Static Analysis and C++

More Than Lint

Neil MacIntosh
neilmac@microsoft.com

```
struct Thing {
    int     someInt;
    int     anotherInt;
};

Result InitializeThing(Thing* thing) {
    // ... local declarations ...

    // validate parameters
    if (thing == nullptr)
        return Result::InvalidParameter;

    // initialize the structure to safe defaults
    memset(&thing, 0, sizeof(Thing));

    // ... do some other setup work...

    return Result::Success;
}
```

```
thing.cpp(15) : warning CXXXX: Overflow using expression '(void *)(&thing)'
Buffer accessed is thing
Buffer is of length 4 bytes [size of variable]
Accessing 8 bytes starting at byte offset 0
```

```
====> memset(thing, 0, sizeof(Thing));
```

Simple bug, hard to track down without help

The “lint” value proposition:

Find defects during construction: save time and money

- reduce the cost of locating and fixing them
- reduce their impact on your customers

- So successful, many “lint” checks migrated into compilers as warnings
  - so always compile with -Wall

- Lots of free, open source and commercial offerings for static analysis of C++ source

- If you’re not seeing the value, complain! (or...contribute!)

```
enum class States { Started, Stopped, Waiting, ... };

void StateMachine::OnStop() {
    // ... see if it's ok to stop ..

    m_current = States::Started;

    // when we're stopped, we need to go wait for restart
    if (m_current == States::Stopped) {
        WaitForStart();
    }
}
```

Scaling the “lint” value proposition

- Need fast turnaround while doing edit-build-debug cycle
  - You want to be inside the developer’s “inner loop”
- Value must be obvious and actionable
  - False positives are tolerable but cannot be overwhelming
  - True positives must be comprehensible and real defects
- Broaden the search: not just safety/reliability bugs
  - API enforcement is a valuable use (Secure CRT, deprecated Crypto....)
  - Performance is increasingly interesting as a target

Fast turnaround

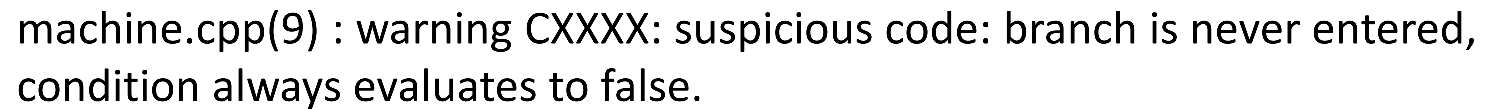
- Tools must run quickly: slower than compiler ok, but not too much
  - Running asynchronously with build helps a lot
  - Parallel execution helps a lot
  - Incremental build\analysis helps a lot
  - Share common work – reuse construction costs

- Simple checks are best
  - syntactic
  - simple flow-sensitive

```
struct Thing {
    int     someInt;
    int     anotherInt;
};

Result InitializeThing(Thing* thing){
    // ... local declarations ...

    // validate parameters
    if (thing == nullptr) {
        return Result::InvalidParameter;
    }

    // initialize the structure to safe defaults
    memset(&thing, 0, sizeof(Thing));

    // ... do some other setup work...

    return Result::Success;
}
```

```
AST_FUNCTIONCALL "memset" "void*(void*,int,unsigned int)"
  AST_ARGUMENTS
    AST_CAST "void*"
      AST_ADDRESS "struct Thing**"
        AST_SYMBOL “Thing" "struct Thing *"
    AST_CONSTANT =0 "int"
    AST_SIZEOFTYPE =8 "unsigned int"
```

```cpp
enum class States { Started, Stopped, Waiting, ... };

void StateMachine::OnStop() {
    // ... see if it's ok to stop ..

    m_current = States::Started;

    // when we're stopped, we need to go wait for restart
    if (m_current == States::Stopped) {
        WaitForStart();
    }
}
```



Make defects obvious and actionable

- Use heuristics to restrict false positives.
  - Success vs. failure paths.
  - Do all roads lead to Rome?
  - Context matters!

