# Undefined Behavior is Not an Error

Barbara Geller & Ansel Sermersheim

CppCon - Sept 2018

# Introduction

- Prologue
- Terminology
- What is Undefined Behavior
- Why Study Undefined Behavior
- Defined Undefined Behavior
- Undefined Behavior is Not an Error
- Sequences
- Undocumented Behavior
- Undocumented Undefined Behavior
- Avoiding Undefined Behavior

# Who is CopperSpice

- Maintainers and Co-Founders of the following projects
  - CopperSpice
    - set of cross platform C++ libraries *(linux, os x, windows)*
  - DoxyPress
    - documentation program for multiple languages and outputs
  - CsString
    - support for UTF-8 and UTF-16, extensible to other encodings
  - CsSignal
    - thread aware signal / slot library
  - libGuarded
    - library which manages access to data shared between threads

# Who is CopperSpice

- Credentials
  - products leverage modern C++ idioms
  - all of our libraries and applications are open source
  - source code hosted on github
  - experience spans multiple platforms and languages
  - active video series teaching C++ with over 30 videos

  - copperspice is an expanding team
    - C++ developers world wide
    - technical documentation reviewers
    - test and quality assurance contributors
    - slack channel for team communication

# Undefined Behavior

- Prologue
  - what does undefined behavior represent for a compiler developer
    - fascinating and intriguing theoretical discussion
    - possible masters degree subject for a compiler designer
    - influences the way they write an optimizer
  - should I study compilers to learn undefined behavior
    - looking at undefined behavior the way compiler designers do may not teach a programmer enough
    - understanding undefined behavior from a compiler point of view may not be very beneficial for most programmers

# Undefined Behavior

- Prologue
  - what do C++ programmers believe about undefined behavior
    - compilers should report undefined behavior as an error
    - experienced developers can avoid bad code
    - not my responsibility
    - undefined behavior is easy to spot and simple to debug
  - what should I study about undefined behavior
    - what do I need to learn
    - can undefined behavior be avoided
    - how do you debug undefined behavior

# Undefined Behavior

- Why Terminology Matters
  - these words have precise meanings in the C++ standard
    - reference
      - invalid -- a variable which refers to a another object
      - an actual data type in C++
    - semantics
      - invalid -- brave lawn mower
      - abstraction, what operations does a data type support
    - optional
      - invalid -- parameter with a default argument
      - data type which may or may not contain a value

# Undefined Behavior

- Why Terminology Matters
  - about to acquire a meaning other than the dictionary definition
    - concepts
      - what are the fundamental concepts of C++
    - contracts
      - did you buy a put option contract on the S&P 500
    - modules
      - my code is laid out in modules since it uses classes
    - transactions
      - did the credit card charge go through
    - spaceship
      - does your “class Shuttle” have a spaceship

# Undefined Behavior

- What the Standard Meant
  - defined behavior
    - code which has a single prescribed meaning
      - `int sum = 5 + 2`
      - `printf(“Hello CppCon”)`
  - implementation defined behavior
    - code which has multiple possible meanings, compiler must consistently pick one and document this choice
    - based on the platform and compiler, not your code
      - `if ( sizeof(int) < sizeof(long) )`

# Undefined Behavior

- What the Standard Meant
  - unspecified behavior
    - code which has multiple possible meanings so the compiler is allowed to choose one at random
      - comparing string literals
        - `if("abc" == "abc")`
  - undefined behavior
    - code which has no meaning
      - dereferencing a null pointer
      - accessing an object after it has been destroyed
      - reading from uninitialized variable

# Undefined Behavior

- Not Part of the Standard
  - valid error
    - file not found, unable to open a file or socket
    - invalid argument, the value was too large or too small
    - out of memory
    - assertion failure
      - value can not be < 0
    - exception
      - container element out of bounds, using at() in std::vector
  - intended behavior
    - program ran as expected
    - successful completion
      - backup finished, doxypress generated documentation

# Undefined Behavior

- What is Undefined Behavior
  - is the result of attempting to execute source code whose behavior is not defined in the C++ standard
  - responsibility of the programmer to write code which never causes undefined behavior
  - a correct program must be free of undefined behavior

  - operations which fall under the umbrella of undefined behavior
    - C++ standard makes no guarantees how the entire program will execute at run time

# Undefined Behavior

- Example 1
  - consider a recipe which defines how to make a chocolate cake
  - the recipe defines the ingredients required

  - recipe says walnuts are optional and you decide to omit them
    - this is similar to “implementation defined”

  - assume the recipe calls for 1 tsp of salt
    - you add 1 cup of salt
    - the salt is the only issue, however the cake will be awful
    - the entire cake is “undefined behavior”, not just the salt

