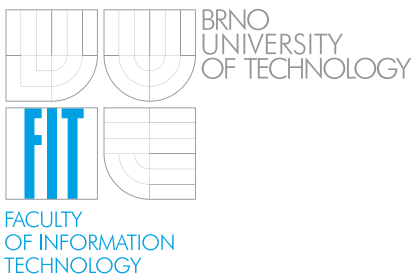
Optimizing C++ Code

Petr Zemek

Brno University of Technology, Faculty of Information Technology
Božetěchova 2, 612 00 Brno, Czech Republic
http://www.fit.vutbr.cz/~izemek



Decompiler Seminar, 2013-10-29

Contents

What is *optimization*?

What is *optimization*?

Typical Types of Optimizations

- execution time
- memory usage
- size of the binary
- power consumption
- communication
- ...

What is *optimization*?

Typical Types of Optimizations

- execution time
- memory usage
- size of the binary
- power consumption
- communication
- ...

Main Focus of This Presentation

- optimizing the speed of C++ code

- Why should we optimize?

- Why should we optimize?
- *Golden rule*: do not optimize

- Why should we optimize?
- *Golden rule*: do not optimize
- *Pareto principle*: 90/10

Basics

- Why should we optimize?
- *Golden rule*: do not optimize
- *Pareto principle*: 90/10
- finding code to be optimized

Basics

- Why should we optimize?
- *Golden rule*: do not optimize
- *Pareto principle*: 90/10
- finding code to be optimized
  - guessing

- Why should we optimize?
- *Golden rule*: do not optimize
- *Pareto principle*: 90/10
- finding code to be optimized
  - guessing
  - measuring time
    - `time`
    - `time(),clock()`

- Why should we optimize?
- *Golden rule*: do not optimize
- *Pareto principle*: 90/10
- finding code to be optimized
  - guessing
  - measuring time
    - `time`
    - `time()`, `clock()`
  - profiling (`gprof`, `valgrind`, ...)

- Why should we optimize?
- *Golden rule*: do not optimize
- *Pareto principle*: 90/10
- finding code to be optimized
  - guessing
  - measuring time
    - `time`
    - `time()`, `clock()`
  - profiling (`gprof`, `valgrind`, ...)
- too many links $\Rightarrow$ may seem like black magic

Example of "Black Magic"

```
 1 const unsigned SIZE = 32768;
 2 int data[SIZE];
 3
 4 for (unsigned c = 0; c < SIZE; ++c) {
 5     data[c] = std::rand() % 256;
 6 }
 7
 8 long long sum = 0;
 9 for (unsigned i = 0; i < 100000; ++i) {
10     for (unsigned c = 0; c < SIZE; ++c) {
11         if (data[c] >= 128) {
12             sum += data[c];
13         }
14     }
15 }
16 std::cout << "sum = " << sum << "\n";
```

Running time: 11.54s

Example of "Black Magic" (Continued)

```
 1 const unsigned SIZE = 32768;
 2 int data[SIZE];
 3
 4 for (unsigned c = 0; c < SIZE; ++c) {
 5     data[c] = std::rand() % 256;
 6 }
 7
 8 std::sort(data, data + SIZE);
 9
10 long long sum = 0;
11 for (unsigned i = 0; i < 100000; ++i) {
12     for (unsigned c = 0; c < SIZE; ++c) {
13         if (data[c] >= 128) {
14             sum += data[c];
15         }
16     }
17 }
18 std::cout << "sum = " << sum << "\n";
```

Running time: 1.93s (without sorting: 11.54s)

Example of "Black Magic" (Continued)

```
T = branch taken
N = branch not taken

// With sorting.
data[] = 0, 1, 2, 3, ... 127, 128, 129, ... 251, 252,...
branch = N  N  N  N  ...   N    T    T  ...   T    T ...
       = NNNNNNNNNNN ... NNNNNNNTTTTTTT ... TTTTTTTT ...

// Without sorting.
data[] = 226, 185, 125, 158, 100, 144, 217, 79, 202, ...
branch =   T    T    N    T    N    T    T   N    T  ...
       = TTNTNTTNT ...   (random)
```

