Basic C++ performance issues

Author: *Sebastien Binet*
Institute: *LAL/IN2P3*
Date: *2011-10-24*

Overview

- Constructors and destructors
- Temporaries
- Cost of virtual functions
- Cost of exceptions
- If and when to inline functions
- Standard library containers
- Templates

Common vocabulary - goal

- `C++` performance has many aspects
  - execution speed
  - code size
  - data size
  - memory footprint at run-time
  - time and space consumed by the edit/compile/link cycle
- `C++` is a *large* language with many features, idioms and constructs
  - constructors/destructors, exceptions, templates, late-binding, overloading, RAII, ...
  - knowing (or having a rough idea of) the cost of these features is important for building a (re-)usable efficient application
    - model of time and space overheads of various `C++` language features

Classes and inheritance

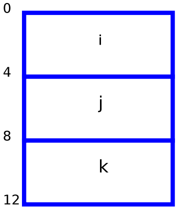
C++ supports object-oriented programming

- involves (possibly deep) inheritance hierarchies of classes
- operations performed on classes and class hierarchies
- space and time overheads of using classes instead of structs ?

Representation overhead

- C++ class with no virtual function
  - no space overhead *wrt* a good old C struct
  - WYSIWYG
  - non-virtual functions do *NOT* take any space in an object
  - ditto for static data
  - ditto for static function

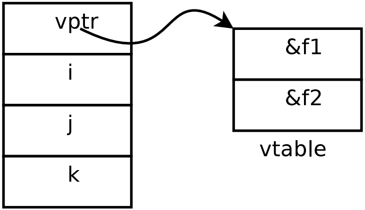
```
struct C
{
  int i;
  int j;
  int k;
};
```




```
class Cxx
{ public:
  int i;
  int j;
  int k;
};
```

Representation overhead

```
class Polymorphic
{
    virtual void f1();
    virtual void f2();
    int i;
    int j;
    int k;
};
```



- a polymorphic class (with at least one virtual function)
  - per-object overhead of 1 pointer (`vptr`)
  - per-class overhead of a `virtual function table`
    - 1 or 2 words per virtual function
  - per-class overhead of a `type information object` (RTTI)
    - `O(10)` bytes
    - name string (identifying the class)
    - couple of words of more infos
    - couple of words for each base class

Basic classes operations

- cost of calling non-virtual, non-static, non-inline member function
- compared to calling a freestanding function with one extra pointer

basic fct call	timings
non-virtual	
`px->f(1)`	0.016
`g(ps,1)`	0.016
non-virtual	
`x.g(1)`	0.016
`g(&s,1)`	0.016
static fct mbr	
`X::h(1)`	0.013
`h(1)`	0.013

Virtual functions

- calling a virtual function
- calling a function through a pointer stored in an array

virtual fct call	timings
virtual	
`px->f(1)`	0.019
`x.f(1)`	0.016
ptr-to-fct	
`p[1](ps,1)`	0.016
`p[1](&s,1)`	0.018

Virtual functions of class templates

- new C++ support structures (vtbl) for **each** specialization
- pure replication of code at the instruction level
- workarounds
  - use non-template helper functions
  - factor out non-parametric functionalities into a non-templated base class

```
void foo_helper_fct(...);
template<class T> class Foo
{...};

class Base { void dostuff(); };
template<class T> class Derived : public Base
{...};
```

Inlining

- calling a function has a cost
- for simple functions, it may be pure overhead
- inlining: directly copy callee's body at call site

	timings
non-inline	
`px->g(1)`	0.016
`x.g(1)`	0.016
inline	
`px->k(1)`	0.006
`x.k(1)`	0.005
macro	
`K(ps,1)`	0.005
`K(&s,1)`	0.005

