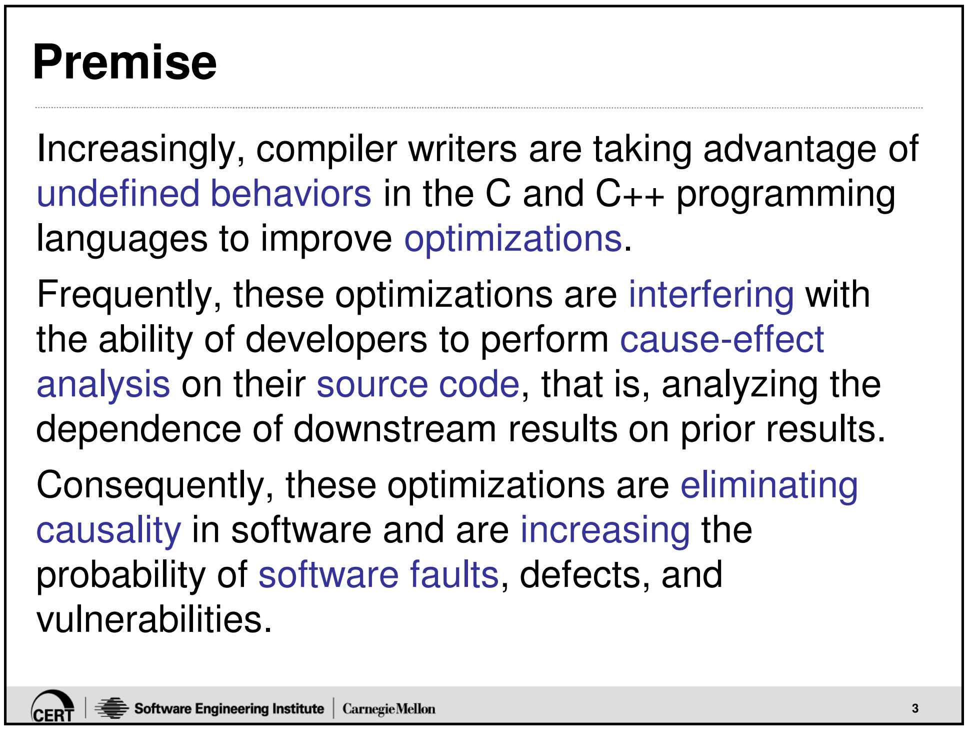
Dangerous Optimizations and the Loss of Causality

CS 15-392

Robert C. Seacord

Software Engineering Institute | Carnegie Mellon

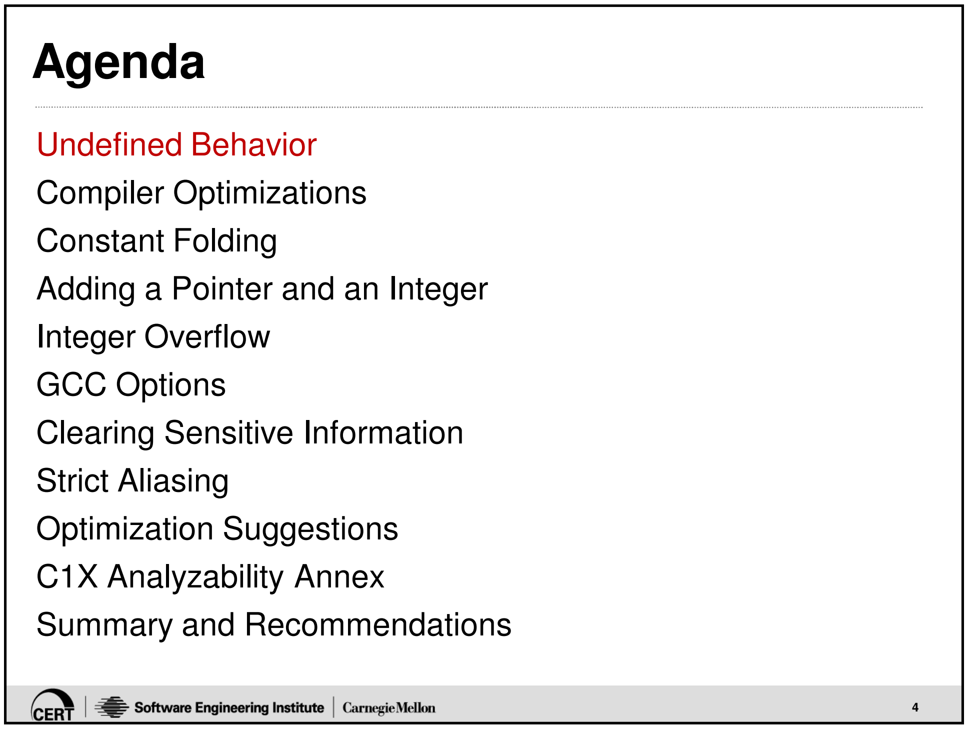


Premise

Increasingly, compiler writers are taking advantage of undefined behaviors in the C and C++ programming languages to improve optimizations.

Frequently, these optimizations are interfering with the ability of developers to perform cause-effect analysis on their source code, that is, analyzing the dependence of downstream results on prior results.

Consequently, these optimizations are eliminating causality in software and are increasing the probability of software faults, defects, and vulnerabilities.



Agenda

Undefined Behavior
Compiler Optimizations
Constant Folding
Adding a Pointer and an Integer
Integer Overflow
GCC Options
Clearing Sensitive Information
Strict Aliasing
Optimization Suggestions
C1X Analyzability Annex
Summary and Recommendations

Conformance [ISO/IEC 9899-1999]

implementation - Particular set of software, running in a particular translation environment under particular control options, that performs translation of programs for, and supports execution of functions in, a particular execution environment.

conforming - Conforming programs may depend on nonportable features of a conforming implementation.

strictly conforming - A strictly conforming program is one that uses only those features of the language and library specified in the international standard. Strictly conforming programs are intended to be maximally portable among conforming implementations and can't, for example, depend on implementation-defined behavior.

Behaviors [ISO/IEC 9899-1999]

implementation-defined behavior - Unspecified behavior whereby each implementation documents how the choice is made.

unspecified behavior - Behavior for which the standard provides two or more possibilities and imposes no further requirements on which is chosen in any instance.

undefined behavior - Behavior, upon use of a nonportable or erroneous program construct or of erroneous data, for which the standard imposes no requirements. An example of undefined behavior is the behavior on integer overflow.

Undefined Behaviors

Undefined behaviors are identified in the standard:

- If a “shall” or “shall not” requirement is violated, and that requirement appears outside of a constraint, the behavior is undefined.
- Undefined behavior is otherwise indicated in this International Standard by the words “undefined behavior”
- by the omission of any explicit definition of behavior.

There is no difference in emphasis among these three; they all describe “behavior that is undefined”.

C99 Annex J.2, “Undefined behavior,” contains a list of explicit undefined behaviors in C99.

Undefined Behaviors

Behaviors are classified as “undefined” by the standards committees to:

- give the implementer license not to catch certain program errors that are difficult to diagnose;
- avoid defining obscure corner cases which would favor one implementation strategy over another;
- identify areas of possible conforming language extension: the implementer may augment the language by providing a definition of the officially undefined behavior.

Implementations may

- ignore undefined behavior completely with unpredictable results
- behave in a documented manner characteristic of the environment (with or without issuing a diagnostic)
- terminate a translation or execution (with issuing a diagnostic).

Agenda

Undefined Behavior
Compiler Optimizations
Constant Folding
Adding a Pointer and an Integer
Integer Overflow
GCC Options
Clearing Sensitive Information
Strict Aliasing
Optimization Suggestions
C1X Analyzability Annex
Summary and Recommendations

"Optimizing compilers are so difficult to get right that we dare say that no optimizing compiler is completely error-free!"

Aho, Lam, Sethi, Ullman in *Compilers: Principles, Techniques, & Tools 2nd Edition*

Compiler Optimizations

The basic design of an optimizer for a C compiler is largely the same as an optimizer for any other procedural programming language.

The fundamental principle of optimization is to replace a computation with a more efficient method that computes the same result.

However, some optimizations change behavior

- Eliminate undefined behaviors (good)
- Introduce vulnerabilities (bad)

"As If" Rule 1

The ANSI C standard specifies the *results* of computations as if on an *abstract machine*, but the *methods* used by the compiler are not specified.

In the abstract machine, all expressions are evaluated as specified by the semantics.

An actual implementation need not evaluate part of an expression if it can deduce that

- its value is not used
- that no needed side effects are produced (including any caused by calling a function or accessing a volatile object).

The compiler's optimizer is free to choose any method that produces the correct result.

"As If" Rule 2

This clause gives compilers the leeway to remove code deemed unused or unneeded when building a program.

This is commonly called the "as if" rule, because the program must run *as if* it were executing on the abstract machine.

While this is usually beneficial, sometimes the compiler removes code that it thinks is not needed, even if the code has been added with security in mind.

Implementation Strategies

Hardware behavior: generate the corresponding assembler code, and let the hardware do whatever the hardware does. For many years, this was the nearly-universal policy, so several generations of C and C++ programmers have assumed that all compilers behave this way

Super debug: provide an intensive debugging environment to trap (nearly) every undefined behavior. This policy severely degrades the application's performance, so is seldom used for building applications.

Total license: treat any possible undefined behavior as a "can't happen" condition. This permits aggressive optimizations.

Total License Example 1

The total license policy has the effect of allowing anything to happen once any undefined behavior occurs in the program.

Consider the following example:

```
if (cond) {
  A[1] = X;
} else {
  A[0] = X;
}
```

A total license implementation could determine that, in the absence of any undefined behavior, the condition `cond` must have value 0 or 1.

That implementation could condense the entire statement into

```
  A[cond] = X;
```

Total License Example 2

On modern hardware, branches are often expensive, especially if the processor predicts them incorrectly, so transformations similar to this one are commonly used today.

If undefined behavior occurred somewhere in the integer arithmetic of `cond`, then `cond` could end up evaluating to a value other than 0 or 1, producing an out-of-bounds store that wasn't apparent from the original source code.

Code review or static analysis would conclude that the program modifies only `A[0]` or `A[1]`.

The total license implementation defeats the ability to analyze the behavior by

- a static analyzer
- a programmer performing a code review

Agenda

Undefined Behavior
Compiler Optimizations
Constant Folding
Adding a Pointer and an Integer
Integer Overflow
GCC Options
Clearing Sensitive Information
Strict Aliasing
Optimization Suggestions
C1X Analyzability Annex
Summary and Recommendations

Constant Folding 1

Constant folding is the process of simplifying constant expressions at compile time.

