# The Current State of (Free) Static Analysis

# Jason Turner

- http://github.com/lefticus/presentations
- http://cppcast.com
- http://chaiscript.com - *Open Session Thursday (tomorrow) morning at 8:00-8:45*
- http://cppbestpractices.com
- Independent Contractor

# Static Analysis

https://en.wikipedia.org/wiki/Static_program_analysis

> *Static program analysis is the analysis of computer software that is performed without actually executing programs...*

- We will be focusing on
  - source code analysis (as opposed to object code)
  - tools that are freely available
  - issues I've seen in the wild
- Technically static analysis includes compiler warnings
  - modern compiler warnings are very sophisticated, and include things that used to be considered 'analysis' (*ex: analysis of printf formatting issues occurs on GCC with no extra warnings enabled*)
  - compiler warnings will only be brought up if they are unique to a particular compiler

# Tools

- cppcheck
- clang's analyzer
- msvc++ analyze
- coverity's 'scan' (free for open source projects)

## Honorable Mention

- metrix++
- copy-paste detectors

# Just Scratching The Surface

- There are dozens (hundreds?) of code analysis tools available.
- cppcheck has 320 checks
- clang has 56 checks
- MSVC has ~286 checks

# Let's Play: Spot the Bug

# assert

```
#include <cassert>

bool do_something(int &i)
{
  ++i;
  return i < 10;
}

int main()
{
  int i = 0;


  assert(do_something(i));

}
```

# assert

```
#include <cassert>

bool do_something(int &i)
{
  ++i;
  return i < 10;
}

int main()
{
  int i = 0;

  // what happens in a release build?
  assert(do_something(i));
  // warning: i is initialized but unused
}
```

# assert

```
#include <cassert>

bool do_something(int &i)
{
  ++i;
  return i < 10;
}

int main()
{
  int i = 0;

  // what happens in a release build?
  assert(do_something(i));
  if (i > 2) { /* ... */ } // no warning
}
```

# assert - Conclusions

- No tool tested complained as long as `i` was used.
- clang and cppcheck say they are supposed to catch assert statements with side effects

# branching.1

```
bool test_value_1() { return true; }

bool do_something()
{
  if (test_value_1()) {

    return true;
  } else {

    return true;
  }
}

int main()
{
  do_something();
}
```

# branching.1

```
bool test_value_1() { return true; }

bool do_something()
{
  if (test_value_1()) {

    return true;
  } else {
    // pointless duplication
    return true;
  }
}

int main()
{
  do_something();
}
```

# branching.2

```
bool test_value_1() { return true; }
bool test_value_2() { return true; }

bool do_something()
{
  if (test_value_1()) {

    if (test_value_2()) {
      return true;
    }

    return true;
  }

  return false;
}

int main()
{
  do_something();
}
```

# branching.2

```
bool test_value_1() { return true; }
bool test_value_2() { return true; }

bool do_something()
{
  if (test_value_1()) {
    // this test is useless
    if (test_value_2()) {
      return true;
    }

    return true;
  }

  return false;
}

int main()
{
  do_something();
}
```

# branching - Duplicate Branch - Conclusions

- Only cppcheck caught either case
- Surprisingly, this more complicated case was caught by Coverity Scan in ChaiScript

```
if (rt.bare_equal(boxed_type))
{
  if (lt.bare_equal(boxed_pod_type))
  {
    return true;
  }

  return true;
}
```

It is unknown why our simplified case was not caught by coverity.

# precision.1

```
// Assume modern 64bit platform
#include <vector>
#include <iostream>

int main()
{
  std::vector<int> v;


  for (unsigned I = 0; I < v.size(); ++I)
  {
    std::cout << v[I] << '\n';
  }
}
```

# precision.1

```
// Assume modern 64bit platform
#include <vector>
#include <iostream>

int main()
{
  std::vector<int> v;

  // this is bad because it limits the loop to 2^32 elements
  for (unsigned l = 0; l < v.size(); ++l)
  {
    std::cout << v[l] << '\n';
  }
}
```

# precision.2

```
// Assume modern 64bit platform
#include <vector>
#include <iostream>

int main()
{
  std::vector<int> v;


  for (long l = 0; l < v.size(); ++l)
  {
    std::cout << v[l] << '\n';
  }
}
```

# precision.2

```
// Assume modern 64bit platform
#include <vector>
#include <iostream>

int main()
{
  std::vector<int> v;

  // this is bad because it limits the loop to 2^63 elements
  for (long l = 0; l < v.size(); ++l)
  {
    std::cout << v[l] << '\n';
  }
}
```