Multiple inheritance

- more complicated binary layout of instances
- for each call, need to *adjust* the `this` pointer to get the right substructure
  - caller applies an offset to `this` from the `vtbl`
  - or use a `thunk`: man-in-the-middle fragment of code

	timings
SI, non-virtual `px->g(1)`	0.016
Base1, non-virtual `pc->g(1)`	0.016
Base2, non-virtual `pc->gg(1)`	0.017
SI, virtual `px->f(1)`	0.019
Base1, virtual `pa->f(1)`	0.019
Base2, virtual `pa->ff(1)`	0.024

Virtual base classes

- additional overhead *wrt* simple multiple inheritance
  - position of base class subobject not known at compile time
  - needs one additional indirection

	timings
SI, non-virtual `px->g(1)`	0.016
VBC, non-virtual `pd->gg(1)`	0.021
SI, virtual `px->f(1)`	0.019
VBC, virtual `pa->f(1)`	0.025

Exception handling

- systematic and robust way to cope with errors
- traditional alternatives
  - returning error codes
  - setting error states indicators (`errno`)
  - calling error handling functions
  - escaping into error handling code using `longjmp`
  - passing along a pointer to a state object w/ each call

```
double f1(int a) { return 1.0 / a; }
double f2(int a) { return 2.0 / a; }
double f3(int a) { return 3.0 / a; }

// no error handling
double g(int x, int y, int z)
{ return f1(x) + f2(y) + f3(z); }
```

Exception handling

- with error handling

```
int error_state = 0;
double f1(int a) {
  if (a <= 0) {
    error_state = 42;
    return 0;
  }
  return 1.0 / a;
}

double g(...) {
  double xx = f1(x);
  if (error_state) {...}
  ...
  return xx+yy+zz;
}
```

- with EH

```
struct Err {...};
double f1(int a) {
  if (a <= 0)
    throw Error(42);
  return 1.0 / a;
}

double g(...) {
  try {
    return f1(x)+f2(y)
           +f3(z);
  } catch (Err& err) {
    ... }
}
```

Exception handling

- 3 sources of overhead
  - data and code associated with `try` blocks
  - data and code associated with the normal execution of additional fcts
  - data and code associated with `throw` expressions
- implementation issues
  - context setup of `try` blocks for associated `catch` clauses
  - `catch` clause needs some kind of type identification
  - clean-up of handled exceptions (memory mgt)
  - ctors/dtors of non-trivial objects
  - ...
- 2 main implementation techniques
  - the 'code' approach
  - the 'table' approach
- both need some kind of RTTI (thus code/data increase)

Exception handling

- the 'code' approach
  - dynamically maintain auxiliary data structures
    - to manage execution contexts
    - to track the list of objects to be unwound (in case an exception occured)
  - associated stack and run-time costs can be significant
  - even when no exception is thrown, bookkeeping is performed
- the 'table' approach (g++)
  - read-only tables are generated
    - to determine the current execution context
    - to locate catch clauses
    - to track the list of objects to be unwound
  - all bookkeeping is pre-computed
  - no run-time cost if no exception is thrown (zero cost overhead for normal execution path)

Templates

- template overheads
  - for each new specialization, generation of a new instantiation of code
  - *can* lead to unexpectedly large amount of code and data
    - EH, vtbl, ...
  - canonical experiment:
    - instantiate 100 `std::list<T*>` for some fixed `T` type
    - instantiate 1 `std::list<T*>` for 100 `T` different types
    - measure programs' size
  - optimization:
    - recognize that all different specializations project onto the same generated machine code
    - can be done by the compiler
    - or by a clever STL implementation
    - ie: implement (under the hood) all `std::list<T*>` in terms of `void*`
  - compilation time

Templates vs inheritance

- templates are usually more runtime efficiency friendly
- deep inheritance trees incur overhead:
  - ctors/dtors
  - pointer indirection / virtual functions

Programmer directed optimizations

usual disclaimer:

- don't do it:
  - early (performance) optimization is the root of all evil
  - spend that time on unit tests (make sure the code is right), documentation and new features
- think twice before applying performance any optimization tips
- make it thrice

---

in the following:

- a few rules of thumb
- cover usual gotchas

Constructors & Destructors

- C++ creates instances of classes with ctors
  - allocate memory
  - initialize fields
- ... and cleans-up/relinquishes resources with dtors

```
/* in good old C */          |  // in C++
{                            |  {
  struct S s;                |    S s;
  S_init(&s);                |    // compute s...
  /* compute s... */         |  }
  S_cleanup(&s);
}
```

in an **ideal** world: **no overhead** introduced by ctor/dtor

- in **practice**:
  - overhead because of inheritance
  - overhead because of composition
- overhead: perform computations which may be rarely needed

Object construction

- in ctors prefer to use initializers
  - no need to do the work twice

```
UsuallyOk::UsuallyOk(...) : m_1(42), m_2(str) {...}

UsuallyBad::UsuallyBad(...)
{ m_1 = ...; m_2 = str; }
```

- define variables as close to use-site than possible
- define variables when ready to initialize (no ctor+assign)

```
X x1 = 42;     X x2; x2 = 42;
```

- passing arguments to a function by value is...
  - **cheap** for built-ins
  - potentially **expensive** for class types
  - prefer passing by const-ref or address

```
void f(const std::string&);
void g(const T*);
```

Implicit conversions & temporaries

- Calling a function with the 'wrong' arg.'s type implies type conversion
- may require work at run-time

```
void f1(double);
f1(7.0);  // no conversion but copy
f1(7);    // conversion: f1(double(7));

void f2(const double&);
f2(7.0); // no conversion
f2(7);   // const double tmp =7; f2(tmp);

void f3(std::string);  std::string s = "foo";
f3(s);     // no conversion but copy
f3("bar"); // f3(std::string("bar"))

void f4(const std::string&);
f4(s);     // no conversion, no copy
f4("f");   // const std::string tmp("f"); f4(tmp);
```

Explicit constructors

consider the class definition:

```
class Rational
{
  friend Rational operator+(const Rational&,
                            const Rational&);
public:
  Rational(int a=0, int b=1) : num(a), den(b) {}
private:
  int num; // Numerator
  int den; // Denominator
};
```

Explicit constructors

and the following snippet:

```
Rational r;
// ...
r = 100;
```

- no assignment operator with `int` so the above will be "translated" to:

```
Rational tmp(100);
r.operator=(tmp);
tmp.~Rational();
```

- usually a good idea to define ctors which can be called with one argument, as **explicit**:

```
explicit Rational(int a=0, int b=1) : num(a), den(b) {}
```

- also good to overload `operator=(T)`

Default constructors

```
class X
{
  A a;
  B b;
  virtual void fct();
};

class Y : public X
{
  C c;
  D d;
};
```

```
class Z : public Y
{
  E e;
  F f;
 public:
  Z() {}
};

Z z;
```

- compiler-generated default constructors are inline
- substantial (!) amount of machine code can be inserted each time a Z is constructed...

Temporary objects

- probably the most acute problem wrt performance and efficiency.
- preventing creation of temporaries benefits
  - run-time speed
    - creating temporaries takes CPU cycles
    - destroying them, too !
  - memory footprint
- understand how and when compilers generate temporary objects
  - initializing objects
  - passing parameters to functions
  - returning values from functions

Temporaries & initialization

quick example:

```
{
  std::string s1 = "Hello";
  std::string s2 = "World";
  std::string s3;

  s3 = s1 + s2; // s3 is now: "HelloWorld"
}
```

where the last statement is equivalent to:

```
{
  std::string _temp;
  operator+(_temp, s1, s2);          // pass _temp by reference
  s3.std::string::operator=(_temp); // assign _temp to s3
  _temp.std::string::~string();     // destroy _temp
}
```

on top of that, the string concatenation function may itself create temporaries.

