Typegrind

Type preserving heap profiler for C++

Zsolt Parragi[1,2]   Zoltan Porkolab[1,3]

[1]Faculty of Informatics
Eotvos Lorand University

[2]NNG Llc

[3]Ericsson Hungary Ltd

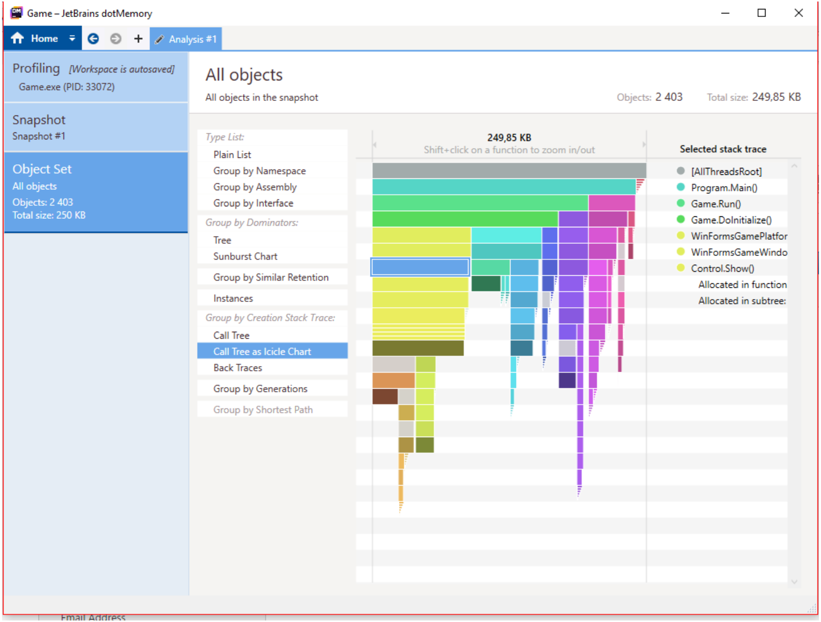
CppNow, 2016

Table of Contents

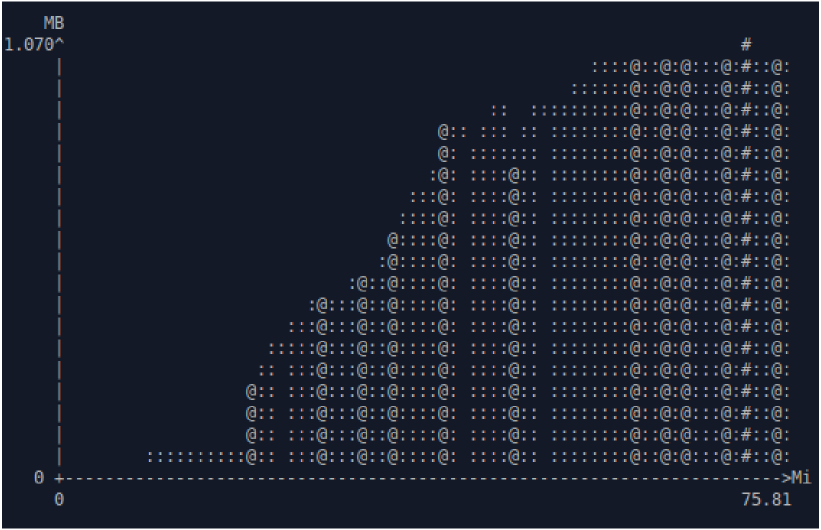
Motivation

- Better understanding
- Bughunting
- Improving performance
- Existing tools
  - DotMemory
  - JProfiler
  - Valgrind massif
  - Visual Studio 2015

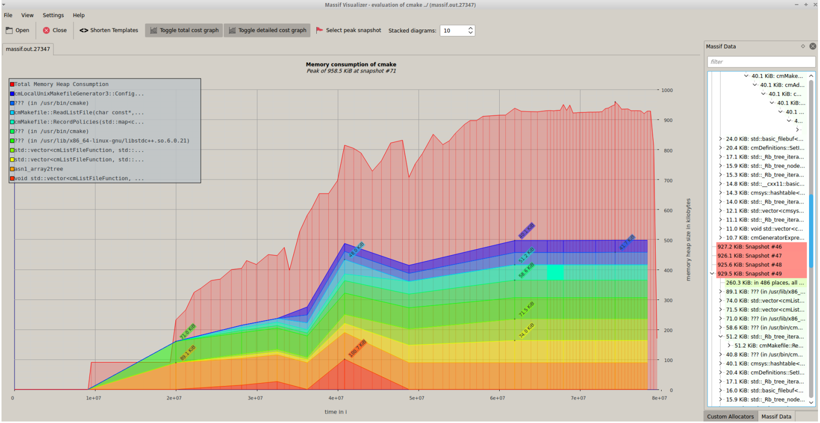
Existing tools - DotMemory



Existing tools - massif



Existing tools - massif



A simple example

```
int main(int argv, char** argc)
{
  int* P = new int(5);
  // ...
  delete P;
}
```

