UCLA Extension Course

C++ Performance Issues

Douglas C. Schmidt

Professor
d.schmidt@vanderbilt.edu
www.cs.wustl.edu/∼schmidt/

Department of EECS
Vanderbilt University
(615) 343-8197

C++ Performance Issues Overview

- Construction/destruction
- Inlining
- Virtual functions
- Static and dynamic libraries
- Dynamic allocation
- Compiler optimizations
- Generality *vs.* performance
- General performance strategies

Construction/Destruction

- Pass-by-value copies objects.
  - Constructor called on creation, destructor called at end of function (or function call, for return values).
  - Pass objects (of types that have constructors/destructors) by **reference**, instead.
    - Use **const** reference, to be safe.
- Don't create local objects unless necessary; create them in innermost scope.

```
// Don't create foo here!
if (option) {
  Foo foo;  // foo only created if option is enabled
  // . . .
}
```


Construction/Destruction

- Use initializer list to avoid default construction of contained objects.

```
template <class T>
Stack<T>::Stack (size_t max_size) {
  array_ = Array<T> (max_size);  // Very inefficient!  array_
                                 // already initialized using
                                 // its default constructor.
  // [. . .]
}
```

- Consider inlining constructors and destructors.
  - Though be *very* careful with inline destructors. For local (stack) objects, they'll be called for every path out of a function. And if the destructor is virtual, it should **not** be inline.
- Bulka and Mayhew measured about 60 percent decrease in performance for additional destructor call.

Inlining

- Inlining removes function call overhead.
- Two ways to inline:
  - Add **`inline`** keyword to function **definition**:

```
inline
int
Foo::status () const { return status_; }
```

  - Define the function in the **class declaration**:

```
class Foo {
public:
  int status () const { return status_; }
};
```

- **`inline`** keyword is suggestion to compiler.

Inlining Mechanics

```
class Foo {
 // [. . .]
 static int incr (int i);
};

int
Foo::incr (int i) {
  return i + 17;
}
```

not inlined:

```
int y = // [. . .];

z = Foo::incr (y);
```

1) save register state
2) set up argument (y)
3) jump to Foo::incr () code
4) add 17 to argument
5) set up return value
6) return to caller
7) restore register state

inlined:

```
int y = // [. . .];

z = y + 17;
```

1) add 17 to y
2) place result in z

Conditional Inlining Support

```
#define INLINE inline  /* Comment out to
                          disable inlining. */

Foo.h:
  class Foo {
  public:
    int status () const;
  };

#if defined (INLINE)
# include "Foo.i"
#endif /* INLINE */

Foo.i:
  INLINE
  int
  status () const {
    // . . . return the status
  }

Foo.cc:
#if !defined (INLINE)
# include "Foo.i"
#endif /* INLINE */
```


Effects of Inlining

- Positive
  - Speeds execution, due to removal of function call overhead.
  - Speeds execution, due to more aggressive optimization.
  - For small functions such as accessors, can cause code size **decrease**!
- Negative
  - For large functions, causes code size increase.
  - Some functions cannot be inlined.
  - Debuggers usually do not see inline functions.

Virtual Functions

- Virtual functions add overhead.
  - Construction requires setup of vtable pointer (single **`long`** copy).
  - Virtual function call is indirect, through vtable.
  - Inlining not possible if object type cannot be determined at compile time.
- Virtual function call time can be 2 to 3 times as non-virtual call.
  - 10's of nanoseconds on several hundred MHz CPU.
  - Insignificant penalty for large functions.
  - Modern compilers can usually remove all of the penalty.
- Second-order effects can be very significant: vtable access can cause cache misses.

Static and Dynamic Libraries

- A static (archive, **`.a`**) library is simply a collection (plus optional index) of object (**`.o`**) files.
  - Linking extracts copies of **`.o`** files from static library and places them in executable.
- A dynamic (shared object, **`.so`**) library resides in memory. Any process (owned by any user) can call its code.
  - Therefore, the (shared) code must be position independent.
  - Called dynamic because actual linking is done at run-time.
  - Each process gets a copy of the static (global) data in the dynamic library.

Dynamic Library Implications for C++

- Dynamic libraries are slower due to position independent object code.
  - Position independence implemented via added level of indirection.
  - In addition to first-order cost of indirection, indirection increases likelihood of cache misses.
- 18 to 25 percent slower for a representative TAO example.

Dynamic Allocation

- Avoid dynamic allocation on critical paths.
  - Allocation/deallocation itself is slow due to heap management.
  - With multithreading, must serialize heap management.
- Fragmentation can impair performance, so avoid repetitive allocation + deallocation.
- If dynamic allocation is necessary, try to do it before entering performance-critical sections.
- Use pools of objects.

Compiler Optimizations

- **`-O`** usually enables optimization, though many compilers have other, more specific or aggressive options, *e.g.*, **`-O3`**, **`-fast`**.
  - Optimization can greatly increase compile time.
  - Optimization can hinder debugging, because the object code no longer directly corresponds to the source code.
  - Optimization can overly aggressive.
- Some compilers disable optimization with **`-g`**. (**`g++`** does not.)

Performance and Generality

- Container design usually trades off performance and generality.
  - For specific applications, custom containers may provide better performance.
  - STL provides good performance, given its generality.
  - For general purpose applications, it's likely that STL will give better performance than a one-off solution.
- Another example of the tradeoff: **`memcpy`** *vs.* **`memmove`** (and **`bcopy`**). **`memcpy`** is faster, but does not allow overlap.
- STL tries to be minimal, but not at the cost of performance.
  - Equality operator is required only for performance.

General Performance Strategies

- Beware of the 80-20 "rule":
  - **80 percent of execution time is spent in only 20 percent of the code.**
- Performance problems are often due to just a few small implementation decisions.
  - (assuming that the design supports good performance)
- Use tools to help isolate performance problems.
  - *e.g.*, time probes (**`gethrtime ()`**), **`prof`**/**`gprof`**, Quantify

For More Information

- Bulka and Mayhew, *Efficient C++: Performance Programming Techniques*, Addison-Wesley, 1999.
  - Andrew summarized in `http://students.cec/~agg1/c++/performance.html`
- Compiler documentation, *e.g.*, **`info gcc`**, for optimization options and discussion.