){}
    virtual void print() = 0;
};

class SubClass : public SuperClass {
public:
    void print(){}
};

void Work(SuperClass* const object ){
    object->print();
}

template <class T>
void WorkTemp(T* const object ){
    object->print();
}

SubClass object;
Work(&object);
WorkTemp<SubClass>(&object);

```

Copy only if needed

Implicit copy is extremely frequent in C++. Be aware when you are using a copy and tried to minimize it. You can forbid copy by putting copy constructor and affectation operator private.

```

class ICannotBeCopied {
    ICannotBeCopied(const ICannotBeCopied& ){}
    ICannotBeCopied& operator=(const ICannotBeCopied& ){ return *this; }

public:
    ICannotBeCopied(){}
};

```

Using explicit

As you may miss some shadow copies, you can miss implicit conversion. To avoid this situation add the keyword “explicit” in front of your constructor. Even if compiler now tries to minimize temporary variables it still happen, using “explicit” is a way to avoid it.

Be careful with memcpy

This very useful and efficient function from C can provoke hard to debug problems. In fact, you cannot copy any objects or any structure as you can do in C even if you are managing your pointer correctly. In fact, the compiler can put some information into your class like a “vtable”

pointer. Copying them can cause run time problem.

To use it safely make your class a POD (Plain Old Data) class. To do so, avoid:

- User defined constructor
- User defined destructor
- User defined copy operator
- Virtual functions
- Base classes
- Private or protected nonstatic members
- Nonstatic data members that are references

Initialize attributes early

Do not initialize attributes in the constructor body. In fact, by doing this the compiler will use the default constructor for your attributes and the assignment in the constructor body. But, empty constructor plus assignment operator may be extremely more costly compare to a specific constructor.

<pre>// Bad class AClass{ string name; public: AClass(const char* const inName){ name = inName; } };</pre>	<pre>// Good class AClass{ string name; public: AClass(const char* const inName) : name(inName) { } };</pre>
--	--

Functions and methods

Functions/methods types

static methods, global functions or nonvirtual methods have the same call cost: a jump in memory.

Value, reference or pointer

As we advice to avoid copy it is useful to use pointers and references. References are the same as constant address pointer.

<pre>void foo(AClass& anObject, AClass* const anotherObject){ ... }</pre>

The only difference is the code readability but it is efficiently equivalent. In case of returning an

object you may consider to fill using a variable declared outside the function.

<pre>AClass anObject = foo(); ... void foo(){ AClass anObject; anObject.setValue(50); return anObject; }</pre>	<pre>AClass anObject; foo(&anObject); ... void foo(AClass* const anObject){ anObject->setValue(50); }</pre>
---	--

In case of native type (long long, long int, ...) passing by value is not more expensive passing by values than passing by references.

RVO (Return Value Optimization)

The RVO is a compiler optimization to avoid a copy when returning an object in a function. It will perform the previous optimization, passing an object reference and fill it instead of returning an object and copying it.

But to help the compiler applying RVO your class must have a copy constructor and your function must be simple with only one return statement.

Stream

Do not flush

In every cases, using `cstdio` or `iostream`, it is important not to flush the output if it is not needed. Flushing the `stdout/output` takes time and if you flush it every time you print your program is going to loose time. With `cstdio`, flushing is allowed using the `fflush` function. With `iostream` this is done using the `flush` method or the `"std::endl"`.

<pre>std::cout << "This line is flushed!" << std::endl; std::cout << "This line is not flushed until buffer gets full." << "\n";</pre>
--

Misceallineous

Using C libraries

When there is an equivalent is it better to use the C++ libraries. Anyway, sometime the C functions have been implemented in C++ so using C from C++ is normal. But, it is advised to be careful in the include statement ("`stdio.h`" becomes "`cstdio`", ...).

Using restrict

Even if the “restrict” keyword is defined as “telling the compiler that an object is pointed by only one pointer” it does not related to threading. In fact, “restrict” has to be used when you have several pointer of the same type and you know that there are unique.