Example of "Black Magic" (Continued)

```
1  T = branch taken
2  N = branch not taken
3
4  // With sorting.
5  data[] = 0, 1, 2, 3, ... 127, 128, 129, ... 251, 252,...
6  branch = N  N  N  N  ...   N    T    T  ...   T    T ...
7         = NNNNNNNNNNN ... NNNNNNNTTTTTTT ... TTTTTTTT ...
8
9  // Without sorting.
10 data[] = 226, 185, 125, 158, 100, 144, 217, 79, 202, ...
11 branch =   T    T    N    T    N    T    T   N    T  ...
12        = TTNTNTTNT ...   (random)
```

http://stackoverflow.com/q/11227809

Levels of Optimization

- design
  - algorithms (linear search vs binary search)
  - data structures (array vs tree)

Levels of Optimization

- design
  - algorithms (linear search vs binary search)
  - data structures (array vs tree)
- language
  - specifics (pointers vs values)
  - standard library (`std::map<>` vs custom solution, `GMP`)

Levels of Optimization

- design
  - algorithms (linear search vs binary search)
  - data structures (array vs tree)
- language
  - specifics (pointers vs values)
  - standard library (`std::map<>` vs custom solution, `GMP`)
- compiler
  - selection (`gcc` vs `icc` vs ...)
  - optimizations (`-O0`, `-O1`, `-O2`, `-O3`, `-Os`, `-Og`, `-Ofast`)

Levels of Optimization

- design
  - algorithms (linear search vs binary search)
  - data structures (array vs tree)
- language
  - specifics (pointers vs values)
  - standard library (`std::map<>` vs custom solution, `GMP`)
- compiler
  - selection (`gcc` vs `icc` vs ...)
  - optimizations (`-O0`, `-O1`, `-O2`, `-O3`, `-Os`, `-Og`, `-Ofast`)
- operating system
  - scheduling (process, I/O), preemption (`nice`)
  - virtual memory management (paging algorithms)

Levels of Optimization

- design
  - algorithms (linear search vs binary search)
  - data structures (array vs tree)
- language
  - specifics (pointers vs values)
  - standard library (`std::map<>` vs custom solution, `GMP`)
- compiler
  - selection (`gcc` vs `icc` vs ...)
  - optimizations (`-O0`, `-O1`, `-O2`, `-O3`, `-Os`, `-Og`, `-Ofast`)
- operating system
  - scheduling (process, I/O), preemption (`nice`)
  - virtual memory management (paging algorithms)
- hardware
  - processor speed, memory size, storage device
  - memory hierarchy (registers, cache, main memory, disk, network)
  - out-of-order execution

Optimization Techniques

- caching

Optimization Techniques

- caching
- using optimized libraries (`GMP`, `BOOST`)

Optimization Techniques

- caching
- using optimized libraries (`GMP`, `BOOST`)
- non-traditional data structures (trie, BDD, ...)

Optimization Techniques

- caching
- using optimized libraries (`GMP`, `BOOST`)
- non-traditional data structures (trie, BDD, ...)
- parallelization

Optimization Techniques

- caching
- using optimized libraries (`GMP`, `BOOST`)
- non-traditional data structures (trie, BDD, ...)
- parallelization
- improving locality of reference

Optimization Techniques

- caching
- using optimized libraries (`GMP`, `BOOST`)
- non-traditional data structures (trie, BDD, ...)
- parallelization
- improving locality of reference
- load everything in memory

Optimization Techniques

- caching
- using optimized libraries (`GMP`, `BOOST`)
- non-traditional data structures (trie, BDD, ...)
- parallelization
- improving locality of reference
- load everything in memory
- pre-computing values

Optimization Techniques

- caching
- using optimized libraries (`GMP`, `BOOST`)
- non-traditional data structures (trie, BDD, ...)
- parallelization
- improving locality of reference
- load everything in memory
- pre-computing values
- movement of invariants before loops

Optimization Techniques

- caching
- using optimized libraries (`GMP`, `BOOST`)
- non-traditional data structures (trie, BDD, ...)
- parallelization
- improving locality of reference
- load everything in memory
- pre-computing values
- movement of invariants before loops
- rewriting parts of code into assembly (portability :( )

Profilers

- What are they? How do they work?