Terms in constant expressions can be

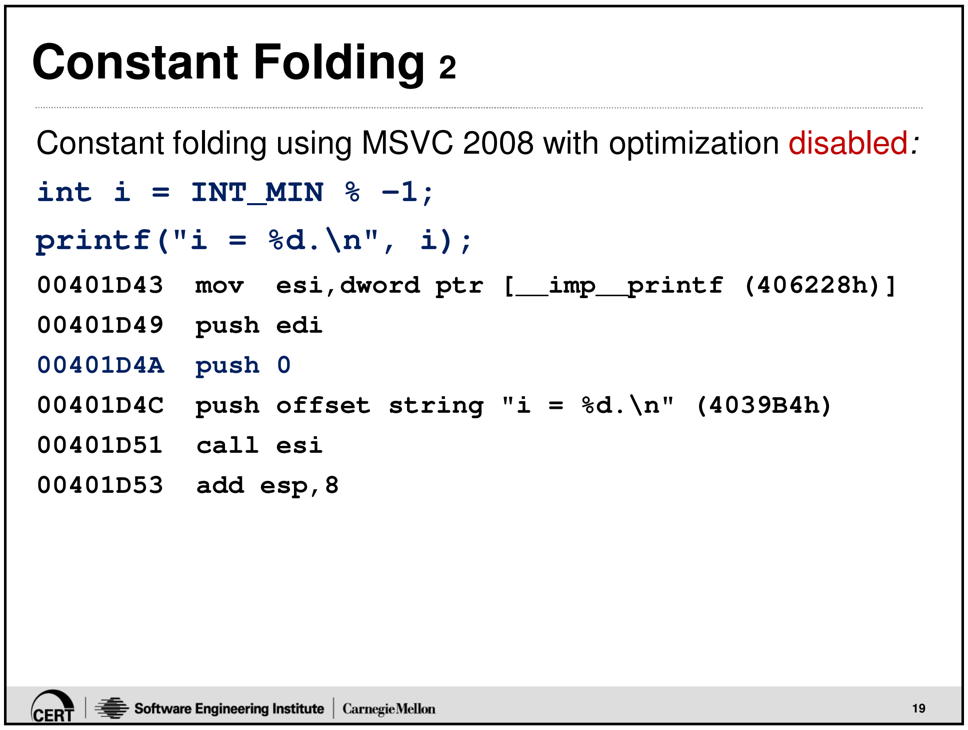
- simple literals, such as the integer 2
- variables whose values are never modified
- variables explicitly marked as constant

For example

```
int i = INT_MIN % -1;
printf("i = %d.\n", i);
```

Outputs:

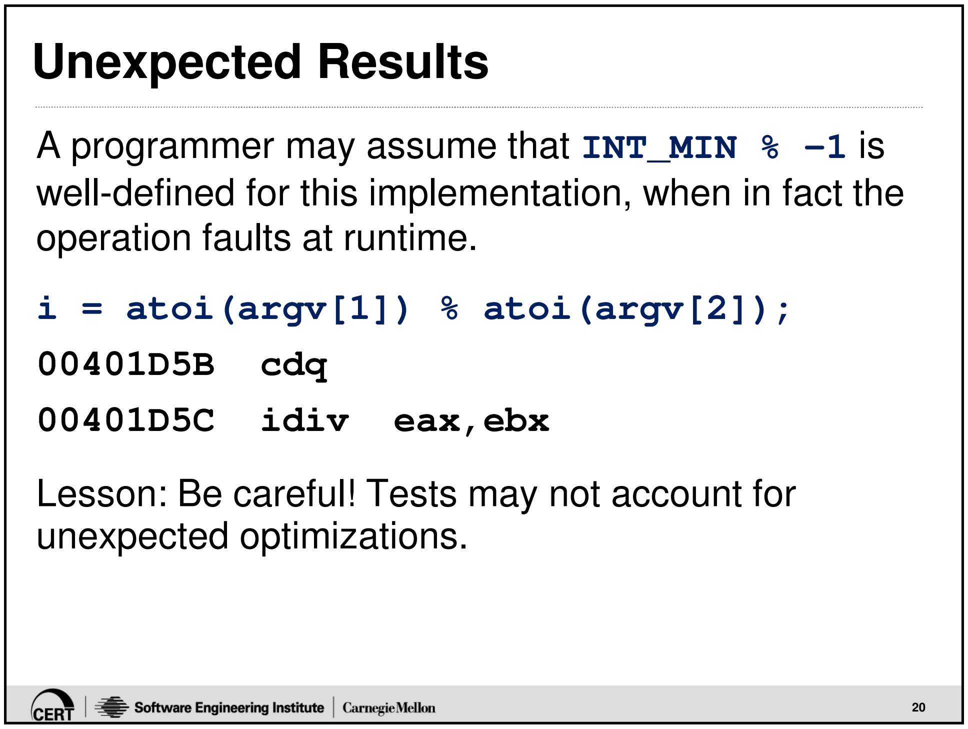
```
i = 0;
```



Constant Folding 2

Constant folding using MSVC 2008 with optimization disabled:

```
int i = INT_MIN % -1;
printf("i = %d.\n", i);
00401D43  mov  esi,dword ptr [__imp__printf (406228h)]
00401D49  push edi
00401D4A  push 0
00401D4C  push offset string "i = %d.\n" (4039B4h)
00401D51  call esi
00401D53  add esp,8
```



Unexpected Results

A programmer may assume that `INT_MIN % -1` is well-defined for this implementation, when in fact the operation faults at runtime.

```
i = atoi(argv[1]) % atoi(argv[2]);
00401D5B  cdq
00401D5C  idiv  eax,ebx
```

Lesson: Be careful! Tests may not account for unexpected optimizations.

Agenda

Undefined Behavior

Compiler Optimizations

Constant Folding

Adding a Pointer and an Integer

Integer Overflow

GCC Options

Clearing Sensitive Information

Strict Aliasing

Optimization Suggestions

C1X Analyzability Annex

Summary and Recommendations

Adding a Pointer and an Integer

From C99 §6.5.6p8:

When an expression that has integer type is added to or subtracted from a pointer, the result has the type of the pointer operand.

An expression like **`P[N]`** is translated into **`*(P+N)`**.

Adding a Pointer and an Integer

C99 Section 6.5.6 says

If both the pointer operand and the result point to elements of the same array object, or one past the last element of the array object, the evaluation shall not produce an overflow; otherwise, the behavior is undefined.

If the result points one past the last element of the array object, it shall not be used as the operand of a unary `*` operator that is evaluated.

Bounds Checking 1

A programmer might code a bounds-check such as

```
char *ptr; // ptr to start of array
char *max; // ptr to end of array
size_t len;
if (ptr + len > max)
  return EINVAL;
```

No matter what model is used, there is a bug.

If `len` is very large, it can cause `ptr + len` to overflow, which creates undefined behavior.

Under the hardware behavior model, the result would typically wrap-around—pointing to an address that is actually *lower* in memory than `ptr`.

Bounds Checking 2

In attempting to fix the bug, the experienced programmer (who has internalized the hardware behavior model of undefined behavior) might write a check like this:

```
if (ptr + len < ptr || ptr + len > max)
  return EINVAL;
```

However, compilers that follow the total license model may optimize out the first part of the check leaving the whole bounds check defeated

This is allowed because

- if `ptr` plus (an unsigned) `len` compares less than `ptr`, then an undefined behavior occurred during calculation of `ptr + len`
- the compiler can assume that undefined behavior never happens
- consequently `ptr + len < ptr` is dead code and can be removed

Algebraic Simplification

Optimizations may be performed for comparisons between `P + V1` and `P + V2`, where `P` is the same pointer and `V1` and `V2` are variables of some integer type.

The total license model permits this to be reduced to a comparison between `V1` and `V2`.

However, if `V1` or `V2` are such that the sum with `P` overflows, then the comparison of `V1` and `V2` will not yield the same result as actually computing `P + V1` and `P + V2` and comparing the sums.

Because of possible overflows, computer arithmetic does not always obey the algebraic identities of mathematics.

Algebraic Simplification Applied

In our example:

```
if (ptr + len < ptr || ptr + len > max)
  return EINVAL;
```

this optimization translates as follows:

```
ptr + len < ptr
ptr + len < ptr + 0
len < 0 (impossible, len is unsigned)
```

Mitigation

This problem is easy to remediate, once it is called to the attention of the programmer, such as by a diagnostic message when dead code is eliminated.

For example, if it is known that `ptr` is less-or-equal-to `max`, then the programmer could write:

```
if (len > max – ptr)
  return EINVAL;
```

This conditional expression eliminates the possibility of undefined behavior.

Another Algebraic Simplification

In this example, the expression `buf + n` may wrap for large values of `n`, resulting in undefined behavior.

```
int f(char *buf, size_t n) {
  return buf + n < buf + 100;
}
```

When compiled using GCC 4.3.0 with the `–O2` option, for example, the expression

```
buf + n < buf + 100
```

is optimized to `n < 100`, eliminating the possibility of wrapping.

Probably not a big deal unless one expression wraps but not the other.

Mitigation

This code example is still incorrect, because it is not safe to rely on compiler optimizations for security.

The undefined behavior can be eliminated by performing the optimization by hand.

```
int f(char *buf, size_t n) {
  return n < 100;
}
```

GCC Details

The behavior of pointer overflow changed as of the following versions:

- gcc 4.2.4
- gcc 4.3.1
- gcc 4.4.0

and all subsequent versions

- 4.2.x where x >= 4
- 4.3.y where y >= 1
- 4.z where z >= 4)

GCC Details

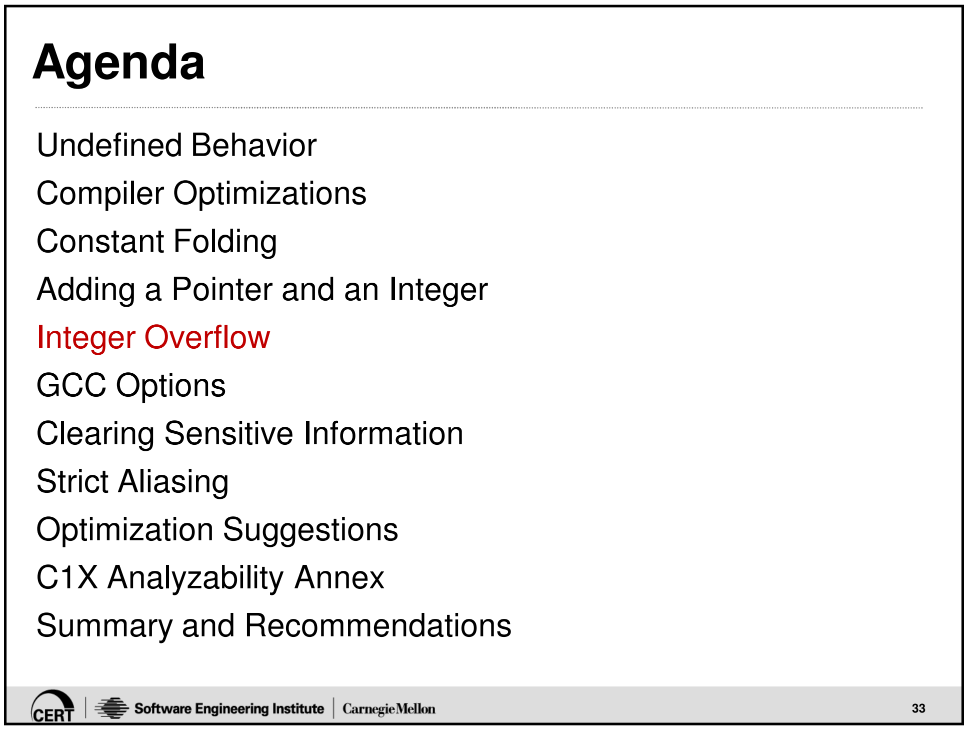
The optimization is

- performed by default at `-O2` and above, including `-Os`.
- not performed by default at `-O1` or `-O0`.