```
void foo(int* value1, int* value2){
    (*value1) = 10; // Does it change value2 value?
    ...
}

void foo(int* __restrict__ value1, int* __restrict__ value2){
    (*value1) = 10; // I know it does not change value2 value
    ...
}
```

Using const

One C++ best practice is to use “const” whenever it is possible. Also, it is recommended to do so from the beginning of the development and not at the end.

Use “const”:

- In method declaration
- In arguments declaration
- In pointer constant value and constant address

The following code illustrates some “const” usage.

```
class AClass {
    ...
    void aMethod() const {
    }
    ...
};

void foo(const AClass& o, AClass* const pt, const AClass* const constPt ){
    ...
}
```

Using inline

Adding the “inline” keyword in a function declaration or defining a method inside the class declaration give hint to the compiler to make this functions inlined. Nevertheless, the compiler decides what is better and may not use your advice.

Pre/Post increment

Use post increment whenever it is possible even in loop progress part. Of course, when working with integer, the compiler usually detect that using the pre-inc is equivalent and faster but in case of more complicated structure, like an iterator, the compiler will not detect it.

```
for( int idx = 0 ; idx < limite ; ++idx ){  
    ....  
}
```

Using metaprog

Metaprog is a powerfull feature of C++. Anyway, the situation where metaprog can be used are mostly rare.

Do not use exceptions

Exceptions is a nice way to handle errors. But it is extremely slow. We recommend to use return value like in C to manage errors in your application.

MPI

Do not use the C++ mpi binding: It is obsolete!

Compilation

About C++ compilation we advice:

- use level 2 of optimization “-O2”, most of the time this is the more efficient output application
- compile with intel to check the compatibility of your application
- use compile flag to ensure the safest compilation. We use “-Wall -Wshadow -Wpointer-arith -Wcast-qual -Wconversion”

Debug

Like most of research application you might have some part related to debug that slow down you program. This compilation statement can be performed using two different approach as illustrated in the following code sample.

```
#include <stdio>  
  
// Comment if needed  
#define USE_DEBUG  
  
#ifndef USE_DEBUG  
    #define LDEBUG(X) X;  
    #define DEBUG_START  
    #define DEBUG_END
```

```

#else
#define LDEBUG(X)
#define DEBUG_START if(false){
#define DEBUG_END }
#endif

int main(){
    LDEBUG( printf("I am in debug\n") );

    DEBUG_START
    printf("I am in debug too\n");
    DEBUG_END

    return 0;
}

```

In case you are not using the debug mode, the “DEBUG_START/END” will be replaced by a “if(false)” statement. With a level 2 of optimization the compiler will remove this statement.

No macro functions or define values

In C++ it is a best practice to remove macro functions or defines values from your code. Otherwise, you have to use inline function and “const” global variables.

<pre> #define MAX 1000 #define Max(X,Y) ((X) > (Y) ? (X) : (Y)) </pre>	<pre> static const int MAX = 1000; template <class NumType> inline NumType Max(const NumType v1, const NumType v2){ return v1 > v2 ? v1 : v2; } </pre>
--	--

Prefer stack to heap

Static allocation is much more faster than dynamic allocation, not only in the allocation step but also in the access. So when it is possible use static allocation which put data in the stack. In other words, use “new/delete” only if you cannot do differently.

Using register

The “register” keyword is a hint for the compiler to advice to put a variable in one of the CPU registers. Like “inline” the compiler is free to ignore the hint.

Prefetch

In some situation it is good to add personal prefetch in the code. Nevertheless, there is not standard keyword for that.