- Clear warning messages that include relevant detail

- Good diagnostic traces help

- Lots of scope for IDE and tool integration
  - Automated fixups

```cpp
// Set of bitflags that control system features
enum SystemLevels {
    // ...
};

const unsigned int g_flags = ENABLE_STANDARD_STUFF | ENABLE_OTHER_STUFF;

void Initialize() {
    if (g_flags & ENABLE_COOL_STUFF) {
        // ... enable the cool functionality ...
    }
}
```

Expression will evaluate to 0 at compile-time....by design!

Warning here will annoy users as creating “dead code” at compile-time based on bit-wise expressions is a common (and useful) configuration technique. So we are silent here.

```cpp
// bitflags that control system features
enum SystemLevels {
    ENABLE_COOL_STUFF,
    ENABLE_OTHER_STUFF,
    ENABLE_STANDARD_STUFF
};

const unsigned int g_flags = ENABLE_STANDARD_STUFF | ENABLE_OTHER_STUFF;

void Initialize() {
    if (g_flags & ENABLE_COOL_STUFF) {
        // ... enable the cool functionality ...
    }
}
```

config.cpp(11) : warning C6313: Incorrect operator: zero-valued flag cannot be tested with bitwise-and. Use an equality test to check for zero-valued flags.

Make defects obvious and actionable

- Use heuristics to restrict false positives
  - Context matters! Success vs. failure paths.
- Clear warning messages that include relevant detail
- Good diagnostic traces help
- Lots of scope for IDE and tool integration
  - Automated fixups

```cpp
// bitflags that control system features
enum SystemLevels {
    ENABLE_COOL_STUFF,
    ENABLE_OTHER_STUFF,
    ENABLE_STANDARD_STUFF
};

const unsigned int g_flags = ENABLE_STANDARD_STUFF | ENABLE_OTHER_STUFF;

void Initialize() {
    if (g_flags & ENABLE_COOL_STUFF) {
        // ... enable the cool functionality ...
    }
}
```

config.cpp(11) : warning C6313: Incorrect operator: ENABLE_COOL_STUFF has a value of zero. Testing it with bit-wise AND will always result in zero. You may have meant to check for equality instead.

Make defects obvious and actionable

- Use heuristics to restrict false positives
  - Context matters! Success vs. failure paths.
- Clear warning messages that include relevant detail
- Good diagnostic traces help
- Lots of scope for IDE and tool integration
  - Automated fixups

Scratch.cpp*

Scratch | (Global Scope)

```
1   // Scratch.cpp
2   //
3   bool cond();
4
5   void foo(int* p, int n)
6   {
7       int* q = nullptr;
8
9       if (n > 10)
10          q = p;
11
12      *p += 2;
13
14      if (n < 120)
15          *q += 12;
16  }
17
```

125 %

C6011 Dereferencing null pointer

Dereferencing NULL pointer 'q'.

Line	Explanation
7	'q' is NULL
9	Skip this branch, (assume 'n>10' is false)
14	Enter this branch, (assume 'n<120')
15	'q' is dereferenced, but may still be NULL

scratch.cpp (Line 15) Memory Safety

Warning

Error List

Entire Solution | 0 Errors | 1 Warning | 0 Messages | Build + IntelliSense | Search Error List

	Code	Description	Project	File	Line
	C6011	Dereferencing NULL pointer 'q'.	Scratch	scratch.cpp	15

What happens when we try to go further?

- No function is an island
  - Need to understand what is happening across function calls

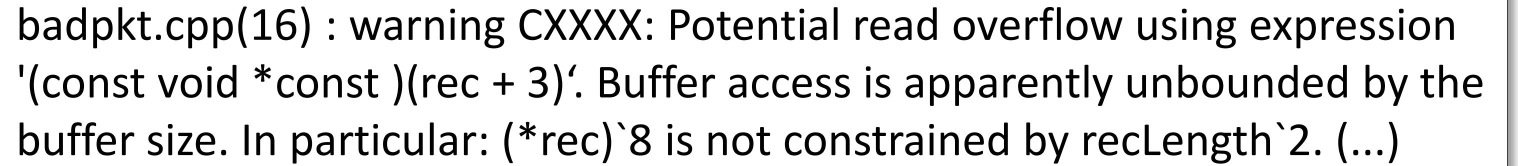
- Interprocedural analysis: analyze the whole program
  - quickly run up against hard problems (NP hard)
  - useful – but conflicts with desire to be close to the inner loop
  - best reserved for very well-specified problems