Profilers

- What are they? How do they work?

Popular Profilers

- `gprof`
- `valgrind --tool=callgrind` + `kcachegrind`

Profilers

- What are they? How do they work?

Popular Profilers

- `gprof`
  - `g++ ... -o prog -pg -g`
  - `./prog`
  - `gprof prog gmon.out > analysis.txt`
  - `$EDITOR analysis.txt`
- `valgrind --tool=callgrind` + `kcachegrind`

Profilers

- What are they? How do they work?

Popular Profilers

- `gprof`
  - `g++ ... -o prog -pg -g`
  - `./prog`
  - `gprof prog gmon.out > analysis.txt`
  - `$EDITOR analysis.txt`
- `valgrind --tool=callgrind` + `kcachegrind`
  - `g++ ... -o prog -g`
  - `valgrind --tool=callgrind --dump-instr=yes --collect-jumps=yes ./prog`
  - `kcachegrind callgrind.out.PID`

- What are they? How do they work?

Popular Profilers

- `gprof`
  - `g++ ... -o prog -pg -g`
  - `./prog`
  - `gprof prog gmon.out > analysis.txt`
  - `$EDITOR analysis.txt`
- `valgrind --tool=callgrind` + `kcachegrind`
  - `g++ ... -o prog -g`
  - `valgrind --tool=callgrind --dump-instr=yes --collect-jumps=yes ./prog`
  - `kcachegrind callgrind.out.PID`

(DEMO)

Useless Copying

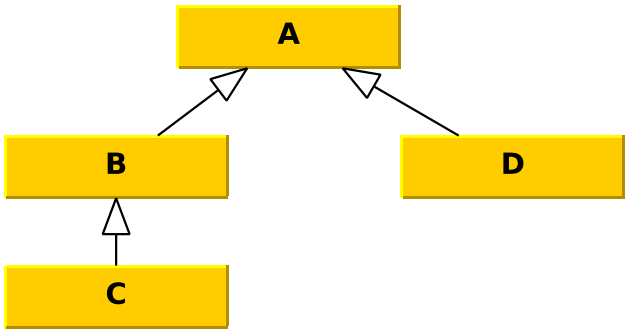
```
 1 int f1(std::string s, unsigned i) {
 2     return s[i];
 3 }
 4
 5 int f2(const std::string &s, unsigned i) {
 6     return s[i];
 7 }
 8
 9 const unsigned SIZE = 500000;
10 std::string s(SIZE, 'a');
11 unsigned sum = 0;
12 for (unsigned i = 0; i < SIZE; ++i) {
13     sum += fX(s, i);
14 }
```

Useless Copying

```
 1 int f1(std::string s, unsigned i) {
 2     return s[i];
 3 }
 4
 5 int f2(const std::string &s, unsigned i) {
 6     return s[i];
 7 }
 8
 9 const unsigned SIZE = 500000;
10 std::string s(SIZE, 'a');
11 unsigned sum = 0;
12 for (unsigned i = 0; i < SIZE; ++i) {
13     sum += fX(s, i);
14 }
```

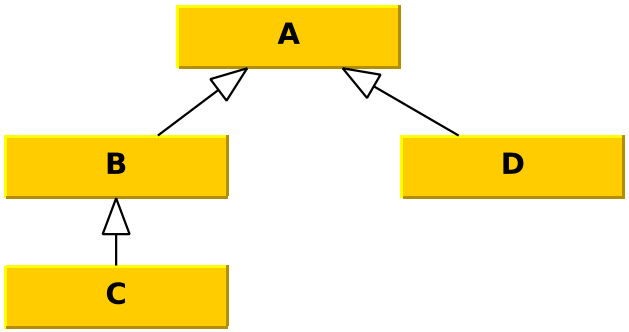
fX	Time (`gcc4.8 -O2`)	Time (`clang3.2 -O2`)
f1	21.5s	21.6s
f2	0.003s	0.003s

dynamic_cast<>




```
1 unsigned cnt = 0;
2 A *p = new C;
3 for (unsigned i = 0; i < 100000000; ++i) {
4     if (dynamic_cast<X *>(p)) {
5         cnt++;
6     }
7 }
8 std::cout << "cnt = " << cnt << "\n";
```