  - replacing milk with soy milk may work, still undefined behavior

# Undefined Behavior

- Why Study Undefined Behavior
  - when programmers are unaware of how complicated and subtle undefined behavior can be
    - rare crashes tend to be ignored
    - undefined behavior is tolerated because the code appears to work
  - web servers, web browsers, and network applications
    - must cope with unsafe input
    - can be compromised and run malicious code
    - undefined behavior can be a security vulnerability

# Undefined Behavior

- Why Study Undefined Behavior - Case Study
  - Task:  Design a meta data registration system to implement run time reflection

  - version 1
    - worked and appeared to be really good code
    - informed that our code was using undefined behavior
    - it was explained that a smart optimizer was allowed to break, remove, or creatively interpret our code
      - expect this to happen within six months
    - now what?

# Undefined Behavior

- Why Study Undefined Behavior - Case Study
  - version 2
    - DOA and never saw the light of day
  - version 3
    - uses compile time counter, method overloading, constexpr, and a recursive template with a non type template parameter
    - reviewed by a prominent committee member
  - version 4
    - currently under development
    - C++17 if constexpr(), std::any, and std:apply

# Undefined Behavior

- Defined Undefined Behavior
  - de-reference a null pointer
  - access of an element in an array which is out of bounds
  - use of an uninitialized variable
  - access to an object using a pointer of a different type
  - use of an object after it has been destroyed
  - infinite loop without side effects
  - race condition
  - shifting more than the width of an integer
  - calling a pure virtual function from a constructor or destructor
  - integer divide by zero
  - signed integer overflow, large signed number plus one
  - and many more. . .

# Undefined Behavior

- Undefined Behavior is Not an Error (UBINAE)
  - undefined behavior has a very specific meaning
  - something which is defined as an error is not undefined behavior

  - an error is well defined
    - code which produces a compile time error
      - missing semicolon, missing header file
      - method signature incompatible with the declaration
      - use of an incomplete data type
      - no matching candidate found for function call
    - code which results in a run time error
      - floating point divide by zero
      - destroying a thread without joining or detaching
      - calling myVector.at(10) on an std::vector with 5 elements

# Undefined Behavior

- EC: The Compiler of the Future
  - evil compiler
  - efficient compiler
    - optimizes your code by discarding all undefined behavior
    - reorders your code based on the fact that undefined behavior is impossible
    - does something unexpected with your undefined behavior
    - something wonderful happens in your program as a result of the undefined behavior

# Undefined Behavior

- Compiler Options
  - if optimization is off the compiler
    - does almost nothing special with your code
    - translates your code as near to literal as possible
    - undefined behavior may do what you expect so it appears your code is working as intended

# Undefined Behavior

- Compiler Options
  - normally optimization is enabled
    - unreachable code can be removed
    - compilers are not required to diagnose undefined behavior
    - code can be “inlined” and then optimized
    - may produce unexpected results when a program has undefined behavior

# Undefined Behavior

- Example 2
  - return statement is missing from a “value returning function”
    - this is undefined behavior
  - you may receive a compiler warning
  - common outcome during execution
    - may result in a crash
    - could return true every time
    - might proceed to the “next function” in the executable

```
bool isGreen() {
  m_data == “green”;
}
```

# Undefined Behavior

- Example 3
  - access an element of a container which is out of bounds
  - operator[ ] returns a reference to an element in the string
  - no test to verify index + 1 and index + 2 are valid positions
  - when the loop reaches the end of the string -- undefined behavior

```
std::string inputStr = "class std::vector<int>";
std::string className;

for (int index = 0; index < inputStr.size(); ++index) {
  if (inputStr[index+1] == ':' && inputStr[index+2] == ':') {
    // found start of class name
    index += 2;
    className = inputStr.substr(index);            // want vector<int>
  }
}
```

# Undefined Behavior

- Example 4
  - Task: Use printf() to debug a crash, output the current QString
  - bug moved around as we changed the location of printf
    - no crash when printing a const char *
    - issue only occurred using legacy version of QString
  - realized printf call was causing undefined behavior
  - found a static initialization ordering problem
    - string class had a static shared null data member
    - made the string not safe to use until the beginning of main()
  - solution was to wrap the static class member as a function local static variable inside a static method

# Undefined Behavior

- Example 5
  - relational comparison of pointers is only defined if the pointers point to members of the same object or elements of the same array
  - if `<` is replaced with `==` the standard says you must return false
  - although comparing A and B is undefined there is a rule which says undefined comparisons are actually unspecified behavior

```
int main(void)
{
  int a = 0;
  int b = 0;

  return &a < &b;                 // unspecified behavior
}
```