The optimization may be

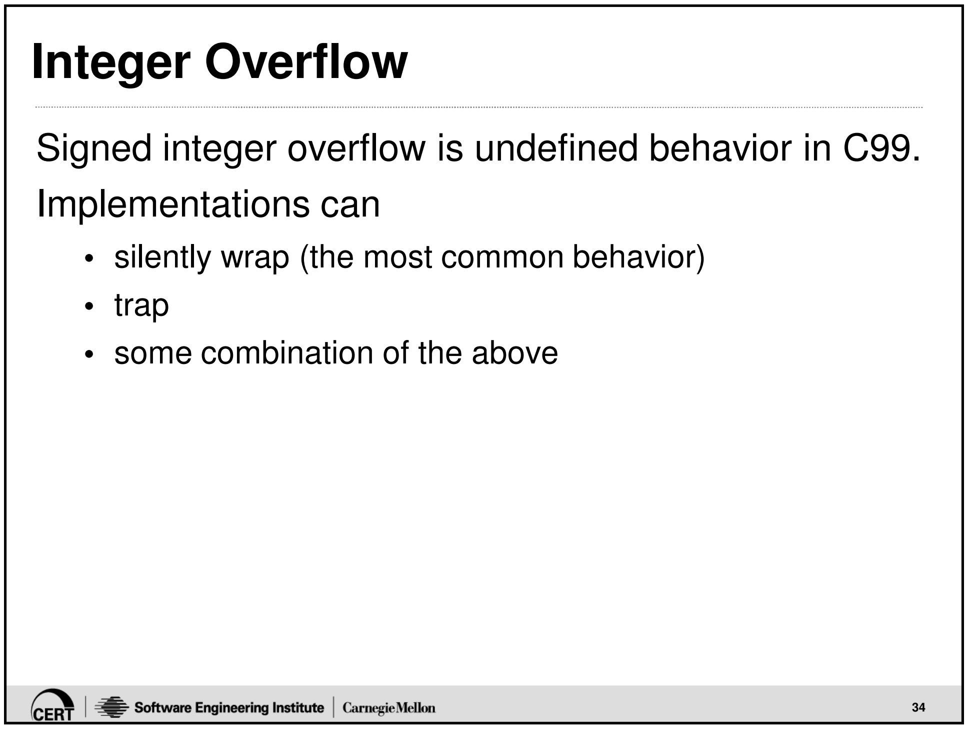
- enabled for `-O1` with the `-fstrict-overflow` option.
- disabled for `-O2` and above with the `-fno-strict-overflow` option.

Cases where optimization occurs may be detected by using `-Wstrict-overflow=N` where `N >= 3`.



Agenda

Undefined Behavior
Compiler Optimizations
Constant Folding
Adding a Pointer and an Integer
Integer Overflow
GCC Options
Clearing Sensitive Information
Strict Aliasing
Optimization Suggestions
C1X Analyzability Annex
Summary and Recommendations



Integer Overflow

Signed integer overflow is undefined behavior in C99.

Implementations can

- silently wrap (the most common behavior)
- trap
- some combination of the above

Integer Overflow

This code assumes that if it keeps doubling a positive number, it will eventually get a negative number.

```
int f() {
  int i;
  int j = 0;
  for (i = 1; i > 0; i += i)
    ++j;
  return j;
}
```

When compiled with `-O2`, gcc v 4.3.2 interprets this code according to the total license model in which overflow can not occur and compiles this code into an infinite loop.

Real World Example

In the following example, derived from the GNU C Library 2.5 implementation of `mktime` (2006-09-09), the code assumes wraparound arithmetic in + to detect signed overflow:

```
time_t t, t1, t2;
int sec_requested, sec_adjustment;
  ...
t1 = t + sec_requested;
t2 = t1 + sec_adjustment;
if (((t1 < t) != (sec_requested < 0))
  | ((t2 < t1) != (sec_adjustment < 0)))
  return -1;
```

Hoisting of Loop-invariant Computations

Loop-invariant code consists of statements which can be moved outside the body of a loop without affecting the semantics of the program.

Loop-invariant code motion (also called hoisting or scalar promotion) is a compiler optimization which performs this movement automatically.

Loop-invariant code which has been hoisted out of a loop is executed less often, providing a speedup.

Optimization Constraints

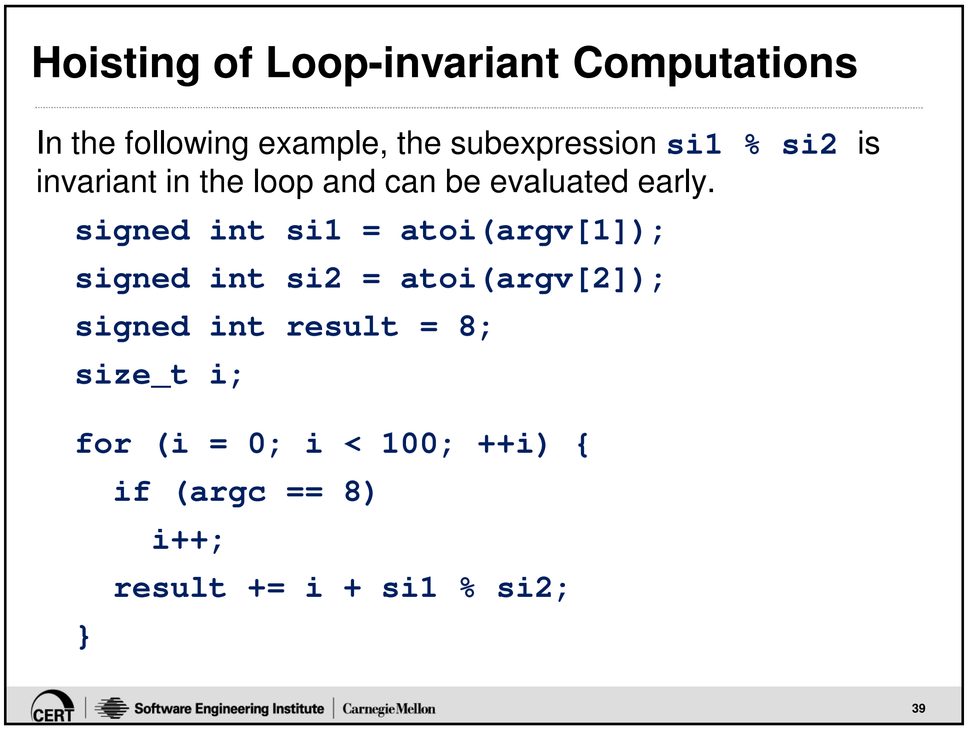
Implementations that detect signed integer overflows are constrained not to transform a program that does not get an integer overflow into a program that does get an integer overflow. It is not permitted to transform:

```
if (z < INT_MAX) {
  y = z + 1;
}
```

into

```
temp = z + 1;
if (z < INT_MAX) {
  y = temp;
}
```

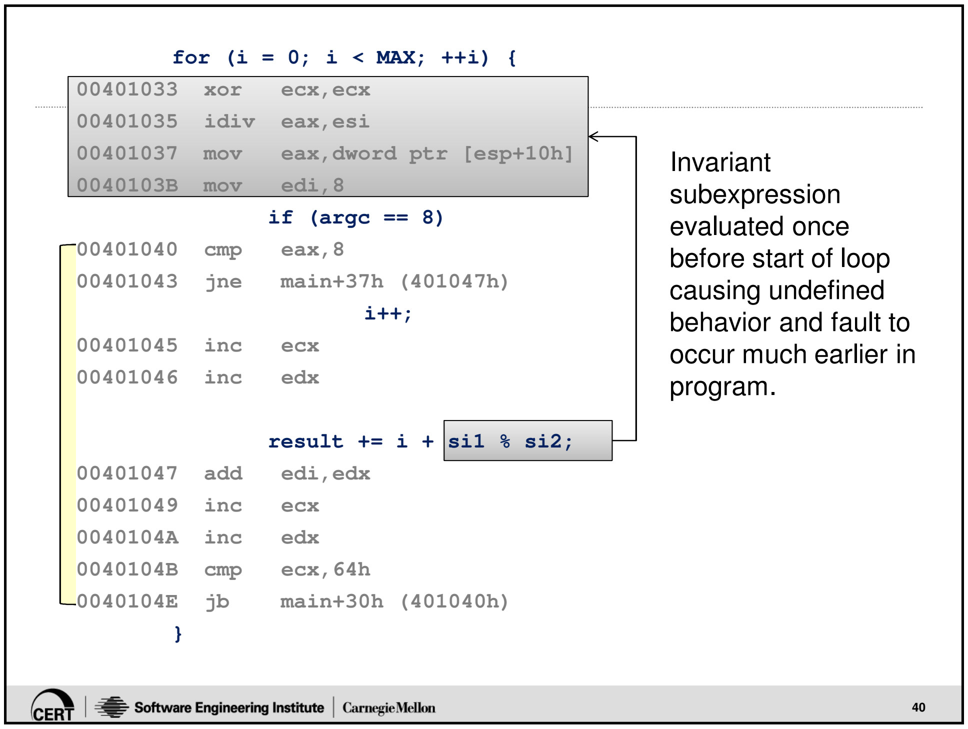
unless the implementation can prove that `z + 1` is not introducing an overflow into a program that never had an overflow before.



Hoisting of Loop-invariant Computations

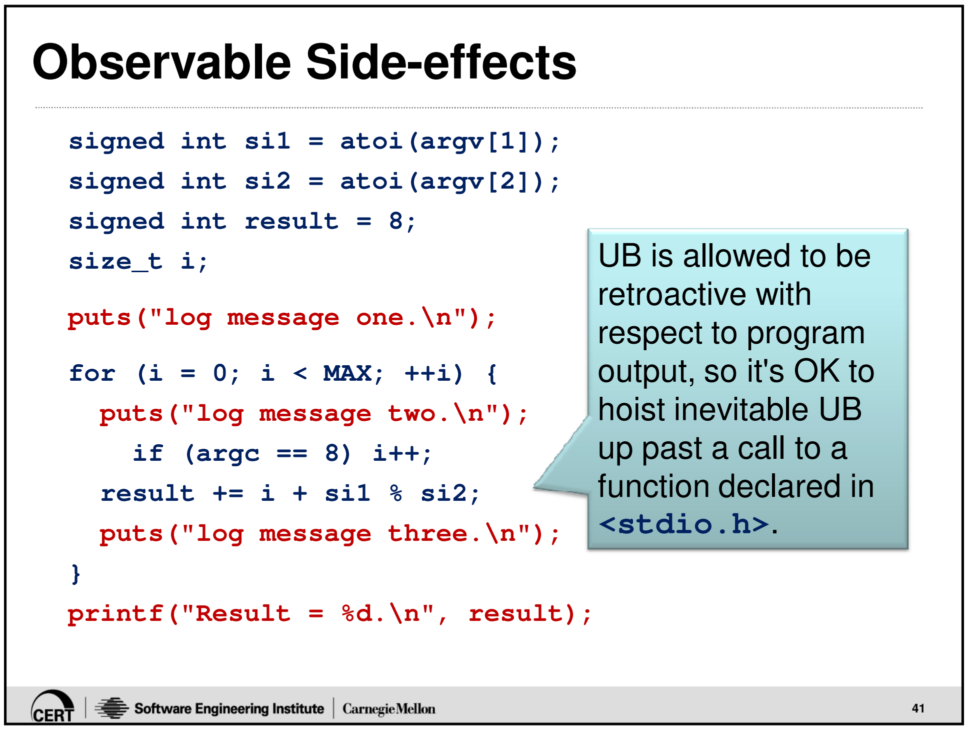
In the following example, the subexpression `si1 % si2` is invariant in the loop and can be evaluated early.

```
signed int si1 = atoi(argv[1]);
signed int si2 = atoi(argv[2]);
signed int result = 8;
size_t i;

for (i = 0; i < 100; ++i) {
  if (argc == 8)
    i++;
  result += i + si1 % si2;
}
```



```
        for (i = 0; i < MAX; ++i) {
00401033  xor   ecx,ecx
00401035  idiv  eax,esi
00401037  mov   eax,dword ptr [esp+10h]
0040103B  mov   edi,8
               if (argc == 8)
00401040  cmp   eax,8
00401043  jne   main+37h (401047h)
                     i++;
00401045  inc   ecx
00401046  inc   edx

               result += i + si1 % si2;
00401047  add   edi,edx
00401049  inc   ecx
0040104A  inc   edx
0040104B  cmp   ecx,64h
0040104E  jb    main+30h (401040h)
        }
```