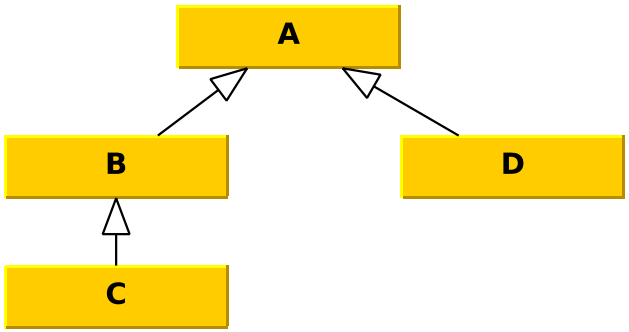
dynamic_cast<>




```
1 unsigned cnt = 0;
2 A *p = new C;
3 for (unsigned i = 0; i < 100000000; ++i) {
4     if (dynamic_cast<X *>(p)) {
5         cnt++;
6     }
7 }
8 std::cout << "cnt = " << cnt << "\n";
```

X	Time (`gcc4.8 -O2`)	Time (`clang3.2 -O2`)
`A`	0.0s	0.0s

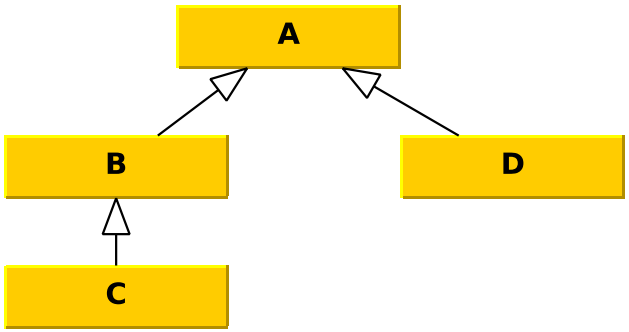
dynamic_cast<>




```
1 unsigned cnt = 0;
2 A *p = new C;
3 for (unsigned i = 0; i < 100000000; ++i) {
4     if (dynamic_cast<X *>(p)) {
5         cnt++;
6     }
7 }
8 std::cout << "cnt = " << cnt << "\n";
```

X	Time (`gcc4.8 -O2`)	Time (`clang3.2 -O2`)
`A`	0.0s	0.0s
`C`	1.6s	4.1s

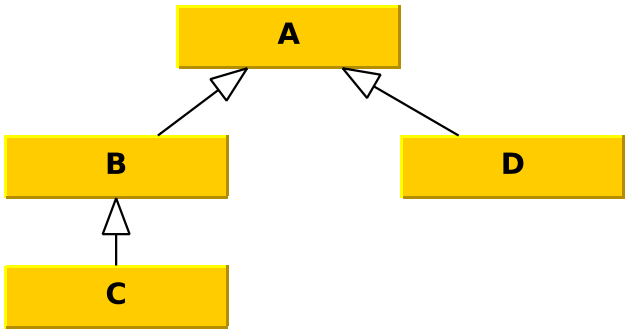
dynamic_cast<>




```
1 unsigned cnt = 0;
2 A *p = new C;
3 for (unsigned i = 0; i < 100000000; ++i) {
4     if (dynamic_cast<X *>(p)) {
5         cnt++;
6     }
7 }
8 std::cout << "cnt = " << cnt << "\n";
```