# Undefined Behavior

- Example 6
  - if max is very large the result will overflow
  - signed overflow of retval would be undefined behavior
  - efficient compiler knows undefined behavior can not happen
    - rewrites and optimizes the for loop as one computation
  - it is your responsibility to ensure this function is never called with a value which is too large

```
int sum_numbers(int max) {
  int retval = 0;

  for (int cnt = 1; cnt <= max; ++cnt) {
    retval += cnt;
  }

  return retval;
}
```

# Undefined Behavior

- Example 7
  - modifying an object which was originally declared const
  - keyword const_cast
    - used to remove the “constness” so the data can be modified
    - using this approach often means there is a design flaw
  - modifying tmp is undefined behavior ***if*** the passed argument was originally declared const

```
void doThing7(const std::string & str) {
   std::string &tmp = const_cast<std::string &>(str);
   tmp = “new information”;
}
```

# Undefined Behavior

- Example 8
  - specializing a standard type trait is undefined behavior
  - if this code were allowed the variable test would be true
  - it would be defined behavior to write your own type trait

```
namespace std {

  template<>
  struct is_pointer<int> : public std::true_type
  { };

}

bool test = std::is_pointer<int>::value;
```

# Undefined Behavior

- Sequence Point
  - a location in your code *(usually at the end of an expression)* where the side effects from all previous expressions are complete and pending expressions beyond that location have not been evaluated

  - your source code defines an order in which expressions are logically intended to be evaluated
  - compilers can reorder the evaluation of expressions however this process is constrained by sequence points

  - sequence points have been part of the core language since the beginning of C++

# Undefined Behavior

- What is a Side Effect
  - an expression has a side effect if it modifies some state
  - a side effect is anything beyond returning a value

  - common side effects
    - reading a volatile object
    - write access to any object
    - calling a library function which performs I/O
    - invoking a function which does any one of the above

```
varA = 5;                   // # of side effects in line 1?
varB = 2  + --varA;         // # of side effects in line 2?
```

# Undefined Behavior

- Sequencing
  - C++11 migrated away from sequence points and introduced a new abstraction called sequencing

  - definition of sequencing
    - expression A can be sequenced before expression B, which is the same as expression B is sequenced after expression A
    - indeterminately sequenced is where one expression is sequenced before the other, however it is unknown which will happen first
    - if A is not sequenced before B and B is not sequenced before A, evaluation of expression A and expression B are unsequenced *(may overlap)*

# Undefined Behavior

- Example 9
  - if two side effects or a side effect and a read occur on the same object and they are unsequenced, you have undefined behavior *(1.9.15, C++11 / 4.6.17, C++17)*
  - are the following expressions undefined behavior?

```
varA = 5;
varA = ++varA + 2;                    // pre increment



varB = 3;
varB = varB++ + 2;                    // post increment
```

# Undefined Behavior

- Example 9
  - if two side effects or a side effect and a read occur on the same object and they are unsequenced, you have undefined behavior *(1.9.15, C++11 / 4.6.17, C++17)*
  - are the following expressions undefined behavior?

```
varA = 5;
varA = ++varA + 2;                  // C++03, undefined behavior
varA == 8;                          // C++11, defined


varB = 3;
varB = varB++ + 2;                  // C++03, undefined behavior
varB == 5;                          // C++11, undefined behavior
                                    // C++17, defined
```

# Undefined Behavior

- Example 10
  - order of function parameter evaluation
    - was originally unspecified, could be interleaved
    - as of C++17 the order changed to indeterminate
  - evaluation of arguments is “sequenced before” the function call

```
int var1;
planDinner( var1 = doThing1(), doThing2(var1) );


var1 = doThing1();
doThing2(var1);
planDinner();

doThing2(var1);
var1 = doThing1();
planDinner();
```

# Undefined Behavior

- Example 11
  - which is the correct result

  - 4 + 3, save to element myArray[4]
  - 4 + 3, save to element myArray[5]
  - 4 + 3, save to element myArray[6]
  - 4 + 3 + 1, save to element myArray[7]
  - 4 + 3, save to element “boot sector”

```
varB = 4;
myArray[varB++] = varB++ + 3;
```

# Undefined Behavior

- Example 11
  - which is the correct result

  - the value computations, but not the side effects, of the operands to any operator are sequenced before the value computation of the result of the operator, but not its side effects *(1.9.15, C++11)*

```
varB = 4;
myArray[varB++] = varB++ + 3;              // C++11, undefined behavior
```

# Undefined Behavior

- Example 11
  - which is the correct result
  - in a subscript expression E1[E2], every value computation and side-effect of E1 is sequenced before every value computation and side effect of E2 *(8.2.1.1, C++17)*

```
varB = 4;
myArray[varB++] = varB++ + 3;          // C++11, undefined behavior


myArray[5] == 7;                       // C++17, defined
varB       == 6;
```