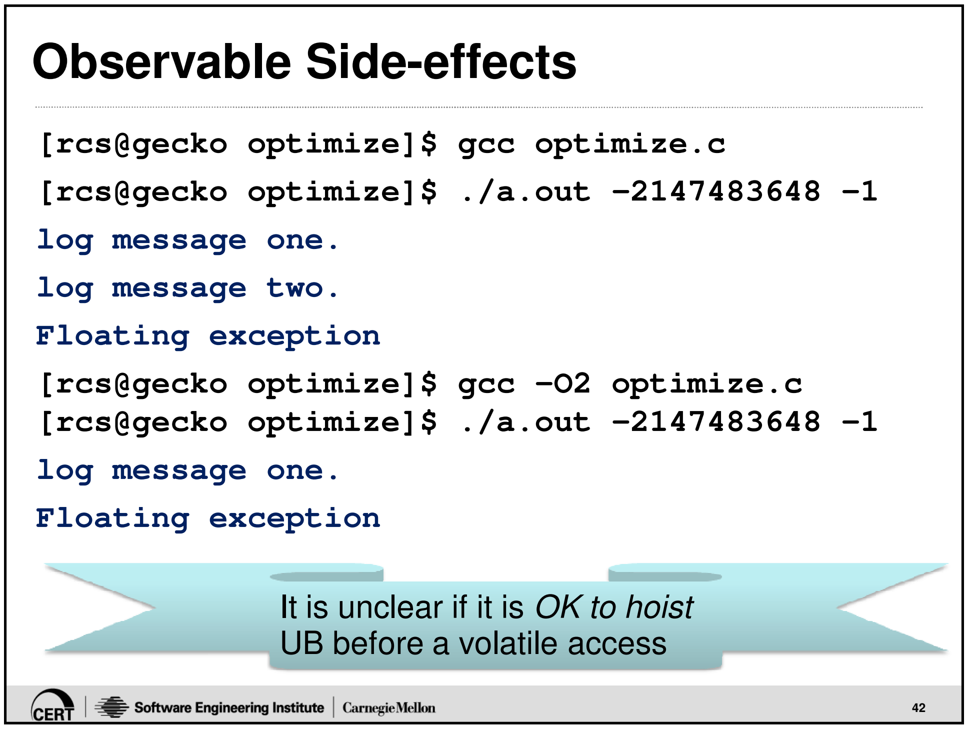
Invariant subexpression evaluated once before start of loop causing undefined behavior and fault to occur much earlier in program.



Observable Side-effects

```
signed int si1 = atoi(argv[1]);
signed int si2 = atoi(argv[2]);
signed int result = 8;
size_t i;

puts("log message one.\n");

for (i = 0; i < MAX; ++i) {
  puts("log message two.\n");
    if (argc == 8) i++;
  result += i + si1 % si2;
  puts("log message three.\n");
}
printf("Result = %d.\n", result);
```

UB is allowed to be retroactive with respect to program output, so it's OK to hoist inevitable UB up past a call to a function declared in `<stdio.h>`.



Observable Side-effects

```
[rcs@gecko optimize]$ gcc optimize.c
[rcs@gecko optimize]$ ./a.out -2147483648 -1
log message one.
log message two.
Floating exception
[rcs@gecko optimize]$ gcc -O2 optimize.c
[rcs@gecko optimize]$ ./a.out -2147483648 -1
log message one.
Floating exception
```

It is unclear if it is *OK to hoist* UB before a volatile access

Incompatible Results 1

Compilers can generate code that is incompatible with wraparound integer arithmetic.

An example is an algebraic simplification

For example, a compiler can translate

```
(i * 2000) / 1000
```

to

```
 i * 2
```

because it assumes that `i * 2000` does not overflow.

The translation is not equivalent to the original when overflow occurs.

Incompatible Results 2

In the typical case of 32-bit signed two's complement wraparound `int`, if `i` has type `int` and value 1073742 the

- non-optimized evaluates to –2147483
- optimized evaluates to the mathematically correct value 2147484.

Changing behavior between the non-optimized test code and the optimized deliverable code

- can have negative consequences if the correct execution of the code depends upon the wrapping behavior
- eliminates undefined behavior causing mathematically incorrect results

Instead, perform algebraic simplification in the source code

- can be assisted by diagnostics such as `-Wstrict-overflow`
- analysis/debugging tools and techniques

Loop Induction Variables

An induction variable is a variable that gets increased or decreased by a fixed amount on every iteration of a loop, or is a linear function of another induction variable.

For example, in the following loop, `i` and `j` are induction variables:

```
for (i = 0; i < 10; ++i) {
    j = 17 * i;
}
```

Strength Reduction

A common compiler optimization is to recognize the existence of induction variables and replace them with simpler computations.

Assuming that the addition of a constant is cheaper than a multiplication, the previous example could be rewritten by the compiler as follows:

```
j = 0;
for (i = 0; i < 10; ++i) {
  j = j + 17;
}
```

Loop Induction

Loop induction optimizations often take advantage of the undefined behavior of signed overflow.

```
int sumc(int lo, int hi) {
    int sum = 0;
    int i;
    for (i = lo; i <= hi; i++)
      sum ^= i * 53;
    return sum;
 }
```

Loop Induction and Overflow

To avoid multiplying by 53 each time through the loop an optimizing compiler might internally transform `sumc` to the following equivalent form:

```
int transformed_sumc (int lo, int hi) {
  int sum = 0;
  int hic = hi * 53;
  int ic;
  for (ic = lo * 53; ic <= hic; ic += 53)
    sum ^= ic;
  return sum;
}
```

This transformation is invalid for wraparound arithmetic when `INT_MAX / 53 < hi`, because then the overflow in computing expressions like `hi * 53` can cause the expression `i <= hi` to yield a different value from the transformed expression `ic <= hic`.

Loop Induction and Overflow

Compilers that use loop induction and similar techniques often do not support reliable wraparound arithmetic when a loop induction variable is involved.

It is not always trivial to say whether the problem affects your code

- loop induction variables are generated by the compiler
- are not visible in the source code

Unsigned Wrap

The C and C++ standards require that unsigned integers wrap according to the rules of modulo arithmetic.

gcc never optimizes based on assuming that they do not wrap.

The IBM XL C/C++ compiler has a `strict_induction` option

- turns off induction variable optimizations that have the potential to alter the semantics of a user's program.
- used if a program contains loop induction variables that overflow or wrap around.
- at `-O` or higher optimization level, the XL C/C++ compiler optimizes both signed and unsigned loop induction variables when the `NOSTRICT_INDUCTION` option is in effect.

Restricted Range Usage 1

For the comparison `a < b`, there is often an implicit subtraction.

- On a machine without condition codes (for example, the Cray-2), the compiler may issue a subtract instruction and check whether the result is negative.
- This is allowed, because the compiler is allowed to assume there is no overflow.

Restricted Range Usage 2

If the user types the expression `a - b` where both `a` and `b` are in the range [`INT_MIN/2, INT_MAX/2`], then the result is in the range (`INT_MIN, INT_MAX`] for a typical two's complement machine.

If all explicitly user-generated values are kept in the range [`INT_MIN/2, INT_MAX/2`], then comparisons will always work even if the compiler performs this optimization.

This has been a trick of the trade in Fortran for some time, and now that optimizing C compilers are becoming more sophisticated, it can be valuable in C.

Agenda

Undefined Behavior
Compiler Optimizations
Constant Folding
Adding a Pointer and an Integer
Integer Overflow
GCC Options
Clearing Sensitive Information
Strict Aliasing
Optimization Suggestions
C1X Analyzability Annex
Summary and Recommendations

`-fstrict-overflow`

Allow the compiler to assume strict signed overflow rules (total license policy).

- signed arithmetic overflow is undefined behavior
- the compiler assume undefined behavior will not happen
- permits various optimizations

For example, the compiler assumes that an expression like `i + 10 > i` is always true for signed `i`.

This assumption is only valid if signed overflow is undefined, as the expression is false if `i + 10` overflows when using twos complement arithmetic.

`-fstrict-overflow`

When this option is in effect any attempt to determine whether an operation on signed numbers will overflow must be written carefully to not actually involve overflow.

The `-fstrict-overflow` option is enabled at levels `-O2`, `-O3`, `-Os`.

`-fwrapv`

This option instructs gcc to assume that signed arithmetic overflow of addition, subtraction and multiplication wraps around using twos-complement representation.

Very similar to `-fno-strict-overflow`

This flag disables optimizations that assume integer overflow behavior is undefined.

This option is enabled by default for the Java front-end, as required by the Java language specification.

`-Wstrict-overflow=n`

This option only applies when `-fstrict-overflow` is active.

It warns about cases where the compiler optimizes based on the assumption that signed overflow does not occur.

Only warns about overflow in cases where the compiler implements some optimization.

Consequently, this warning depends on the optimization level.

`-Wstrict-overflow=n`

An optimization which assumes that signed overflow does not occur is safe as long as the values of the variables involved are such that overflow does not occur.

Therefore this warning can easily give a false positive: a warning about code which is not actually a problem.

No warnings are issued for the use of undefined signed overflow when estimating how many iterations a loop will require, in particular when determining whether a loop will be executed at all.

`-Wstrict-overflow=1`

Warn about cases which are both questionable and easy to avoid.

For example:

```
x + 1 > x;
```

With `-fstrict-overflow`, the compiler simplifies this to 1.

This level of `-Wstrict-overflow` is enabled by `-Wall`; higher levels are not, and must be explicitly requested.

`-Wstrict-overflow=2`

Also warns about cases where a comparison is simplified to a constant.

For example: `abs(x) >= 0`. This can only be simplified when `-fstrict-overflow` is in effect, because `abs(INT_MIN)` overflows to `INT_MIN`, which is less than zero.

`-Wstrict-overflow=3,4`

`-Wstrict-overflow=3` also warns about other cases where a comparison is simplified.

For example:

`x + 1 > 1` is simplified to `x > 0`.

This is the lowest warning level at which arithmetic simplification leading to bounds checking errors are diagnosed.

`-Wstrict-overflow=4` also warns about other simplifications not covered by the above cases.

For example:

`(x * 10) / 5` is simplified to `x * 2`.

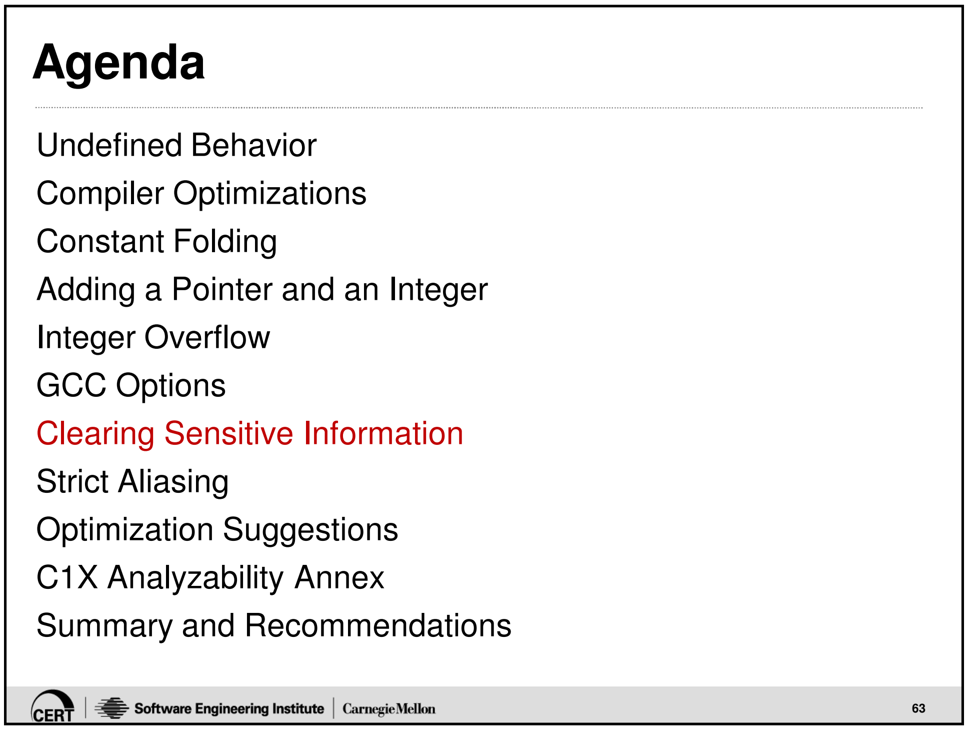
`-Wstrict-overflow=5`

Also warns about cases where the compiler reduces the magnitude of a constant involved in a comparison.