# precision.3

```
// Assume modern 64bit platform
#include <vector>
#include <iostream>

int main()
{
  std::vector<int> v;


  for (int I = 0; I < v.size(); ++I)
  {
    std::cout << v[I] << '\n';
  }
}
```

# precision.3

```
// Assume modern 64bit platform
#include <vector>
#include <iostream>

int main()
{
  std::vector<int> v;

  // this is bad because it limits the loop to 2^31 elements
  for (int I = 0; I < v.size(); ++I)
  {
    std::cout << v[I] << '\n';
  }
}
```

# precision.3

```
// Assume modern 64bit platform
#include <vector>
#include <iostream>

int main()
{
  std::vector<int> v;

  // this is bad because it limits the loop to 2^31 elements
  for (int i = 0; i < v.size(); ++i) // what else is odd?
  {
    std::cout << v[i] << '\n';
  }
}
```

# precision.3

```
// Assume modern 64bit platform
#include <vector>
#include <iostream>

int main()
{
  std::vector<int> v;

  // this is bad because it limits the loop to 2^31 elements
  for (int I = 0; I < v.size(); ++I) // Empty vector
  {
    std::cout << v[I] << '\n';
  }
}
```

# precision.3

```
// Assume modern 64bit platform
#include <vector>
#include <iostream>

int main()
{
  std::vector<int> v;

  // this is bad because it limits the loop to 2^31 elements
  for (int I = 0; I < v.size(); ++I) // Anything else odd?
  {
    std::cout << v[I] << '\n';
  }
}
```

# precision.3

```
// Assume modern 64bit platform
#include <vector>
#include <iostream>

int main()
{
  std::vector<int> v;

  // this is bad because it limits the loop to 2^31 elements
  for (int I = 0; I < v.size(); ++I) // We should be using a range-based loop
  {
    std::cout << v[I] << '\n';
  }
}
```

# precision - Loss of Sign and Size - Conclusion

- clang and MSVC warn about sign comparisons
- No tool catches both

Bonus

- cppcheck warns us we are iterating over an empty vector
- clang-tidy's modernize checks warn us the loop should be a range based loop

# precision - Real World

Adapted from http://googleresearch.blogspot.no/2006/06/extra-extra-read-all-about-it-nearly.html

```
#include <vector>

uint64_t binarySearch(const std::vector <int64_t> &v, int64_t key) {
  int low = 0;
  int high = v.size() - 1; // loss of precision

  while (low <= high) {
    int mid = (low + high) / 2;
    int64_t midVal = v[mid];   // What happens with > 2B objects?

    if (midVal < key) {
      low = mid + 1;
    } else if (midVal > key) {
      high = mid - 1;
    } else {
      return mid; // key found
    }
  }

  return -(low + 1); // key not found. What if negative?
}

int main()
{
  return binarySearch(std::vector<int64_t>{1,2,3,4,5}, 2);
}
```

# deadcode

```
#include <system_error>

enum Enum
{
  Value1,
  Value2,
  Value3
};

int go(Enum t_enum) {
  switch (t_enum) {
  case Enum::Value1:
    return 0;
  case Enum::Value2:
    return 1;
  default:
    throw std::runtime_error("unknown value");
  }
  throw std::runtime_error("unknown value");
}

int main()
{
  go(Enum::Value3);
}
```

# deadcode

```
#include <system_error>

enum Enum
{
  Value1,
  Value2,
  Value3
};

int go(Enum t_enum) {
  switch (t_enum) {
  case Enum::Value1:
    return 0;
  case Enum::Value2:
    return 1;
  default:
    throw std::runtime_error("unknown value");
  }
  throw std::runtime_error("unknown value");// this code is unreachable
}

int main()
{
  go(Enum::Value3);
}
```

# deadcode - Conclusions

- MSVC warns about this *in the IDE*
- clang warns with `-Weverything` / clang-tidy warns

Bonus

- coverity scan notes that the `throw` is unhandled in `main`

# nullptr.1

```
void do_something()
{
  int *i = nullptr;

  *i = 5;
}

int main()
{
  null_dereference_1();
}
```

# nullptr.1

```
void do_something()
{
  int *i = nullptr;
  // dereferencing obviously null value
  *i = 5;
}

int main()
{
  null_dereference_1();
}
```

# nullptr.2

```
int *get_i() {
  return nullptr;
}

void do_something()
{
  int *i = get_i();

  *i = 5;
}

int main()
{
  null_dereference_2();
}
```

# nullptr.2

```
int *get_i() {
  return nullptr;
}

void do_something()
{
  int *i = get_i();
  // Indirect dereference
  *i = 5;
}

int main()
{
  do_something();
}
```