Temporaries, loops and type mismatch

- what's wrong with that code (short of being midly useful) ?

```
Complex operator+(const Complex& rhs,
                  const Complex& lhs);

Complex a, b;
for (int i=0; i<100; ++i) a = i*b + 1.0;
```

- temporary generated to represent the complex `1+0j`
- lift the constant expression out of the loop

```
Complex one(1.0);
for (int i=0; i<100; ++i) a = i*b + one;
```

- a clever optimizer *might* do it for you (YMMV)

Eliminate temporaries with [some-op]=()

the following snippet generates 3 temporaries:

```
std::string s1,s2,s3,s4;
std::string s5 = s1 + s2 + s3 + s4;
```

the following does not:

```
std::string s5 = s1;
s5 += s2;
s5 += s3;
s5 += s4;
```

Pass by value

avoid writing APIs which use this pattern

```
void f(T t) { /* do something with t*/ }

{
  T t;
  f(t);
}
// is equivalent to:
{
  T t;
  T _temp;
  _temp.T::T(t);  // copy construct _temp from t
  f(_temp);       // pass _temp by reference
  _temp.T::~T();  // destroy _temp
}
```

Return by value

another source of temporaries is function return value:

```
std::string fct()
{
  std::string s;
  ... // compute 's'
  return s;
}

// the following snippet:
{
  std::string p;
  // ...
  p = fct();
}
```

```
// is equivalent to: (pseudo-code)
{
  std::string p;
  // ...
  std::string _temp;
  // pass _temp by reference
  fct(_temp);

  // assign _temp to p
  p.std::string::operator=(_temp);

  // destroy _temp
  _temp.std::string::~string();
}
```

Return value - corollary

- so we don't like (performance-wise) functions which return objects

```
class T
{
 public:
  T operator++(int i); // foo++
  T operator++();      // ++foo
  ...
};
```

- prefer prefix over postfix increment operator

```
for (std::vector<T>::iterator
      it = vec.begin(),
      end= vec.end();
     it != end; ++it) { // <-- and NOT: it++
   //...
}
```

Return value optimization (RVO)

- one way to side-step inefficiency of return by value: write 'C-like' APIs:

```
T fct();
T t;
//...
t = fct();

void compute_t(T& t);
T t;
compute_t(t);
```

- another way is to enable the compiler to apply RVO...

RVO

```
class Complex {
 public:
  Complex(double re=0., double im=0.);
  double re, im;
};

Complex operator+(const Complex& a, const Complex& b) {
  Complex res;
  res.re = a.re + b.re;
  res.im = a.im + b.im;
  return res;
}

Complex c1,c2,c3;
c3 = c1 + c2;
```

RVO

- without any optimization, the emitted (pseudo)code would look like:

```
Complex _tmp;
_add_complex(_tmp, c1, c2);
c3.operator=(_tmp);
_tmp.~Complex();

void _add_complex(Complex &_tmp,
                  const Complex &a, const Complex &b) {
  Complex ret;
  //... as previously
  _tmp.operator=(ret);
  ret.~Complex();
  return;
}
```

- how to remove all these temporaries and their associated c/dtors ?

RVO

- rewrite the add function to remove the local named temporary
- use an unnamed temporary to help the compiler:

```
Complex operator+(const Complex &a, const Complex &b) {
 double re = a.re + b.re;
 double im = a.im + b.im;
 return Complex(re, im);
}
```

- note that complicated functions with multiple return statements are harder to elect for RVO
- RVO is **not mandatory**
  - done at the discretion of the compiler
  - inspection of generated code + trial&error

inlining basics

- replaces a function call with a verbatim copy of the function at call-site
  - kind of like a C-macro
- works around the overhead of calling functions.
- 2 ways to express *intent* of inlining a function

```
class FourMom {
 float m_px, m_py, m_pz, m_ene;
public:
 // implicit inlining:
 // definition provided w/ declaration
 float px() const { return m_px; }
 void set_px(float px);
};

// use inline keyword
inline void FourMom::set_px(float px) { m_px = px; }
```

inlining basics

- at source-code level, inlined functions are used like any other function:

```
int main(int, char**)
{
 FourMom mom;
 mom.set_px(20.*GeV);
 std::cout << "px: " << mom.px()
           << std::endl;
 return 0;
}
```