For example:

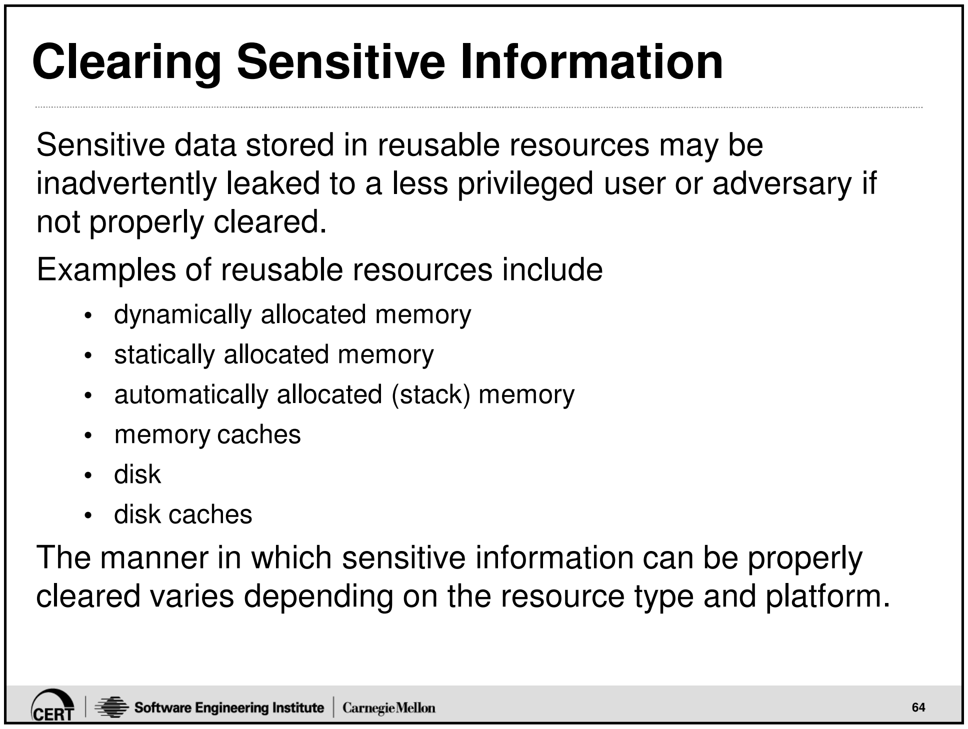
`x + 2 > y` is simplified to `x + 1 >= y`.

This is reported only at the highest warning level because this simplification applies to many comparisons, so this warning level results in a large number of false positives.



Agenda

Undefined Behavior
Compiler Optimizations
Constant Folding
Adding a Pointer and an Integer
Integer Overflow
GCC Options
Clearing Sensitive Information
Strict Aliasing
Optimization Suggestions
C1X Analyzability Annex
Summary and Recommendations



Clearing Sensitive Information

Sensitive data stored in reusable resources may be inadvertently leaked to a less privileged user or adversary if not properly cleared.

Examples of reusable resources include

- dynamically allocated memory
- statically allocated memory
- automatically allocated (stack) memory
- memory caches
- disk
- disk caches

The manner in which sensitive information can be properly cleared varies depending on the resource type and platform.

Dynamic Memory

Dynamic memory managers are not required to clear freed memory and generally do not because of the additional runtime overhead.

Furthermore, dynamic memory managers are free to reallocate this same memory.

As a result, it is possible to accidentally leak sensitive information if it is not cleared before calling a function that frees dynamic memory.

Reallocating memory using the `realloc()` function is a regenerative case of freeing memory.

Clearing Sensitive Information

To prevent information leakage, dynamic memory containing sensitive information should be sanitized before being freed.

This is commonly accomplished by clearing the allocated space (that is, filling the space with `'\0'` characters).

```
void getPassword(void) {
  char pwd[64];
  if (GetPassword(pwd, sizeof(pwd))) {
    /* check password */
  }
  memset(pwd, 0, sizeof(pwd));
}
```

Dead Code Removal

Compilers may remove code sections if the optimizer determines that doing so will not alter the behavior of the program.

An optimizing compiler could employ “dead store removal”; that is, it could decide that `pwd` is never accessed after the call to `memset()`, therefore the call to `memset()` can be optimized away.

Consequently, the password remains in memory, possibly to be discovered by some other process requesting memory.

There are several solutions to this problem, but no solution appears to be both portable and optimal.

`ZeroMemory()`

This example uses the `ZeroMemory()` function provided by many versions of the MSVC.

```
void getPassword(void) {
  char pwd[64];
  if (retrievePassword(pwd, sizeof(pwd))) {
    /* check password */
  }
  ZeroMemory(pwd, sizeof(pwd));
}
```

A call to `ZeroMemory()` may be optimized out in a similar manner as a call to `memset()`.

SecureZeroMemory()

The MSVC `SecureZeroMemory()` function guarantees that the compiler does not optimize out this call when zeroing memory.

```
void getPassword(void) {
  char pwd[64];
  if (retrievePassword(pwd, sizeof(pwd))) {
    /* check password */
  }
  SecureZeroMemory(pwd, sizeof(pwd));
}
```

#pragma Directives

The `#pragma` directives in this example instruct the compiler to avoid optimizing the enclosed code.

```
void getPassword(void) {
  char pwd[64];
  if (retrievePassword(pwd, sizeof(pwd))) {
    /* check password */
  }
#pragma optimize("", off)
  memset(pwd, 0, sizeof(pwd));
#pragma optimize("", on)
}
```

This `#pragma` directive is supported on some versions of Microsoft Visual Studio and may be supported on other compilers.

Volatile-qualified Types

An object that has volatile-qualified type may be modified in ways unknown to the implementation or have other unknown side effects.

The `volatile` keyword imposes restrictions on access and caching.

According to the C99 Rationale [ISO/IEC 03]:

No cacheing through this lvalue: each operation in the abstract semantics must be performed (that is, no cacheing assumptions may be made, since the location is not guaranteed to contain any previous value).

In the absence of this qualifier, the contents of the designated location may be assumed to be unchanged except for possible aliasing.

Dead Code Removal

This code accesses the buffer after the call to `memset()`.

```
void getPassword(void) {
  char pwd[64];
  if (retrievePassword(pwd, sizeof(pwd))) {
    /* checking of password */
  }
  memset(pwd, 0, sizeof(pwd));
  *(volatile char*)pwd = *(volatile char*)pwd;
}
```

Some implementations nullify only the first byte and leave the remainder intact.

secure_memset() Solution

The volatile type qualifier informs the compiler that the memory should be overwritten and that the call to the memset_s() function should not be optimized out.

```
void *secure_memset(void *v, int c, size_t n)
{
  volatile unsigned char *p = v;
  while (n--)
    *p++ = c;
  return v;
}
```

secure_memset() Solution

Prevents the clearing of memory from being optimized away, and should work on any standard-compliant platform.

However, some compilers violate the standard by not always respecting the volatile qualifier.

Also, this compliant solution may not be as efficient as possible because the volatile type qualifier prevents the compiler from optimizing the code at all.

Typically, some compilers replace calls to memset() with equivalent assembly instructions that are much more efficient than the memset() implementation.

Implementing a secure_ memset() function as shown in the example may prevent the compiler from using the optimal assembly instructions and may result in less efficient code.

7.21.6.2 The `memset_s` Function

Synopsis

```
errno_t memset_s(void * restrict s,
  rsize_t smax, int c, rsize_t n)
```

Description

The `memset_s` function copies the value of `c` (converted to an `unsigned char`) into each of the first `n` characters of the object pointed to by `s`. Unlike `memset`, any call to `memset_s` shall be evaluated strictly according to the rules of the abstract machine, as described in 5.1.2.3. That is, any call to `memset_s` shall assume that the memory indicated by `s` and `n` may be accessible in the future and therefore must contain the values indicated by `c`.

Clearing Sensitive Information

While necessary for working with sensitive information, this `memset_s()` function may not be sufficient, as it does nothing to prevent memory from being swapped to disk, or written out in a core dump.

More information on such issues is available at the CERT C Secure Coding guideline MEM06-C.

Common Error Using Volatile

This example contains a common mistake in which a volatile variable is used to signal a condition about a non-volatile data structure to another thread:

```
volatile int buffer_ready;
char buffer[BUF_SIZE];
void buffer_init() {
  for (size_t i = 0; i < BUF_SIZE; i++)
    buffer[i] = 0;
  buffer_ready = 1;
}
```

The for-loop does not access any volatile locations or perform any side-effecting operations.

Therefore, the compiler is free to move the loop below the store to `buffer_ready`, defeating the developer’s intent.

`volatile` Semantics

The semantics of `volatile` can’t always be trusted.

```
void *pointer = (void *)0xDEADBEEF;
/* … */
while (*((volatile int *)ptr) & FLAG){}
```

gcc generates a single read instead of a loop for the following code when

- reading and writing from registers in Linux code
- compiled with a special branch of gcc based off 4.3.2

Unclear what C99 requires in this situation.

Compiler Bugs

The following C code for a function that resets a watchdog timer in a hypothetical embedded system:

```
/* linker maps to the proper IO register */
extern volatile int WATCHDOG;
void reset_watchdog() {
  WATCHDOG = WATCHDOG; /* load, then store */
}
```

Regardless of optimization level, a conforming compiler must covert this to object code that loads and then stores the **`WATCHDOG`** register.

Compiler Bugs

Recent versions of GCC for IA-32 emit correct assembly code:

```
reset_watchdog:
     movl WATCHDOG, %eax
     movl %eax, WATCHDOG
     ret
```

However, the latest version of GCC's port to the MSP430 microcontroller compiles the code into the following assembly:

```
reset_watchdog:
     ret
```

Agenda

Undefined Behavior

Compiler Optimizations

Adding a Pointer and an Integer

Integer Overflow

GCC Options

Clearing Sensitive Information

Strict Aliasing

Optimization Suggestions

C1X Analyzability Annex

Summary and Recommendations

Compatible Types in C

Two types are compatible if their types are the same.

Additional rules for determining whether two types are compatible are described in C99

- §6.7.2 for type specifiers
- §6.7.3 for type qualifiers
- §6.7.5 for declarators

Two types need not be identical to be compatible.

If there are no rules allowing types to be compatible, they are not.

Compatible Types in C

Qualified types

- For two qualified types to be compatible, both must have the identically qualified version of a compatible type; the order of type qualifiers within a list of specifiers or qualifiers does not affect the specified type.

Pointer types

- For two pointer types to be compatible, both must be identically qualified and both must be pointers to compatible types.

Array types

- For two array types to be compatible, both must have compatible element types, and if both size specifiers are present, and are integer constant expressions, then both size specifiers must have the same constant value.

Function Type Compatibility

A function type declared using the new prototype-style declaration (such as `int tree (int x)`) is compatible with another function type declared with a function prototype if:

- The return types are compatible.
- The parameters agree in number (including an ellipsis if one is used).
- The parameter types are compatible. For each parameter declared with a qualified type, its type for compatibility comparison is the unqualified version of the declared type.