X	Time (`gcc4.8 -O2`)	Time (`clang3.2 -O2`)
A	0.0s	0.0s
C	1.6s	4.1s
B	3.6s	5.2s

dynamic_cast<>




```
1 unsigned cnt = 0;
2 A *p = new C;
3 for (unsigned i = 0; i < 100000000; ++i) {
4     if (dynamic_cast<X *>(p)) {
5         cnt++;
6     }
7 }
8 std::cout << "cnt = " << cnt << "\n";
```

X	Time (`gcc4.8 -O2`)	Time (`clang3.2 -O2`)
A	0.0s	0.0s
C	1.6s	4.1s
B	3.6s	5.2s
D	6.8s	5.4s

Buffering

```
1 for (size_t i = 0; i < 10000000; ++i) {
2     // Print "Here comes a string..." and a new line
3 }
4
5 // (1)
6 std::cout << "Here comes a string..." << std::endl;
7
8 // (2)
9 std::cout << "Here comes a string..." << '\n';
```

Buffering

```
1 for (size_t i = 0; i < 10000000; ++i) {
2     // Print "Here comes a string..." and a new line
3 }
4
5 // (1)
6 std::cout << "Here comes a string..." << std::endl;
7
8 // (2)
9 std::cout << "Here comes a string..." << '\n';
```

X	Time (`gcc4.8 -O2`)	
(1)	real	1m3.699s
	user	0m4.030s
	sys	0m34.203s
(2)	real	0m2.066s
	user	0m0.810s
	sys	0m0.687s

Exploiting Exceptions

```
 1 unsigned f1(unsigned i) {
 2     throw i;
 3 }
 4
 5 unsigned f2(unsigned i) {
 6     return i;
 7 }
 8
 9 unsigned sum = 0;
10 for (unsigned i = 0; i < 10000000; ++i) {
11     try {
12         sum += fX(i);
13     } catch (unsigned value) {
14         sum += value;
15     }
16 }
```

Exploiting Exceptions

```
 1 unsigned f1(unsigned i) {
 2     throw i;
 3 }
 4
 5 unsigned f2(unsigned i) {
 6     return i;
 7 }
 8
 9 unsigned sum = 0;
10 for (unsigned i = 0; i < 10000000; ++i) {
11     try {
12         sum += fX(i);
13     } catch (unsigned value) {
14         sum += value;
15     }
16 }
```

fX	Time (`gcc4.8 -O2`)	Time (`clang3.2 -O2`)
f1	31.1s	26.1s
f2	0.007s	0.004s

Iterating Over Containers

```
1 typedef std::vector<int> IntVec;
2 IntVec v(500000000, 0);
```

Iterating Over Containers

```
1 typedef std::vector<int> IntVec;
2 IntVec v(500000000, 0);

3 // (1) gcc4.8 -O0 -> 9.39s
4 for (IntVec::iterator i = v.begin(); i != v.end(); i++) {
5     *i = 5;
6 }
```

Iterating Over Containers

```
1  typedef std::vector<int> IntVec;
2  IntVec v(500000000, 0);

3  // (1) gcc4.8 -O0 -> 9.39s
4  for (IntVec::iterator i = v.begin(); i != v.end(); i++) {
5      *i = 5;
6  }

7  // (2) gcc4.8 -O0 -> 7.61s
8  for (IntVec::iterator i = v.begin(); i != v.end(); ++i) {
9      *i = 5;                                          //  ^^^
10 }
```

Iterating Over Containers

```
1  typedef std::vector<int> IntVec;
2  IntVec v(500000000, 0);

3  // (1) gcc4.8 -O0 -> 9.39s
4  for (IntVec::iterator i = v.begin(); i != v.end(); i++) {
5      *i = 5;
6  }

7  // (2) gcc4.8 -O0 -> 7.61s
8  for (IntVec::iterator i = v.begin(); i != v.end(); ++i) {
9      *i = 5;                                          //  ^^^
10 }

11 // (3) gcc4.8 -O0 -> 4.52s
12 for (IntVec::iterator i = v.begin(), e = v.end();
13         i != e; ++i) {                  // ^^^^^^^^^^^
14     *i = 5;
15 }
```