A simple example

```
void* malloc( std::size_t size );
void* operator new(std::size_t count) { /* ... */ }
void free( void* ptr );
void operator delete(void* ptr) noexcept { /* ... */ }

int main(int argv, char** argc)
{
  int* P = new int(5);
  // ...
  delete P;
}
```

Monitoring malloc/free

- Requires no program modifications
- Easy to use
- Fast
- Unusable with internal memory pools

Disabling internal memory pools

- Minimal source modification
- Still no type information

Extracting type information?

- Source code
- Debug symbols
- Disabled inlining

At least in theory...

Table of Contents

Macro new and delete

```
#define new NEW_MARKER() * new
#define delete DELETE_HANDLER() =
struct NEW_MARKER{};
struct DELETE_MARKER{};

template<typename T>
operator *(NEW_MARKER m, T* ptr) { ... }

template<typename T>
operator =(DELETE_HANDLER m, T* ptr) {  //...
  delete ptr;
}
```

Problem 1: arrays

```
#define delete DELETE_HANDLER() =

// DELETE_HANDLER() = [] someArray;
delete[] someArray;
```

Problem 1: arrays

```
#define new NEW_MARKER() * new

template<typename T>
operator *(NEW_MARKER m, T* ptr) {
 size_t typeSize = sizeof(T);
 size_t arraySize = ???
}
```

Problem 2: calling operator new

```
#define new NEW_MARKER() * new

// char* arr = reinterpret_cast<char*>(
//                  ::operator NEW_MARKER() * new(512)
//              );
char* arr = reinterpret_cast<char*>(::operator new(512));
```

Problem 3: standard containers

```
std::vector<int> v(512);
v.resize(1024);
```

Problem 3: standard containers

```
std::vector< int, std::allocator<int> > v(512);
v.resize(1024);
```

Problem 3: standard containers

```
std::vector< int, std::allocator<int> > v(512);
v.resize(1024);

// ... somewhere in vector ...

allocator.allocate(...);
allocator.deallocate(...);
```

Problem 4: placement new

```
#define new NEW_MARKER() * new

char buffer[512];
//string *p = NEW_MARKER() * new (buffer) string("hello");
string *p = new (buffer) string("hello");
```

Table of Contents

Table of Contents

Basic idea

```
typedef int myint;
myint* a = new myint[3];

myint* a = TYPEGRIND_LOG_NEW_ARRAY(
            "example.cpp:7",
            "myint",
            "int",
            sizeof(int),
            3,
            (new myint[3])
          );
```

Using clang tooling

- Easy to use AST matchers and rewriters
- Works with most C++ programs
- Great test infrastructure
- Possible tightly integrated version with clang

Problem 1: conditional macros

```
#if defined(__clang__)
// ....
#elif defined(__GNUC__) || defined(__GNUG__)
// ....
#elif defined(_MSC_VER)
// ....
#endif
```

Problem 2: generic macros

```
#define FACTORY_MACRO(TYPE, NAME) TYPE NAME = new TYPE;

// ...

FACTORY_MACRO(int, justAnInt);
```

Problem 3: templates

```
template<typename T>
void functionWhichCreates() {
  T* t = new T;
}
```

Problem 3: templates

```
template<typename T>
void functionWhichCreates() {
  T* t = TYPEGRIND_LOG_NEW(
           "example.cpp:7",
           "??",
           "??",
           sizeof(T),
           (new T)
         );
}
```

Problem 3: templates

```
template<typename T>
void functionWhichCreates() {
  T* t = TYPEGRIND_LOG_NEW(
          "example.cpp:7",
          demangle(typeid(T).name()),
          demangle(typeid(T).name()),
          sizeof(T),
          (new T)
        );
}
```