- Other approach is intraprocedural analysis: analyze each function in isolation
  - need to understand the semantics for called functions
  - infer based on heuristics, type system, source annotations

```cpp
// returns the number of bytes written to buffer or -1 on error
int MakePacket(int recLength, byte* rec, int bufSize, byte* buffer) {
    int payloadLength = -1;

    if (recLength < 3)
        return -1;

    payloadLength = (rec[1] << 8) + rec[2];
    if (bufSize < payloadLength + 3)
        return -1;

    buffer[0] = rec[0];
    buffer[1] = rec[1];
    buffer[2] = rec[2];

    if (memcpy_s(buffer + 3, bufSize - 3, rec + 3, payloadLength) != 0)
        return -1;

    return payloadLength + 3;
}
```

```
// return number of bytes read\written from socket or < 0 on error
int OS::Socket::Read(byte* buf, size_t bufSize);
int OS::Socket::Write(byte* buf, size_t bufSize);
...
int rc = OS::Socket::Read(readBuffer.data(), readBuffer.size());
if (rc <= 0) return -1;

rc = MakePacket(readBuffer.size(), readBuffer.data(),
       writeBuffer.size(), writeBuffer.data());
if (rc <= 0) return -1;

rc = OS::Socket::Write(writeBuffer.data(), rc);
if (rc <= 0)  return -1;
...
```

```
_Success_(return >= 0) int MakePacket(int recLength, _In_reads_(recLength)
byte* rec, int bufSize, _Out_writes_to_(bufSize, return) byte* buffer) {
    int payloadLength = -1;

    if (recLength < 3)
        return -1;

    payloadLength = (rec[1] << 8) + rec[2];
    if (bufSize < payloadLength + 3)
        return -1;

    buffer[0] = rec[0];
    buffer[1] = rec[1];
    buffer[2] = rec[2];

    if (memcpy_s(buffer + 3, bufSize - 3, rec + 3, payloadLength) != 0)
        return -1;

    return payloadLength + 3;
}
```