Iterating Over Containers

```
1  typedef std::vector<int> IntVec;
2  IntVec v(500000000, 0);

3  // (1) gcc4.8 -O0 -> 9.39s
4  for (IntVec::iterator i = v.begin(); i != v.end(); i++) {
5      *i = 5;
6  }

7  // (2) gcc4.8 -O0 -> 7.61s
8  for (IntVec::iterator i = v.begin(); i != v.end(); ++i) {
9      *i = 5;                                          //  ^^^
10 }

11 // (3) gcc4.8 -O0 -> 4.52s
12 for (IntVec::iterator i = v.begin(), e = v.end();
13         i != e; ++i) {                    // ^^^^^^^^^^^
14     *i = 5;
15 }

16 // (4) gcc4.8 -O0 -> 2.99s
17 for (IntVec::size_type i = 0, e = v.size(); i < e; ++i) {
18     v[i] = 5;
19 }
```

Iterating Over Containers (Continued)

```
1  typedef std::vector<int> IntVec;
2  IntVec v(500000000, 0);

3  // (1) gcc4.8 -O2 -> 0.31s
4  for (IntVec::iterator i = v.begin(); i != v.end(); i++) {
5      *i = 5;
6  }

7  // (2) gcc4.8 -O2 -> 0.31s
8  for (IntVec::iterator i = v.begin(); i != v.end(); ++i) {
9      *i = 5;                                          //  ^^^
10 }

11 // (3) gcc4.8 -O2 -> 0.31s
12 for (IntVec::iterator i = v.begin(), e = v.end();
13         i != e; ++i) {                    // ^^^^^^^^^^^
14     *i = 5;
15 }

16 // (4) gcc4.8 -O2 -> 0.30s
17 for (IntVec::size_type i = 0, e = v.size(); i < e; ++i) {
18     v[i] = 5;
19 }
```

Last Sips

- optimize only if
  - your code is working
  - you have tests for your code
  - you know what should be optimized (profiling)
  - the optimization is worth it (work, time, readability)

Last Sips

- optimize only if
  - your code is working
  - you have tests for your code
  - you know what should be optimized (profiling)
  - the optimization is worth it (work, time, readability)
- do not perform useless micro-optimizations (e.g. `a << 1`)

Last Sips

- optimize only if
  - your code is working
  - you have tests for your code
  - you know what should be optimized (profiling)
  - the optimization is worth it (work, time, readability)
- do not perform useless micro-optimizations (e.g. `a << 1`)
- do not optimize prematurely

Last Sips

- optimize only if
  - your code is working
  - you have tests for your code
  - you know what should be optimized (profiling)
  - the optimization is worth it (work, time, readability)
- do not perform useless micro-optimizations (e.g. `a << 1`)
- do not optimize prematurely
- always perform benchmarks (avoid *pessimization*)

Last Sips

- optimize only if
  - your code is working
  - you have tests for your code
  - you know what should be optimized (profiling)
  - the optimization is worth it (work, time, readability)
- do not perform useless micro-optimizations (e.g. `a << 1`)
- do not optimize prematurely
- always perform benchmarks (avoid *pessimization*)
- know your language, compiler, OS, architecture, ...

Last Sips

- optimize only if
  - your code is working
  - you have tests for your code
  - you know what should be optimized (profiling)
  - the optimization is worth it (work, time, readability)
- do not perform useless micro-optimizations (e.g. `a << 1`)
- do not optimize prematurely
- always perform benchmarks (avoid *pessimization*)
- know your language, compiler, OS, architecture, ...
- practice makes perfect

Discussion

Standard Containers in C++

Sequence Containers

- `vector`
- `list` (http://tiny.cc/vector-list-deque)
- `deque`

Container Adaptors

- `stack`
- `queue`
- `priority_queue`

Associativity Containers

- `set, multiset`
- `map, multimap`

Unordered Associativity Containers (TR1, C++11)

- `unordered_set, unordered_multiset`
- `unordered_map, unordered_multimap`