Type Compatibility

Two structure, union, or enumerated types declared in separate translation units are compatible if their tags and members satisfy the following requirements:

- If one is declared with a tag, the other shall be declared with the same tag.
- If both are complete types, then the following additional requirements apply:
  - there shall be a one-to-one correspondence between their members such that each pair of corresponding members are declared with compatible types, such that if one member of a corresponding pair is declared with a name, the other member is declared with the same name.

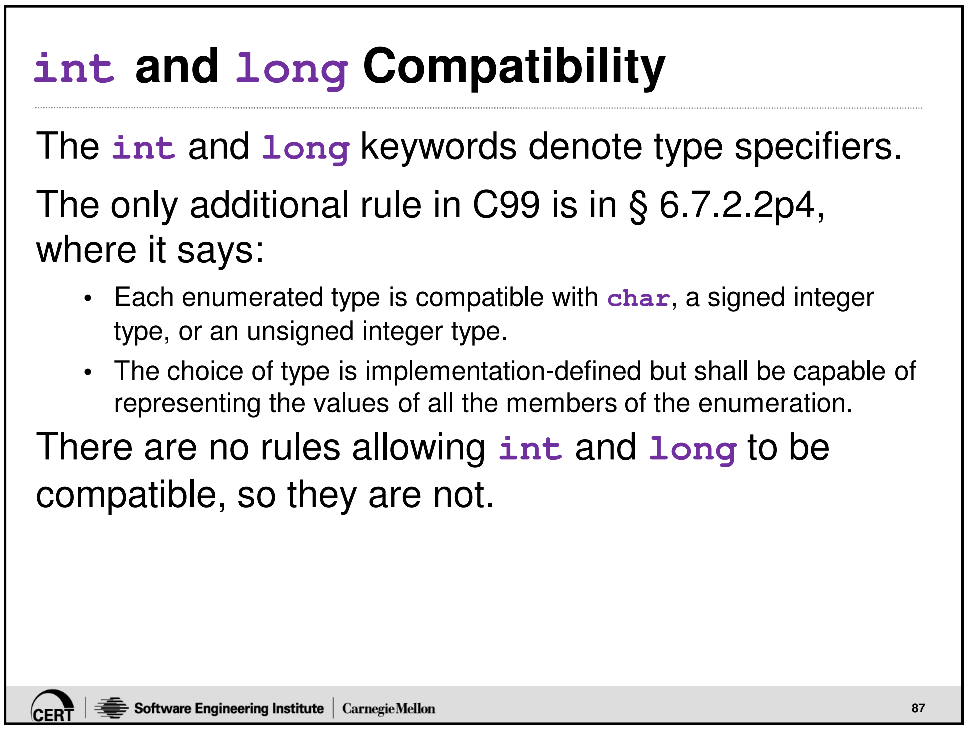
For two structures, corresponding members shall be declared in the same order. For two structures or unions, corresponding bit-fields shall have the same widths.

For two enumerations, corresponding members shall have the same values.

Incompatible Types

The following types, which may appear to be compatible, are not:

- `unsigned int` and `int` types are not compatible
- `char`, `signed char`, and `unsigned char` types are not compatible
- `int` and `long` types are not compatible, even if they have the same representation



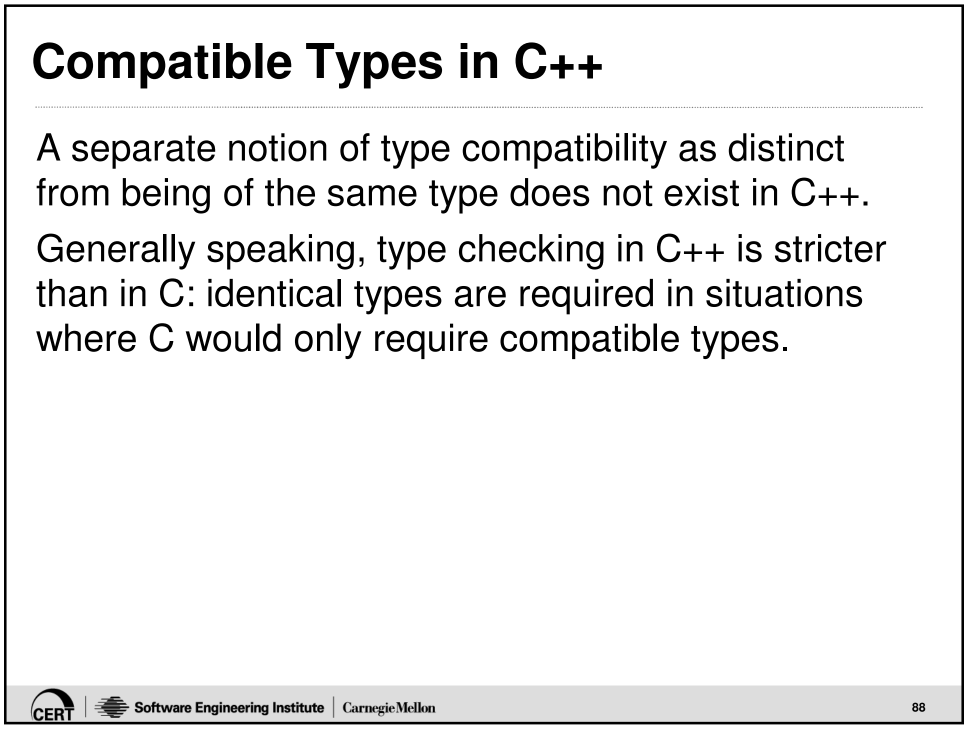
`int` and `long` Compatibility

The `int` and `long` keywords denote type specifiers.

The only additional rule in C99 is in § 6.7.2.2p4, where it says:

- Each enumerated type is compatible with `char`, a signed integer type, or an unsigned integer type.
- The choice of type is implementation-defined but shall be capable of representing the values of all the members of the enumeration.

There are no rules allowing `int` and `long` to be compatible, so they are not.



Compatible Types in C++

A separate notion of type compatibility as distinct from being of the same type does not exist in C++.

Generally speaking, type checking in C++ is stricter than in C: identical types are required in situations where C would only require compatible types.

Effective Type

The purpose of the effective type rules is to impute a type (dynamically) to a dynamically-allocated object.

The effective type of an object for an access to its stored value is the declared type of the object, if any (allocated objects have no declared type).

If a value is stored into an object having no declared type through an lvalue having a type that is not a character type, then the type of the lvalue becomes the effective type of the object for that access and for subsequent accesses that do not modify the stored value.

Effective Type

If a value is copied into an object having no declared type using `memcpy()` or `memmove()`, or is copied as an array of character type, then the effective type of the modified object for that access and for subsequent accesses that do not modify the value is the effective type of the object from which the value is copied, if it has one.

For all other accesses to an object having no declared type, the effective type of the object is simply the type of the lvalue used for the access.

Effective Types 1

```
 struct st {
   char c; int i; long l; double d;
} s = { 1, 2, 3, 4 };

char *p = malloc(sizeof s); assert(p);  // none at all
char *p0 = malloc(sizeof s); assert(p0);
memcpy(p0, &s, sizeof (s));  // struct st

void *p1 = malloc(sizeof s); assert(p1);
memcpy(p1, &s, sizeof (s));  // struct st
memcpy(p1, &s.i, sizeof (int));  // int
memcpy(p1, &s.i, sizeof (int)); // int
memcpy(p1, (float *) &s.i, sizeof (int)); // float
```

Effective Types 2

```
void *p2 = malloc(sizeof s); assert(p2);
memcpy(p2, (void *)&s, sizeof (s));  // struct st

void *p3 = malloc(sizeof s); assert(p3);
memcpy(p3, (char *)&s, sizeof (s));  // struct st

void *p4 = malloc(sizeof s); assert(p4);
*(struct st *)p4 = s; // struct st
```

Aliasing Rules

One pointer is an alias of another pointer when both refer to the same location or object.

An object's stored value can only be accessed by an lvalue expression that has:

- a type compatible with the effective type of the object,
- a qualified version of a type compatible with the effective type of the object,
- a type that is the signed or unsigned type corresponding to the effective type of the object,
- a type that is the signed or unsigned type corresponding to a qualified version of the effective type of the object,
- an aggregate or union type that includes one of the aforementioned types among its members (including, recursively, a member of a subaggregate or contained union), or
- a character type.

Aliasing Rules

In `f1`, `*pi` can be assumed to be loop-invariant, because the type-based aliasing rules don't allow `*pd` to be an alias for `*pi`.

```
void f1(int *pi, double *pd, double d) {
  for (int i = 0; i < *pi; i++) {
    *pd++ = d;
  }
}
```

Therefore, it is valid to transform the loop such that `*pi` is loaded only once, at the top of the loop.

Aliasing Rules

In `f2`, `*pi` cannot be assumed to be loop-invariant.

```
struct S { int a, b; };
void f2(int *pi, struct S *ps, struct S s) {
  for (int i = 0; i < *pi; i++) {
    *ps++ = s;
  }
}
```

`*pi` (having type `int`) can be modified by an lvalue that has aggregate type including a member of type `int`.

`struct S` is such a type, and the lvalue `*ps` has that type.

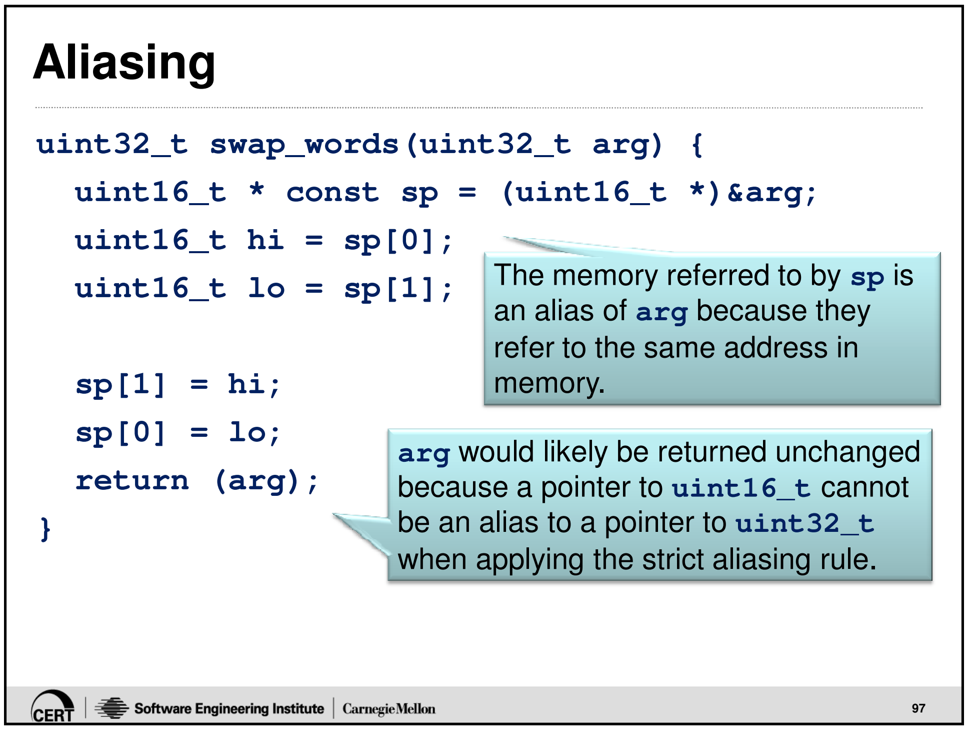
Aliasing Rules

Therefore, `f2` could be called as follows:

```
struct S a[10] = { [9].b = 1000 };
f2(&a[9].b, a, (struct S){ 0, 0 });
```

Despite the fact that the initial value of `*pi` in the loop is too large for the array, this usage would nevertheless be safe.

`*pi` would have to be reloaded for each iteration, and on the tenth iteration, the value of `*pi` would become zero, the loop would terminate before the loop writes past the end of the array.