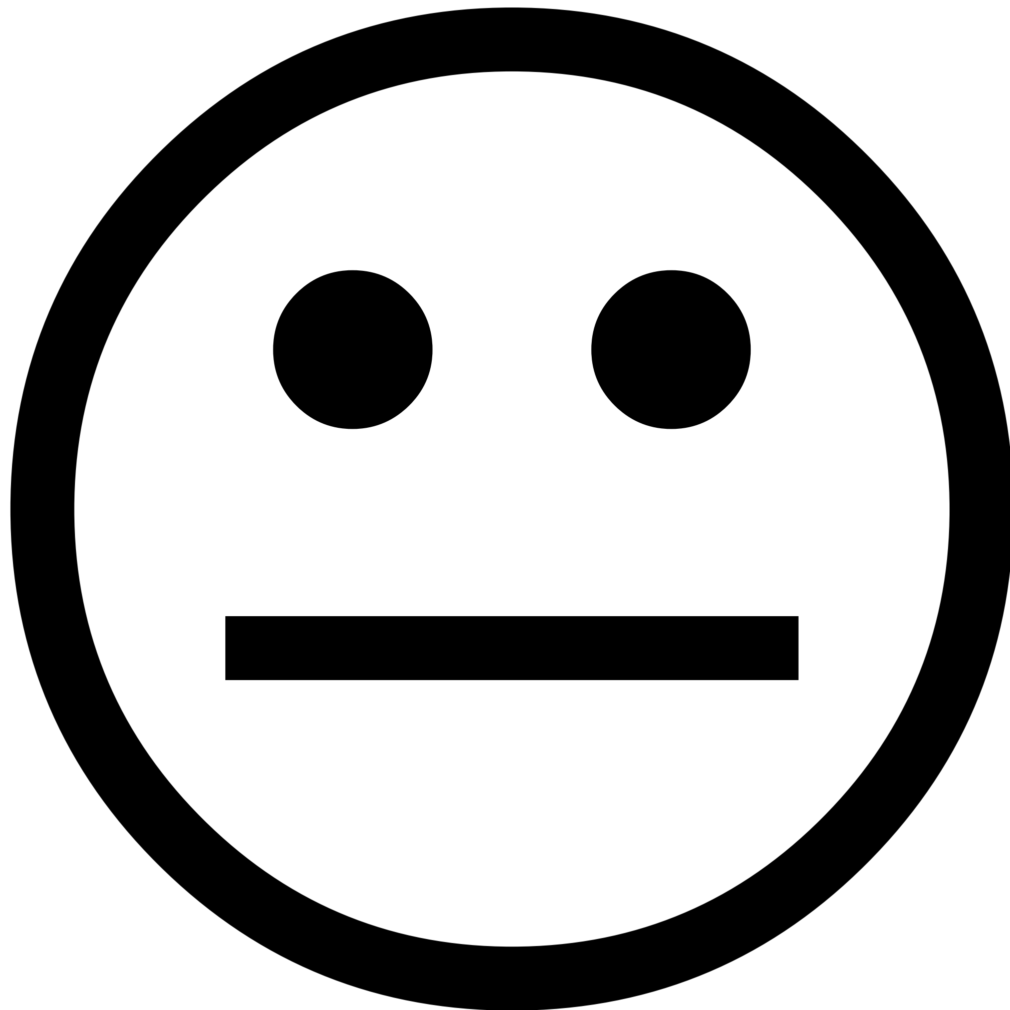
```
// return number of bytes read\written from socket or < 0 on error
int OS::Socket::Read(byte* buf, size_t bufSize);
int OS::Socket::Write(byte* buf, size_t bufSize);
...
int rc = OS::Socket::Read(readBuffer.data(), readBuffer.size());
if (rc <= 0) return -1;

rc = MakePacket(readBuffer.size(), readBuffer.data(),
       writeBuffer.size(), writeBuffer.data());
if (rc <= 0) return -1;

rc = OS::Socket::Write(writeBuffer.data(), rc);
if (rc <= 0)  return -1;
...
```

```
// return number of bytes read\written from socket or < 0 on error

_Success_(return >= 0) int OS::Socket::Read(
_Out_writes_to_(bufSize, return) byte* buf, size_t bufSize);

_Success_return >= 0) int OS::Socket::Write(_In_reads_(bufSize) byte* buf,
size_t bufSize);

...

int rc = OS::Socket::Read(readBuffer.data(), readBuffer.size());

if (rc <= 0) return -1;


rc = MakePacket(readBuffer.size(), readBuffer.data(),
       writeBuffer.size(), writeBuffer.data());

if (rc <= 0) return -1;


rc = OS::Socket::Write(writeBuffer.data(), rc);

if (rc <= 0)  return -1;

...
```

Pros and cons of annotations

- Suddenly, we can find subtle defects that previously eluded us
- Intentions are clearer and we can reduce false positives

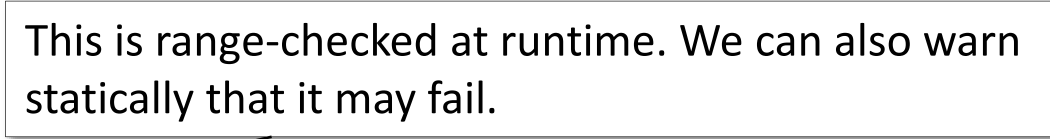
- They are viral
- They are not source code
- They are a form of language extension
- Tools must interpret them – consistency is important



```
bool MakePacket(array_view<byte> rec, array_buffer<byte> buffer) {

    if (recLength < 3)
        return -1;

    int payloadLength = (rec[1] << 8) + rec[2];
    if (buffer.length() < payloadLength + 3)
        return false;
    buffer.set_used(payloadLength + 3);

    buffer[0] = rec[0];
    buffer[1] = rec[1];
    buffer[2] = rec[2];

  if (memcpy_s(buffer.data() + 3, buffer.length() - 3, rec + 3, payloadLength) != 0)
        return false;

    return payloadLength + 3;
}
```