# Undefined Behavior

- C++17 Compiler Warnings
  - looking at the C++17 standard we know this code is no longer undefined behavior however current compilers show a warning
  - GCC 7.3, GCC 8.2
    - *operation on 'varB' may be undefined [-Wsequence-point]*
  - clang 6.0, clang 7.0
    - *unsequenced modification and access to 'varB' [-Wunsequenced]*

```
varB = 4;
myArray[varB++] = varB++ + 3;
```

# Undefined Behavior

- C++17 Compiler Warnings
  - similar examples are shown as being defined in C++17 which exhibit the same compiler warning messages
    - https://en.cppreference.com/w/cpp/language/eval_order
  - paraphrased from GCC documentation:
    - C++17 standard will define the order of evaluation of operands in more cases: in particular it requires the right-hand side of an assignment be evaluated before the left-hand side, so these examples are no longer undefined
    - this warning will still be shown to help people avoid writing code that is undefined in earlier versions of C++

# Undefined Behavior

- Undocumented Behavior - Flat Map
  - sorted vector of key /value pairs
  - intended for smaller data sets
  - stored in contiguous memory like an std::vector
  - has an API similar to std::map
  - uses less memory than standard map classes

```
QFlatMap<int, QString> data;
```

# Undefined Behavior

- Undocumented Behavior - Flat Map
  - C++ standard does not define a flat map container
  - there is no implementation of a flat map in the STL

  - abstraction of QFlatMap
    - should meet the STL associative container requirements
    - API should be sensible
  - implementation
    - designed to work with any data type for the key or value
    - key needs to be comparable
    - both the key and value need to be copyable

# Undefined Behavior

- Undocumented Undefined Behavior
  - doing something the standard does not specify (1) as defined, (2) as an error, or (3) having undefined behavior, is defined by the standard to be undefined behavior
  - a new class like QFlatMap adds functionality and new behavior
    - once defined the name of the class has a meaning
    - this class must be written correctly or it will have undefined behavior
  - anything which is not explicitly defined in the standard is undefined behavior *(section 3.27, C++17)*

# Undefined Behavior

- Avoiding Undefined Behavior
  - pay attention to compiler warnings
  - read the C++ standard or cppreference.com
  - try your code with multiple compilers
  - code reviews
  - test crazy corner cases
  - treat undefined behavior as a critical bug

  - static analyzer
    - coverity
    - clang static analyzer
    - purify

# Undefined Behavior

- Avoiding Undefined Behavior
  - clang
    - ASan Address Sanitizer
    - MSan Memory Sanitizer
    - UBSan Undefined Behavior Sanitizer
    - TSan Thread Sanitizer
  - gcc
    - ASan Address Sanitizer
    - UBSan Undefined Behavior Sanitizer
  - valgrind
    - third party product, combination of ASan and UBSan

# Presentations

- ❑ Why CopperSpice
- ❑ Why DoxyPress
- ❑ Compile Time Counter
- ❑ Modern C++ Data Types (references)
- ❑ Modern C++ Data Types (value categories)
- ❑ Modern C++ Data Types (move semantics)
- ❑ CsString library (unicode)
- ❑ CsString library (library design)
- ❑ Multithreading in C++
- ❑ Multithreading using libGuarded
- ❑ Signals and Slots
- ❑ Build Systems
- ❑ Templates in the Real World
- ❑ Copyright Copyleft
- ❑ What's in a Container
- ❑ Modern C++ Threads
- ❑ C++ Undefined Behavior
- ❑ Regular Expressions
- ❑ Using DoxyPress
- ❑ Type Traits
- ❑ C++ Tapas (typedef, forward declarations)
- ❑ Lambdas in C++
- ❑ C++ Tapas (typename, virtual, pure virtual)
- ❑ Overload Resolution
- ❑ Futures & Promises
- ❑ Special Member Functions
- ❑ C++ in Review
- ❑ Thread Safety
- ❑ Constexpr Static Const
- ❑ Next video October 11

Please subscribe to our YouTube Channel

https://www.youtube.com/copperspice

# Libraries

- CopperSpice
  - libraries for developing GUI applications
- CsSignal Library
  - standalone thread aware signal / slot library
- CsString Library
  - standalone unicode aware string library
- libGuarded
  - standalone multithreading library for shared data

# Applications

- KitchenSink
  - one program which contains 30 demos
  - links with almost every CopperSpice library

- Diamond
  - programmers editor which uses the CopperSpice libraries

- DoxyPress & DoxyPressApp
  - application for generating source code and API documentation

# Where to find CopperSpice

- www.copperspice.com

- ansel@copperspice.com
- barbara@copperspice.com

- source, binaries, documentation files
  - download.copperspice.com

- source code repository
  - github.com/copperspice

- discussion
  - forum.copperspice.com