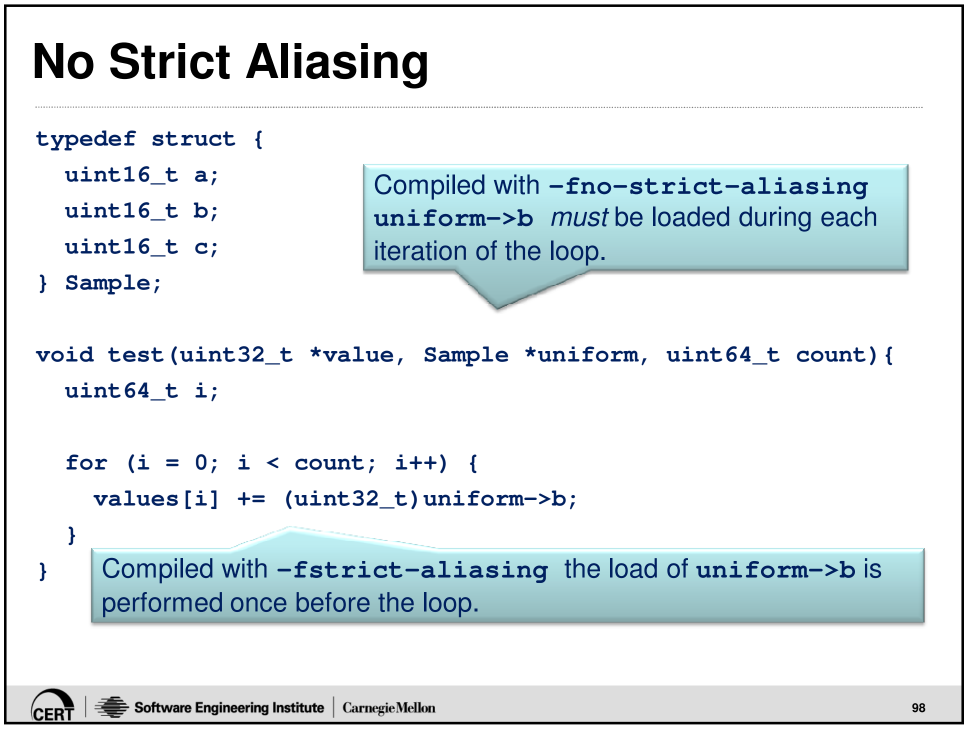


Aliasing

```
uint32_t swap_words(uint32_t arg) {
  uint16_t * const sp = (uint16_t *)&arg;
  uint16_t hi = sp[0];
  uint16_t lo = sp[1];

  sp[1] = hi;
  sp[0] = lo;
  return (arg);
}
```

The memory referred to by `sp` is an alias of `arg` because they refer to the same address in memory.

`arg` would likely be returned unchanged because a pointer to `uint16_t` cannot be an alias to a pointer to `uint32_t` when applying the strict aliasing rule.

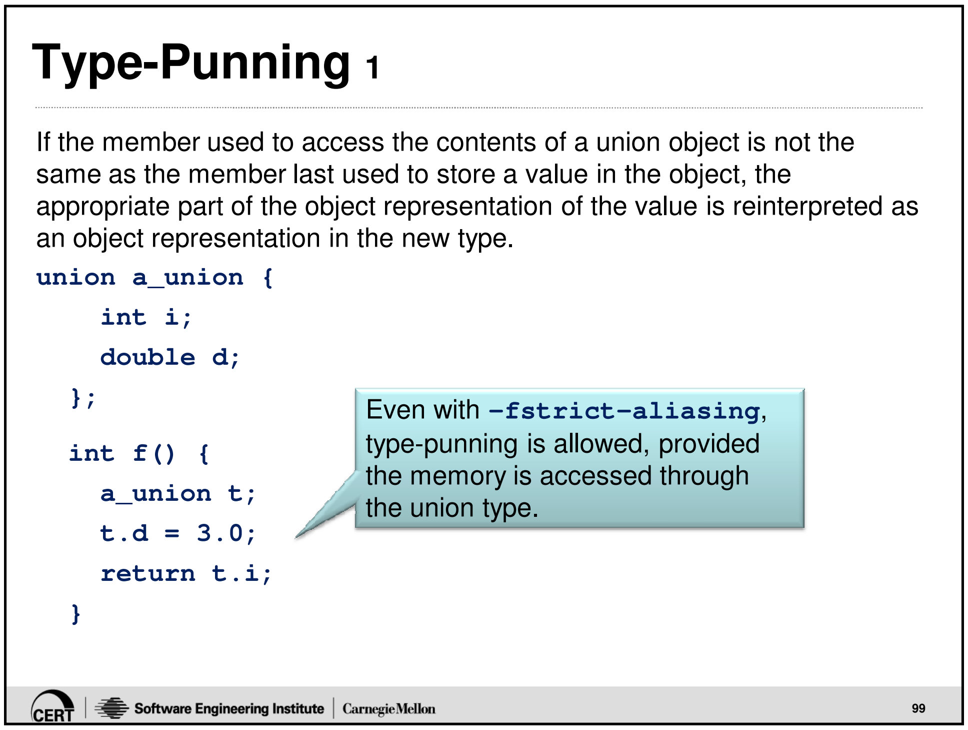


No Strict Aliasing

```
typedef struct {
  uint16_t a;
  uint16_t b;
  uint16_t c;
} Sample;

void test(uint32_t *value, Sample *uniform, uint64_t count){
  uint64_t i;

  for (i = 0; i < count; i++) {
    values[i] += (uint32_t)uniform->b;
  }
}
```

Compiled with `-fno-strict-aliasing` `uniform->b` *must* be loaded during each iteration of the loop.

Compiled with `-fstrict-aliasing` the load of `uniform->b` is performed once before the loop.

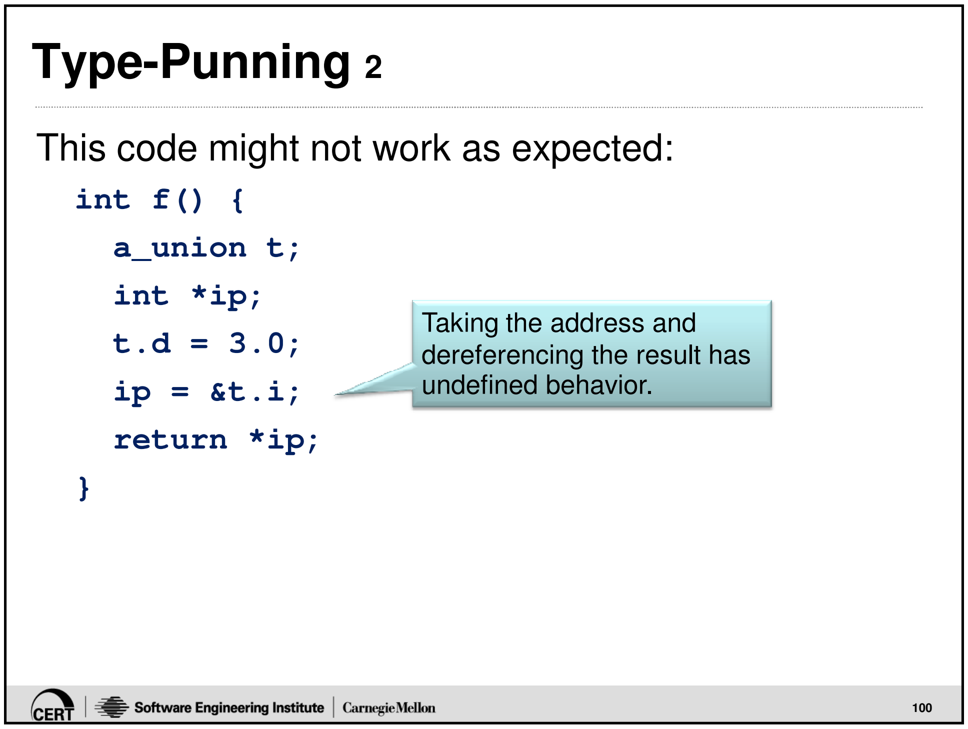


Type-Punning 1

If the member used to access the contents of a union object is not the same as the member last used to store a value in the object, the appropriate part of the object representation of the value is reinterpreted as an object representation in the new type.

```
union a_union {
    int i;
    double d;
  };

  int f() {
    a_union t;
    t.d = 3.0;
    return t.i;
  }
```

Even with `-fstrict-aliasing`, type-punning is allowed, provided the memory is accessed through the union type.



Type-Punning 2

This code might not work as expected:

```
  int f() {
    a_union t;
    int *ip;
    t.d = 3.0;
    ip = &t.i;
    return *ip;
  }
```

Taking the address and dereferencing the result has undefined behavior.

Type-Punning 3

Access by taking the address, casting the resulting pointer and dereferencing the result has undefined behavior, even if the cast uses a union type.

```
int f() {
  double d = 3.0;
  return ((union a_union *) &d)->i;
}
```

Agenda

Undefined Behavior
Compiler Optimizations
Constant Folding
Adding a Pointer and an Integer
Integer Overflow
GCC Options
Clearing Sensitive Information
Strict Aliasing
Optimization Suggestions
C1X Analyzability Annex
Summary and Recommendations

inline, register, restrict

C99 defines several keywords which have no other consequence than to influence optimization

inline

- suggests that calls to the function be as fast as possible.
- the extent to which such suggestions are effective is implementation-defined.

register

- suggests that access to the object be as fast as possible.
- the extent to which such suggestions are effective is implementation-defined.

restrict

An object that is accessed through a restrict-qualified pointer has a special association with that pointer.

This association requires that all accesses to that object use, directly or indirectly, the value of that particular pointer.

The intended use of the restrict qualifier is to promote optimization, and deleting all instances of the qualifier from all preprocessing translation units composing a conforming program does not change its meaning (i.e., observable behavior).

restrict

The restrict qualifier addresses the problem that potential aliasing can inhibit optimizations.

Specifically, if a translator cannot determine that two different pointers are being used to reference different objects, then it cannot apply optimizations such as

- maintaining the values of the objects in registers rather than in memory
- reordering loads and stores of these values.

noreturn Function Attribute

You can declare a function as **noreturn** to inform the compiler the function never returns.

```
void fatal () __attribute__ ((noreturn));
void fatal (/* ... */) {
  /* Print error message. */
  exit (1);
 }
```

The **noreturn** keyword tells the compiler to assume that fatal cannot return.

It can then optimize without regard to what would happen if fatal ever did return.

Optimization Suggestions

Optimizations may be performed without any suggestions from the programmer.

Some suggestions like `register` and `inline` may be ignored because the compiler can usually do a better job.

Some suggestions like `noreturn` may be followed even if they are obviously wrong.

Analysis is usually better than suggestions because there is less chance for error.