# nullptr.3

```
#include <memory>

void do_something()
{
  std::shared_ptr<int> i;
  // dereferencing obviously null value
  *i = 5;
}

int main()
{
  do_something();
}
```

# nullptr - Null Dereferences - Conclusions

- Everyone caught the obvious one
- Only cppcheck could catch the indirect nullptr dereference
- No tools could catch the smart pointer version

# array

```
int main()
{
  int a[5];
  return a[5];
}
```

# array

```
int main()
{
  int a[5];
  return a[5]; // indexing past end of array
}
```

# std::array

```
#include <array>

int main()
{
  std::array<int, 5> a;
  return a[5];
}
```

# std::array

```
#include <array>

int main()
{
  std::array<int, 5> a;
  return a[5]; // indexing past end of array
}
```

# array - Conclusions

- The c-style array is caught by all tools
- `std::array` issue is only caught by cppcheck

Bonus

- MSVC notices that the c-style array is used uninitialized.

# lambdas.1

```
#include <functional>
#include <iostream>

std::function<void ()> do_something()
{
  int some_value = 1;


  return [&some_value](){ std::cout << some_value << '\n'; };
}

int main()
{
  const auto f = do_something();
  f();
}
```

# lambdas.1

```
#include <functional>
#include <iostream>

std::function<void ()> do_something()
{
  int some_value = 1;

  // some_value won't exist when this function is returned
  return [&some_value](){ std::cout << some_value << '\n'; };
}

int main()
{
  const auto f = do_something();
  f();
}
```

# lambdas.2

```
#include <functional>
#include <iostream>

std::function<void ()> do_something()
{
  int some_value;


  return [some_value](){ std::cout << some_value << '\n'; };
}

int main()
{
  const auto f = do_something();
  f();
}
```

# lambdas.2

```
#include <functional>
#include <iostream>

std::function<void ()> do_something()
{
  int some_value;

  // some_value is used uninitialized
  return [some_value](){ std::cout << some_value << '\n'; };
}

int main()
{
  const auto f = do_something();
  f();
}
```

# lambdas.3

```
#include <functional>
#include <iostream>

std::function<void ()> do_something()
{
  int some_value;


  return [&some_value](){ std::cout << some_value << '\n'; };
}

int main()
{
  const auto f = do_something();
  f();
}
```

# lambdas.3

```
#include <functional>
#include <iostream>

std::function<void ()> do_something()
{
  int some_value;

  // uninitialized and captured local by reference
  return [&some_value](){ std::cout << some_value << '\n'; };
}

int main()
{
  const auto f = do_something();
  f();
}
```

# lambdas - Capture Local - Conclusions

- cppcheck, clang, and coverity got confused as to whether the variable was initialized if captured by reference
- msvc caught nothing

**No tool warned about the actual capture local by reference and return**

# &&

```
#include <utility>


struct Object {
  void do_something() {}
};

void take(Object &&) { }

void do()
{
  Object o;
  take(std::move(o));
  o.do_something();
}

int main()
{
  do();
}
```

# &&

```
#include <utility>


struct Object {
  void do_something() {}
};

void take(Object &&) { }

void do()
{
  Object o;
  take(std::move(o));
  o.do_something(); // use of local after move
}

int main()
{
  do();
}
```

# && - Use After Move - Conclusions

- No tool commented on this problem at all.

Bonus

- cppcheck points out that *technically* `Object::do_something` can be static

# bonus slide - iterator mismatch

```
#include <vector>

int main()
{
  std::vector<int> v;
  std::vector<int> v2(2,4);
  std::vector<int> v3(3,4);

  v.insert(v.begin(), v2.begin(), v3.end());
}
```

# bonus slide - iterator mismatch

```
#include <vector>

int main()
{
  std::vector<int> v;
  std::vector<int> v2(2,4);
  std::vector<int> v3(3,4);

  v.insert(v.begin(), v2.begin(), v3.end()); // coverity catches this
}
```

# bonus slide - pointer invalidation

```
#include <memory>

int main()
{
  auto s = std::make_shared<int>(1);

  int *p = s.get();
  *p = 1;

  s = std::make_shared<int>(2);
  *p = 5;
}
```

# bonus slide - pointer invalidation

```
#include <memory>

int main()
{
  auto s = std::make_shared<int>(1);

  int *p = s.get();
  *p = 1;

  s = std::make_shared<int>(2); // nobody catches this yet
  *p = 5;
}
```