Problem 3: templates

```
template<typename T>
struct typegrind_name {
  static const char* name;
};
template<typename T>
void functionWhichCreates() {
  T* t = TYPEGRIND_LOG_NEW(
            "example.cpp:7",
            typegrind_name<T>::name,
            typegrind_name<T>::name,
            sizeof(T),
            (new T)
          );
}
struct S{ struct I{}; };
template<>
const char* typegrind_name<S::I>::name = "int";
```

Problem 3: templates

```
template<typename T, int>
struct typegrind_specific_name {
  static const char* name;
};
template<typename T>
void functionWhichCreates() {
  T* t = new TYPEGRIND_LOG_NEW(
            "example.cpp:7",
            typegrind_canonical_name<T>::name,
            typegrind_specific_name<T, 42>::name,
            sizeof(T),
            (T)
          );
}
struct S{ struct I{}; };
template<> const char*
typegrind_specific_name<S::I, 42>::name = "my_int";
```

Problem 4: allocators

```
void vector<T>::resize (size_type n, ...)
{
  TYPEGRIND_LOG_METHOD_ENTER(
    "void vector<T>::resize,
    "configurable_name",
    TYPEGRIND_OWNERSHIP_FLAG
  );
}
```

Problem 4 (b): initializer lists

```
struct Parent {
  Parent() { int i = new int(3); }
};
struct Child : public Parent {
  Child() {
    TYPEGRIND_LOG_METHOD_ENTER(
      ...,
      TYPEGRIND_OWNERSHIP_FLAG
    );
  }
};
```

Problem 4 (b): initializer lists

```
struct Parent :
    private TYPEGRIND_INHERITANCE_MARKER<Parent>
{
  Parent() { int i = new int(3); }
};
struct Child :
    private TYPEGRIND_INHERITANCE_MARKER<Child>,
    public Parent {
  Child() {
    TYPEGRIND_LOG_METHOD_ENTER(
      ...,
      TYPEGRIND_OWNERSHIP_FLAG
    );
  }
};
```

Problem 5: file locations

- We have to modify external (standard) libraries
- Double by copying them to a different locations
- Based on a mapping configuration

Table of Contents

Logging API

- Source-to-source compiler decorates using TYPEGRIND_* named macros
  - TYPEGRING_LOG_NEW
  - TYPEGRIND_LOG_NEW_ARRAY
  - TYPEGRIND_LOG_METHOD_ENTER
  - etc
- Loggers define them to something meaningful

Logging API limitations

- Implement them as libraries
- Keep as much as possible inline
- Do not include anything in their public headers

```
// in some_logger.h
#include <iostream>
#define TYPEGRIND_SOME_LOG(...) std::cout << "logging";
```

Logging API limitations

- Implement them as libraries
- Keep as much as possible inline
- Do not include anything in their public headers

```
// in some_logger.h
#define TYPEGRIND_SOME_LOG(...) my_logging_fw("logging");
my_logging_fw(const char*);

// in some_logger.cpp
#include <iostream>

my_logging_fw(const char* str) { std::cout << str; }
```

Logging API limitations

- Be ready for early (before main) allocations

```
struct S {
  S() { new int(5); }
};
S s;
int main() {
 //
}
```

Logging API limitations

- There is no standard way to pass configuration (e.g. as a command line parameter)
- No configuration (e.g. output filename is based on current timestamp)
- Compile time configuration
- Configuration singleton with a config file

Including loggers

- With PCH: add it to the precompiled header
- Without PCH: the source-to-source compiler can prepend includes

Basic example loggers

- NOP
- DemoCout
- CSV
- BufferedBinary

Table of Contents

Goals

- Minimal (speed) overhead
- Minimal output
- Threadsafe

Implementation concept

- Output is separated into two categories (files)
  - String table: pointer - string mappings
  - Run table: array with numerical data and string pointers
- Thread local buffers resulting in two files per thread

String table format

- Text based (CSV)
- Two columns: pointer address and string content
- Contains every string used by the logger

String table generation

- Key problem: print strings just once (per thread)
- Requires quick decision
- Basic implementation: Using a hash table
- More complex:
  - Requires changes in the source transformation
  - Using static initialization