Agenda

Undefined Behavior

Compiler Optimizations

Constant Folding

Adding a Pointer and an Integer

Integer Overflow

GCC Options

Clearing Sensitive Information

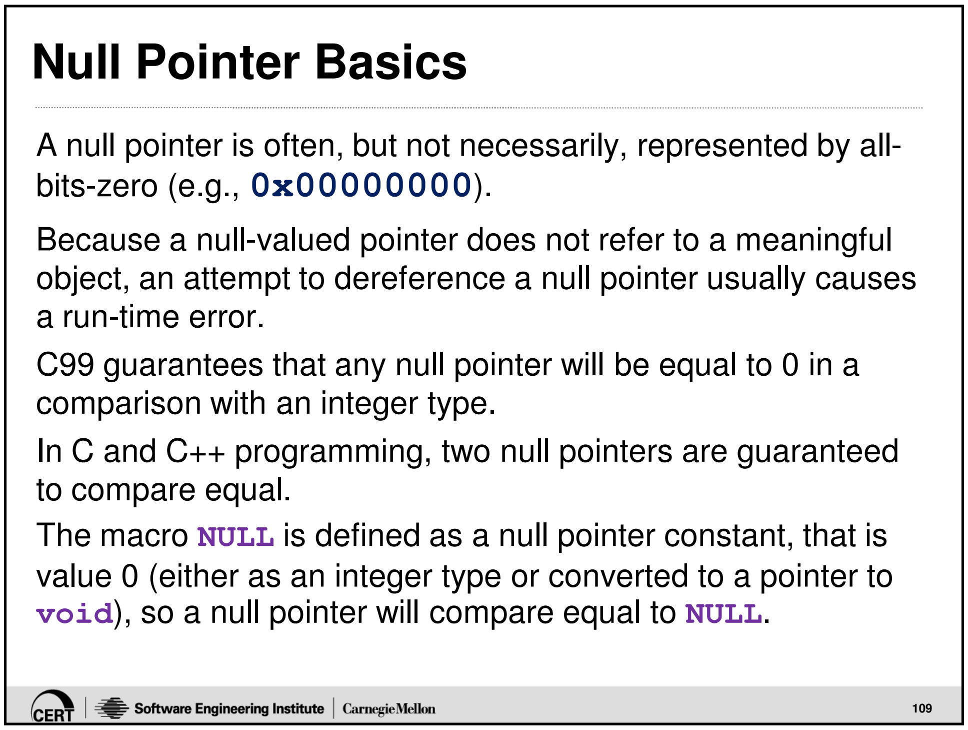
Strict Aliasing

Optimization Suggestions

Null-pointer dereference

C1X Analyzability Annex

Summary and Recommendations



Null Pointer Basics

A null pointer is often, but not necessarily, represented by all-bits-zero (e.g., `0x00000000`).

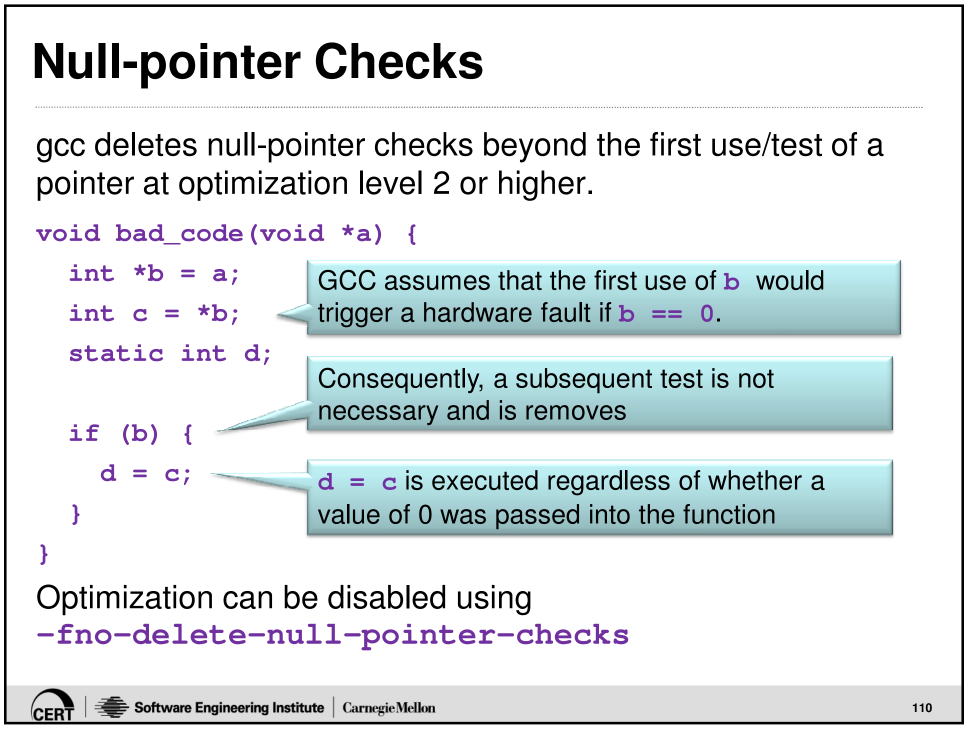
Because a null-valued pointer does not refer to a meaningful object, an attempt to dereference a null pointer usually causes a run-time error.

C99 guarantees that any null pointer will be equal to 0 in a comparison with an integer type.

In C and C++ programming, two null pointers are guaranteed to compare equal.

The macro `NULL` is defined as a null pointer constant, that is value 0 (either as an integer type or converted to a pointer to `void`), so a null pointer will compare equal to `NULL`.

CERT | Software Engineering Institute | Carnegie Mellon 109



Null-pointer Checks

gcc deletes null-pointer checks beyond the first use/test of a pointer at optimization level 2 or higher.

```
void bad_code(void *a) {
  int *b = a;
  int c = *b;
  static int d;

  if (b) {
    d = c;
  }
}
```

GCC assumes that the first use of `b` would trigger a hardware fault if `b == 0`.

Consequently, a subsequent test is not necessary and is removes

`d = c` is executed regardless of whether a value of 0 was passed into the function

Optimization can be disabled using `-fno-delete-null-pointer-checks`

CERT | Software Engineering Institute | Carnegie Mellon 110

From Linux kernel 2.6.30

```
static unsigned int tun_chr_poll(struct file *file,
poll_table * wait)  {
  struct tun_file *tfile = file->private_data;
  struct tun_struct *tun = __tun_get(tfile);
  struct sock *sk = tun->sk;
  unsigned int mask = 0;

  if (!tun) return POLLERR;
  DBG(KERN_INFO "%s: tun_chr_poll\n", tun->dev->name);
  /* … */
}
```

`sk` initialized to `tun->sk`

Checks if `tun == NULL`

GCC optimization removes the `if (!tun)` check because it is performed after the assignment.

Why is this a Problem?

The Linux kernel could be locally exploited (before Linux 2.6.23) by

- mapping page zero with `mmap()` and crafting it with malicious instructions
- trigger the bug in the process' context.

Because the kernel's data and code segment both have a base of zero, a null pointer dereference makes the kernel access page zero, a page filled with bytes the attacker controls.

For targets such as the ARM7 that do not have a hardware memory manager, null pointer dereferences cause silent failures and exploitable vulnerabilities.

Linux Kernel Patch

```
- struct sock *sk = tun->sk;
+ struct sock *sk;
  unsigned int mask = 0;

  if (!tun)
      return POLLERR;

+ sk = tun->sk;
+
  DBG(KERN_INFO "%s: tun_chr_poll\n", tun->dev->name);
```

Agenda

Undefined Behavior
Compiler Optimizations
Constant Folding
Adding a Pointer and an Integer
Integer Overflow
GCC Options
Clearing Sensitive Information
Strict Aliasing
Optimization Suggestions
C1X Analyzability Annex
Summary and Recommendations

C1X Analyzability Annex

This annex specifies optional behavior that can aid in the analyzability of C programs.

An implementation that defines `_ _STDC_ANALYZABLE_ _` shall conform to the specifications in this annex.

Definitions

out-of-bounds store: an (attempted) access (3.1) that, at run time, for a given computational state, would modify (or, for an object declared volatile, fetch) one or more bytes that lie outside the bounds permitted by this Standard.

bounded undefined behavior: undefined behavior (3.4.3) that does not perform an out-of-bounds store.

NOTE 1 The behavior might perform a trap.

NOTE 2 Any values produced or stored might be indeterminate values.

critical undefined behavior: undefined behavior that is not bounded undefined behavior.

NOTE The behavior might perform an out-of-bounds store or perform a trap.

Requirements

If the program performs a trap (3.19.5), the implementation is permitted to invoke a runtime-constraint handler. Any such semantics are implementation-defined.

All undefined behavior shall be limited to bounded undefined behavior, except for the following which are permitted to result in critical undefined behavior.

Critical Undefined Behaviors

An object is referred to outside of its lifetime (6.2.4).

An lvalue does not designate an object when evaluated (6.3.2.1).

A pointer is used to call a function whose type is not compatible with the pointed-to type (6.3.2.3).

The operand of the unary * operator has an invalid value (6.5.3.2).

Addition or subtraction of a pointer into, or just beyond, an array object and an integer type produces a result that points just beyond the array object and is used as the operand of a unary * operator that is evaluated (6.5.6).

An argument to a library function has an invalid value or a type not expected by a function with variable number of arguments (7.1.4).

The value of a pointer that refers to space deallocated by a call to the free or realloc function is used (7.21.3).

A string or wide string utility function is instructed to access an array beyond the end of an object (7.22.1, 7.27.4).

Agenda

Undefined Behavior
Compiler Optimizations
Constant Folding
Adding a Pointer and an Integer
Integer Overflow
GCC Options
Clearing Sensitive Information
Strict Aliasing
Optimization Suggestions
C1X Analyzability Annex
Summary and Recommendations

Recommendations

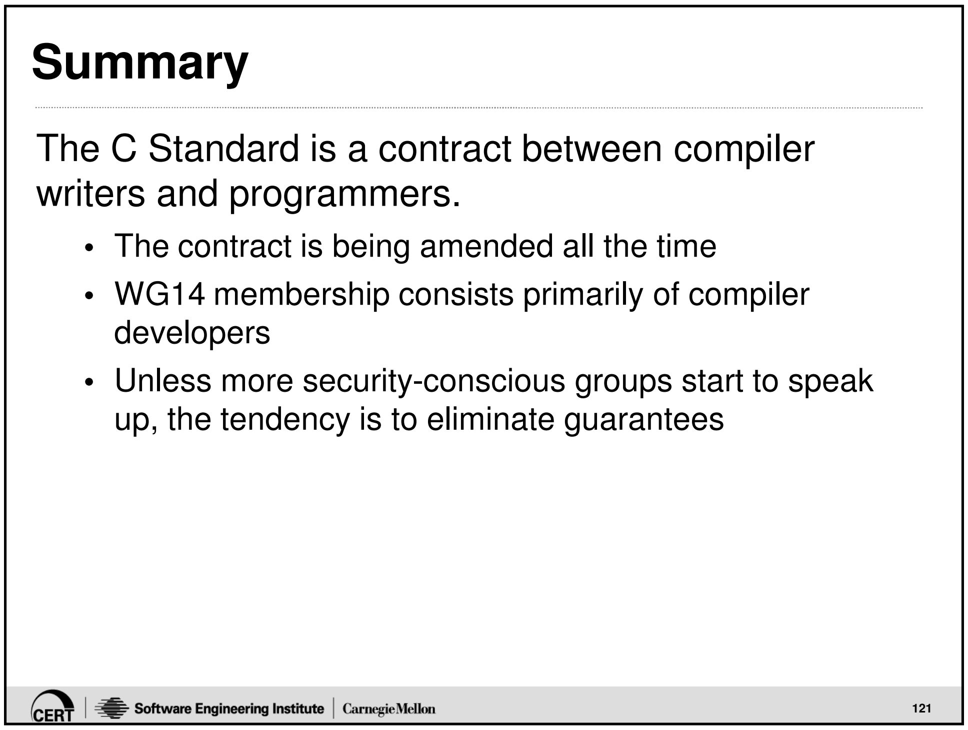
Avoid undefined behaviors in your code, even if your code appears to be working (for the time being).

Find and eliminate dead code yourself instead of letting the compiler do it.

Some optimizations may eliminate undefined behaviors, while others may introduce vulnerabilities.

Go go ahead and compile at `-O2`

- Use compiler diagnostics such as `-Wstrict-overflow` to determine if the compiler is optimizing based assumptions that are different than your own.
- In many cases, you can rewrite the source code to more closely resemble the optimized code and eliminate these warnings



Summary

The C Standard is a contract between compiler writers and programmers.

- The contract is being amended all the time
- WG14 membership consists primarily of compiler developers
- Unless more security-conscious groups start to speak up, the tendency is to eliminate guarantees

CERT | Software Engineering Institute | Carnegie Mellon 121