- code expanded inline at call site:
  - call site must know the definition of the function
  - compilation coupling
  - potential compilation time increase

cross-call optimizations

```
int main(int, char**)
{
 FourMom mom;
 mom.set_px(20.*GeV);
 std::cout << "px: " << mom.px()
           << std::endl;
 return 0;
}
```

- inlining is most nutritious with cross-call optimizations

cross-call optimizations

```
int main(int, char**)
{
 FourMom mom;
 mom.m_px = 20.*GeV;
 std::cout << "px: " << mom.m_px
           << std::endl;
 return 0;
}
```

- inlining is most nutritious with cross-call optimizations

cross-call optimizations

```
int main(int, char**)
{
 FourMom mom;
 mom.m_px = 20.*GeV;
 std::cout << "px: " << mom.m_px
           << std::endl;
 return 0;
}
```

- inlining is most nutritious with cross-call optimizations

```
int main(int, char**)
{
 std::cout << "px: " << 20.
           << std::endl;
 return 0;
}
```

why not inline

- code expansion
  - disk space
  - memory size
  - cache size, increase cache fault
  - code size
- compilation coupling
- recursive methods

Standard Template Library (STL)

- a powerful combination of containers and generic algorithms
- performance guarantees of the asymptotic complexity of containers and algorithms:
  - an approximation of algorithm performance - big-O notation
  - `O(N), O(N*N),...`
- choosing the right container is based on the type of frequent and critical operations applied on it
  - various trade-offs
  - no one true best container
  - only best compromise for task at hand
- containers manage storage space for their elements
- provide methods to access elements, directly or through iterators

std::vector

- a sequence container
- organize data into a strictly linear arrangement
- contiguous storage
- good locality of reference
- allow `O(1)` random access
- inefficient at removing/inserting elements other than at the end: `O(N)`
- do not forget to give **adequate hint size** before `push_back` calls:

```
std::vector<T> v;
v.reserve(n);
v.push_back(make_t());
```

- prefer to use `container::empty()` instead of `container::size()==0`

std::list

- a sequence container
- doubly linked list
- efficient insertion and removal anywhere in the container: `O(1)`
- efficient at moving (blocks of) elements within the container or between containers (`O(1)`)

associative containers

- `std::map<K,V,Cmp,Alloc>`
  - unique key-values
  - elements follow a strict weak ordering (at all time)
  - efficient access of elements by key (logarithmic complexity)
  - logarithmic complexity for insertion
- `std::tr1::unordered_map<K,V,Hash,Pred,Alloc>` (hash_map)
  - unique key-values
  - constant time insertion/access

better than STL ?

- STL is generic
- if you know something about the problem's domain, you can squeeze some perfs wrt STL.

e.g. compare strings of a known format "aaaa1" and "aaaa2"

- the STL is an uncommon combination of abstraction, flexibility and efficiency (curtosy of generic programming)
- 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 STL doesn't, it is unlikely you will beat STL by a wide enough margin
- outperforming STL is still possible in some specific scenarios

Concluding remarks

- C++ is a wide and powerful language, difficult to really master entirely
- be wary of using fancy constructs and features
  - when in doubt, choose simplicity
- pay attention to compiler warnings
- strive for warning-free builds
- innocently looking C++ code can be treacheous
- profile before sprinkling your code with optimizations
- remember the code the C++ compiler automatically generates for you
- remember the trade-offs of inlining

Remember, with great power, comes great responsibility