Running table format

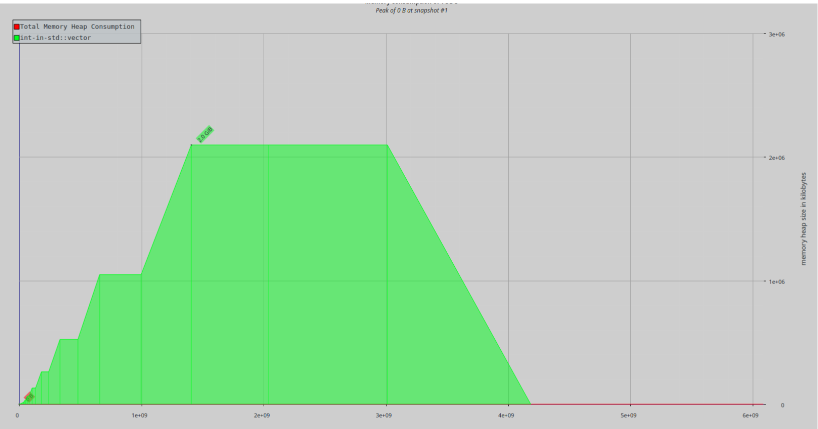
- Binary format, fixed size struct
- Columns:
  - Timestamp
  - Location info (pointer)
  - Record type (allocation, free, method marker, ...)
  - Target pointer
  - Type name (pointer)
  - Canonical type name (pointer)
  - Size
  - Owner's method name
  - Owner's custom name
  - Owner's flags

Table of Contents

Loading the logs

- Requires some post processing
  - Pairing allocations and frees with the same addresses
  - Calculating object lifetime
  - Linearizing multithreaded results
- Can be converted into different output formats
  - EJDB: embedded database for C++, usable from many other languages
  - Massif's output format: with type names instead of callstacks, can be used with massif visualisers
  - CSV: easily readable by anything

Generating graphs



Generating graphs

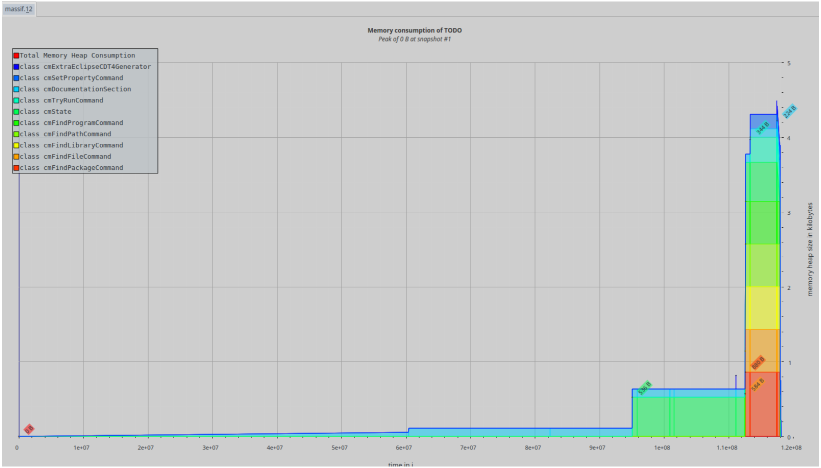


Table of Contents

Configuration file

```
{
    "prepend_include": [
        "typegrind/logger/buffered_binary.h"
    ],
    "mapping": {
      ".": "../CMake-instrumented/",
      "/usr/include": "../CMake-instrumented/usr-include"
    },
    "watch": [
      {
        "regex": "std::vector.*",
        "name": "std::vector",
        "flags": 1
      }
    ]
}
```

Processing files

```ruby
require 'json'
content = File.read('compile_commands.json')
JSON.parse(content).each do |entry|
  puts `clang-typegrind #{entry['file']} 2>&1`
end
```

```
ruby process.rb | tee typegrind.log
```

Table of Contents

Table of Contents

Handling placement new

```
return new (Context) CXXTypeidExpr(
    TypeInfoType.withConst(),
    Operand,
    SourceRange(TypeidLoc, RParenLoc)
  );
```

Improved ownership logging

```
std::vector<std::string> s;
s.push_back("hello");
s.push_back("world");
s[0] = "another string";
```

Better preprocessor handling

```
// some_header.h
#ifdef X
#define TYPE int
#else
#define TYPE double

// some_other_file.cpp

#define X
#include "some_header.h"
// ...
#undef X
#include "some_header.h"
```

Understanding type hierarchy

```
class Parent {};
class Child1: public Parent {};
class Child2: public Parent {};
```

More tools / loggers

- Graph generation
- Real time (network based) logging
- Interactive UI
- Method marker based measurements
  - Function execution time
  - No-leak functions

Table of Contents