```
#ifdef __GNUC__
    #define Prefetch_Read(X) __builtin_prefetch(X)
    #define Prefetch_Write(X) __builtin_prefetch(X,1,1)
#else
    #ifdef __INTEL_COMPILER
        #define Prefetch_Read(X) _mm_prefetch(X,_MM_HINT_T0)
        #define Prefetch_Write(X) _mm_prefetch(X,_MM_HINT_T0)
    #else
        #warning compiler is not defined
        #define Prefetch_Read(X)
        #define Prefetch_Write(X)
    #endif
#endif
#endif
```

Declare variable when needed

In C++ you can declare a variable anywhere you want. But taking advantage of that we can:

- Initialize each variable we declare
- Construct a variable only if we need it

Stl

Using the Stl containers

C++ comes with a entire standard library that includes containers. These containers are sufficient in most of the cases, also they are working efficiently and have only a few bugs. You may use the stl in these situations:

- you want to test the core of your code quickly and do not want to implement your own containers
- the container you are using do not count a lot in your application efficiency
- you are using more than just the containers and you do not want to implement all the stl part you need

However, when using stl containers, keep in minds how they work to avoid heavy mistakes like: inserting in the front of a vector, accessing randomly in a list, etc....

In our case, we use our own container to gain profit from the knowledge we have about our data and their use.

Iterators

Fallow some rules with the stl iterators:

- use const when appropriate

- do not create temporary objects on a loop
- use pre-inc
- if you modify a container, iterator can become invalide

```
// Bad !  
for(list<int>::iterator iter = list.begin() ; iter != list.end() ; iter++ ){  
....  
}  
  
// Good !  
const list<int>::iterator end = list.end();  
for(list<int>::iterator iter = list.begin() ; iter != end ; ++iter ){  
....  
}
```

Annexes - Tools

ITAC - Intel Trace Collector

“Intel® Trace Collector for MPI applications produces tracefiles that can be analyzed with Intel® Trace Analyzer performance analysis tool.” In other word, ITAC is a set of tools to create, view and analyse trace in parallel and distributed applications. It works with both thread and MPI.

Usage:

In C++, to describe your application with ITAC you need to create ITAC objects at the beginning of functions or regions you want to trace. The creation of this object marks the beginning of the area and the end of life of this same object means the end of the area.

```
VT_Region::VT_Region(const char * symname, const char * classname)
VT_Region::VT_Region(const char * symname, const char * classname, const
char * file, int line)

VT_Function::VT_Function(const char * symname, const char *
classname)
VT_Function::VT_Function(const char * symname, const char * classname,
const char * file, int line)
```

Example:

```
void AClass:AMethod(){
    VT_Function function(__FUNCTION__, "Fmm" , __FILE__ , __LINE__);
    ....
    {
        VT_Region region("My Compute Region", __FUNCTION__ , __FILE__ , __LINE__)
        ....
    }
}
```

Because we usually do not want to use trace in a release version, the best thing to do is to use a macro.

```
// #define SCALFMM_USE_TRACE

#ifndef SCALFMM_USE_TRACE
    #define FTRACE( X )
#else
    #define FTRACE( X ) X
#endif

// Then use it as
```

```
FTRACE( VT_Function function(__FUNCTION__, "Fmm" , __FILE__ , __LINE__ ) );
```

Compilation:

ITAC requires "VT.h" from `-$VT_ROOT/include` directory and "libVT.a".

Example:

```
mpicxx -I$VT_ROOT/include -trace -openmp files.cpp ../Src/Utils/FDebug.cpp -O2 -o file.exe  
// or  
mpicc ctest.o -L$VT_LIB_DIR -IVT $VT_ADD_LIBS -o ctest
```

Execution:

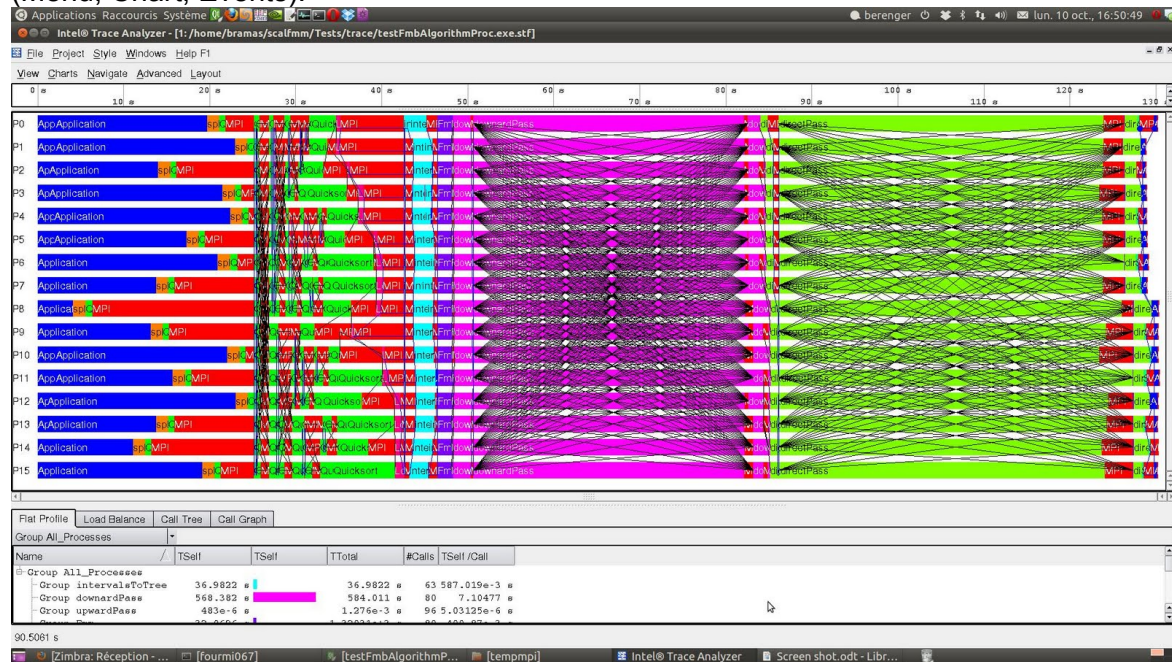
ITAC puts the trace files in the exec directory. It is advised to start your application from a separate directory. You have to start your application as usual to generate the trace.

View:

The trace generates several files, we only use the .stf file.

```
>> traceanalyzer testFmbAlgorithmProc.exe.stf
```

It opens the Trace Analyzer, from here you can group/ungroup events and plotting a chart (Menu, Chart, Events).



Thread trace:

You can trace thread activity by creating a VT Region at the beginning of life of the thread and destroy it when the thread is over.

Example:

```
#pragma omp parallel  
{
```

```
FTRACE( VT_Region region("Thread", __FUNCTION__ , __FILE__ , __LINE__ ) );
...
}
```

Links:

<http://software.intel.com/en-us/articles/intel-trace-analyzer-and-collector-users-guides-pdf/>

VTune - Study and improve your threaded application

VTune is a tool to analyse your application threaded or not. Compile your application with the intel compiler and run it with VTune to see all the hardware counter and the execution details. You can view the details by compiling with “-g” option.

Links:

http://software.intel.com/sites/products/documentation/hpc/amplifierxe/en-us/2011Update/win/ug_docs/index.htm

DDT - Debug your parallel application

DDT is a debugger for parallel application, it supports MPI, OpenMP and PThread.

References

Efficient C++ Performance Programming Techniques, Dov Bulka, David Mayhew, Addison Wesley, 1999

C++ Footprint and Performance Optimization, R. Alexander, G. Bensley, Addison Wesley, 2000

C++ for Game Programmers, Noel Llopis, Charles River Media, 2003

Exceptional C++: 47 Engineering Puzzles, Programming Problems, and Solutions, Herb Sutter, Addison Wesley, 1999

More Exceptional C++, Herb Sutter, Addison Wesley, 2001