# Honorable Mention - metrix++

- analyzes code for various metrics
- can calculate cyclomatic complexity - a measure of code's complexity - and warn on overly complex sections of code

```
static Common_Types get_common_type(const Boxed_Value &t_bv)
{
  const Type_Info &inp_ = t_bv.get_type_info();

  if (inp_ == typeid(int)) {
    return get_common_type(sizeof(int), true);
  } else if (inp_ == typeid(double)) {
    return Common_Types::t_double;
  } else if (inp_ == typeid(long double)) {
    return Common_Types::t_long_double;
  } else if (inp_ == typeid(float)) {
    return Common_Types::t_float;
  } else if (inp_ == typeid(char)) {
    return get_common_type(sizeof(char), std::is_signed<char>::value);
  } else if (inp_ == typeid(unsigned char)) {
    return get_common_type(sizeof(unsigned char), false);
  } else if (inp_ == typeid(unsigned int)) {
    return get_common_type(sizeof(unsigned int), false);
  } else if (inp_ == typeid(long)) {
    return get_common_type(sizeof(long), true);
  } else if (inp_ == typeid(unsigned long)) {
    return get_common_type(sizeof(unsigned long), false);
/* etc... */
```

# Honorable Mention - Copy-Paste-Detectors

- From the PMD project can detect duplicated code throughout your code base

```
} else {
  if (*s == '\\') {
    if (is_escaped) {
      match.push_back('\\');
      is_escaped = false;
    } else {
      is_escaped = true;
    }
  } else {
    if (is_escaped) {
      switch (*s) {
        case ('b') : match.push_back('\b'); break;
        case ('f') : match.push_back('\f'); break;
        case ('n') : match.push_back('\n'); break;
        case ('r') : match.push_back('\r'); break;
        case ('t') : match.push_back('\t'); break;
        case ('\"') : match.push_back('\"'); break;
        case ('\'') : match.push_back('\''); break;
        case ('$') : match.push_back('$'); break;
```

# Downsides - False Positives

```
#include <vector>

template<typename T>
T add(T t, T u)
{
  return t + u;
}

template<typename ... T>
std::vector<int> add_values(int value, T ... t)
{
  return {add(t, value)...};
}

int main()
{
  add_values(4);
}
```

# Downsides - False Positives

```
#include <vector>

template<typename T>
T add(T t, T u)
{
  return t + u;
}

template<typename ... T>
std::vector<int> add_values(int value, T ... t)
{
  return {add(t, value)...}; // msvc complains `value` is unused in 0th case
}

int main()
{
  add_values(4);
}
```

# downsides - false sense of security

ChaiScript passed all of these analyzers (plus some free trials of commercials ones).

And then we discovered the sanitizers.

And a few more issues were found.

And then we discovered fuzzy testing

And MANY more issues were found.

# Real World Cleanup

```
bool Switch() {
  bool retval = false;

  size_t prev_stack_top = m_match_stack.size();

  if (Keyword("switch")) {
    retval = true;

    if (!Char('(')) {
      throw exception::eval_error("Incomplete 'switch' expression", File_Position(m_line, m_col), *m_filename);
    /* snip */
    while (Eol()) {}

    if (Char('{')) {
      retval = true;
      /* snip */
    }

    build_match(AST_NodePtr(new eval::Switch_AST_Node()), prev_stack_top);


  }

  return retval;
}
```

# Real World Cleanup

```
bool Switch() {


  size_t prev_stack_top = m_match_stack.size();

  if (Keyword("switch")) {


    if (!Char('(')) {
      throw exception::eval_error("Incomplete 'switch' expression", File_Position(m_line, m_col), *m_filename);
    /* snip */
    while (Eol()) {}

    if (Char('{')) {

      /* snip */
    }

    build_match(AST_NodePtr(new eval::Switch_AST_Node()), prev_stack_top);

    return true;
  } else {
    return false;
  }
}
```

# Conclusion

- C style issues are largely a "solved problem"
- But there are still many ways to abuse current best practices
- Modern C++ and C++ >= 11 analysis checking still has a long way to go
- You must use a combination of compilers / analyzers
- C++ Core Guidelines will almost certainly start steering the course of analysis soon

# Actions

- Consider `-Werror -Weverything` (with selective disables) on clang
- Consider `/W3 /WX /analyze` (with selective disables) on MSVC
- Consider building your own analysis with libclang or adding to cppcheck
- Consider adding to https://github.com/lefticus/AnalysisTestSuite where these tests and others live

# Jason Turner

- http://github.com/lefticus
- http://chaiscript.com - *Open Session Thursday (tomorrow) morning at 9:00-9:30*
- http://cppcast.com
- http://cppbestpractices.com
- Independent Contractor - *Always looking for new clients*