Can still catch error the same way.

```cpp
bool MakePacket(array_view<byte> rec, array_buffer<byte> buffer)
{
    if (rec.length() < 3)
        return false;

    int payloadLength = (rec[1] << 8) + rec[2];

    rec = rec.first(payloadLength + 3)

    if (buffer.length() < rec.length())
        return false;

    buffer.set_used(rec.length());

    copy(begin(rec), end(rec), begin(buffer));

    return true;
}
```



```
// return number of bytes read\written from socket or < 0 on error

_Success_(return >= 0) int OS::Socket::Read(
_Out_writes_to_(bufSize, return) byte* buf, size_t bufSize);

_Success_return >= 0) int OS::Socket::Write(_In_reads_(bufSize) byte* buf,
size_t bufSize);

...

int rc = OS::Socket::Read(readBuffer.data(), readBuffer.size());

if (rc <= 0) return -1;


array_buffer<byte> packet = writeBuffer;

if (!MakePacket(readBuffer, packet)) return -1;


rc = OS::Socket::Write(packet.data(), packet.used_length());

if (rc <= 0)  return -1;

...
```

Good and bad of types

- Suddenly, we can find subtle defects that previously eluded us
- Intentions are clearer and we can reduce false positives
- Now we can use the type checker to do some of the work for us!
- SOME DEFECTS ARE NO LONGER POSSIBLE – THEY DON'T COMPILE

- They are not viral – you can preserve legacy code and ABIs
- They are ~~not~~ source code
- They are **not** are a form of language extension
- You don't have to interpret them, they are precisely defined

The new “lint” ~~value proposition~~ manifesto:

Find defects during construction: save time and money

Prevent defects from being constructed.

- reduce the cost of locating and fixing them
- reduce their impact on your customers
- increase developer productivity
- add information to programs - improve them

CppCoreGuidelines

- Guidelines effort shares these principles:
  - use types for correct-by construction programs
  - use types to improve semantic clarity of programs

- Clearer intent ➔ more effective static analysis results

- Contribute back lessons from the bug patterns we have seen

Building CppCoreCheck

- Built some analyzers for coding guideline profiles
  - bounds
  - types
  - lifetime (under construction)

- bounds + types: less than 600 lines of C++ against our framework

- Will become available as a CTP around VS 2015 Update 1

Analysis Framework

- C++ framework for writing local (intraprocedural) analyses
- Portable, compiler-agnostic (once we have a parse tree)
- Uses a compiler-independent intermediate representation (types, symbols, expressions...)
- Supports reporting warnings (and potential fixes), understands suppression mechanisms, uses a single consistent output format (SARIF)
- Supports different levels of analysis
  - Simple, callback based API for getting a CFG that can be walked
  - Simple, callback based API for doing a path-sensitive walk of a CFG

Analysis Execution Engine

- For all analyses...
    - construction of shared, immutable IR
    - warning reporting, suppressions
- For flow-sensitive analyses:
    - standard sort and traversal algorithms available for flow analysis over CFG
- For path-sensitive analyses:
    - construction of shared, immutable IR
    - optimized path-sensitive traversal over CFG
    - expression evaluation, value tracking, memory model
    - constraint evaluation, path feasibility
    - loop widening, annotation (attribute) support and more...

Join us!

- Even if you just run these tools, or someone else's....
  - Give feedback and suggestions

- Help make everyone's inner loop better
  - Contribute new checks, bugfixes, test cases, ideas

- Resources:
  - http://microsoft.github.io/CodeAnalysis
  - https://github.com/sarif-standard/
  - https://msdn.microsoft.com/en-us/library/d3bbz7tz.aspx (Code Analysis in VS)
  - http://clang.llvm.org/extra/clang-tidy/