Integer overflow detection using gcc/clang or g++/ clang++.

gcc/g++

Integer overflows occur when the result of an arithmetic operation on integers exceeds the maximum value that can be represented by the data type, leading to unexpected and potentially dangerous behavior.

To detect integer overflows using GCC and G++ compilers, you can use the -ftrapv flag, which enables the trapping of integer overflow errors at runtime. This flag instructs the compiler to generate code that checks for integer overflows and raises an exception or signal when such an error occurs.

Here's an example of how to use the -ftrapv flag with GCC and G++ compilers:$ gcc -ftrapv -o program program.c

This command compiles the C source file program.c with the -ftrapv flag and generates an executable program.

Similarly, for C++ programs compiled with G++ compiler, you can use the same flag:$ g++ -ftrapv -o program program.cpp

Note that enabling the -ftrapv flag may affect the performance of your program and may not catch all possible integer overflow errors. It's also important to ensure that your code uses appropriate data types and handles arithmetic operations correctly to avoid integer overflow errors.

clang/clang++

First, install clang/clang++ compiler and use the -fsanitize option while compiling the program.

Write a program (your choice) with integer overflow and use the above options for detecting the overflow.

Clang provides a number of ways to control code generation. The options are listed below.

-f[no-]sanitize=check1,check2,...

Turn on runtime checks for various forms of undefined or suspicious behavior.

This option controls whether Clang adds runtime checks for various forms of undefined or suspicious behavior, and is disabled by default. If a check fails, a diagnostic message is produced at runtime explaining the problem. The main checks are:

- `-fsanitize=address`: **AddressSanitizer**, a memory error detector.
- `-fsanitize=thread`: **ThreadSanitizer**, a data race detector.
- `-fsanitize=memory`: **MemorySanitizer**, a detector of uninitialized reads. Requires instrumentation of all program code.
- `-fsanitize=undefined`: **UndefinedBehaviorSanitizer**, a fast and compatible undefined behavior checker.
- `-fsanitize=dataflow`: **DataFlowSanitizer**, a general data flow analysis.
- `-fsanitize=cfi`: **control flow integrity** checks. Requires `-flto`.
- `-fsanitize=kcfi`: kernel indirect call forward-edge control flow integrity.
- `-fsanitize=safe-stack`: **safe stack** protection against stack-based memory corruption errors.

Available checks are:

- `-fsanitize=alignment`: Use of a misaligned pointer or creation of a misaligned reference. Also sanitizes assume_aligned-like attributes.
- `-fsanitize=bool`: Load of a `bool` value which is neither `true` nor `false`.
- `-fsanitize=builtin`: Passing invalid values to compiler builtins.
- `-fsanitize=bounds`: Out of bounds array indexing, in cases where the array bound can be statically determined. The check includes `-fsanitize=array-bounds` and `-fsanitize=local-bounds`. Note that `-fsanitize=local-bounds` is not included in `-fsanitize=undefined`.
- `-fsanitize=enum`: Load of a value of an enumerated type which is not in the range of representable values for that enumerated type.
- `-fsanitize=float-cast-overflow`: Conversion to, from, or between floating-point types which would overflow the destination. Because the range of representable values for all floating-point types supported by Clang is [-inf, +inf], the only cases detected are conversions from floating point to integer types.
- `-fsanitize=float-divide-by-zero`: Floating point division by zero. This is undefined per the C and C++ standards, but is defined by Clang (and by ISO/IEC/IEEE 60559 / IEEE 754) as producing either an infinity or NaN value, so is not included in `-fsanitize=undefined`.
- `-fsanitize=function`: Indirect call of a function through a function pointer of the wrong type.
- `-fsanitize=implicit-unsigned-integer-truncation`, `-fsanitize=implicit-signed-integer-truncation`: Implicit conversion from integer of larger bit width to smaller bit width, if that results in data loss. That is, if the demoted value, after casting back to the original width, is not equal to the original value before the downcast. The `-fsanitize=implicit-unsigned-integer-truncation` handles conversions between two `unsigned` types, while `-fsanitize=implicit-signed-integer-truncation` handles the rest of the conversions - when either one, or both of the types are signed. Issues caught by these sanitizers are not undefined behavior, but are often unintentional.
- `-fsanitize=implicit-integer-sign-change`: Implicit conversion between integer types, if that changes the sign of the value. That is, if the original value was negative and the new value is positive (or zero), or the original value was positive, and the new value is negative. Issues caught by this sanitizer are not undefined behavior, but are often unintentional.
- `-fsanitize=integer-divide-by-zero`: Integer division by zero.
- `-fsanitize=nonnull-attribute`: Passing null pointer as a function parameter which is declared to never be null.
- `-fsanitize=null`: Use of a null pointer or creation of a null reference.
- `-fsanitize=nullability-arg`: Passing null as a function parameter which is annotated with `_Nonnull`.
- `-fsanitize=nullability-assign`: Assigning null to an lvalue which is annotated with `_Nonnull`.
- `-fsanitize=nullability-return`: Returning null from a function with a return type annotated with `_Nonnull`.
- `-fsanitize=objc-cast`: Invalid implicit cast of an ObjC object pointer to an incompatible type. This is often unintentional, but is not undefined behavior, therefore the check is not a part of the `undefined` group. Currently only supported on Darwin.
- `-fsanitize=object-size`: An attempt to potentially use bytes which the optimizer can determine are not part of the object being accessed. This will also detect some types of undefined behavior that may not directly access memory, but are provably incorrect given the size of the objects involved, such as invalid downcasts and calling methods on invalid pointers. These checks are made in terms of `__builtin_object_size`, and consequently may be able to detect more problems at higher optimization levels.
- `-fsanitize=pointer-overflow`: Performing pointer arithmetic which overflows, or where either the old or new pointer value is a null pointer (or in C, when they both are).

-fsanitize=return: In C++, reaching the end of a value-returning function without returning a value.

-fsanitize=returns-nonnull-attribute: Returning null pointer from a function which is declared to never return null.

-fsanitize=shift: Shift operators where the amount shifted is greater or equal to the promoted bit-width of the left hand side or less than zero, or where the left hand side is negative. For a signed left shift, also checks for signed overflow in C, and for unsigned overflow in C++. You can use -fsanitize=shift-base or -fsanitize=shift-exponent to check only left-hand side or right-hand side of shift operation, respectively.

-fsanitize=unsigned-shift-base: check that an unsigned left-hand side of a left shift operation doesn't overflow. Issues caught by this sanitizer are not undefined behavior, but are often unintentional.

-fsanitize=signed-integer-overflow: Signed integer overflow, where the result of a signed integer computation cannot be represented in its type. This includes all the checks covered by -ftrapv, as well as checks for signed division overflow (INT_MIN/-1), but not checks for lossy implicit conversions performed before the computation (see -fsanitize=implicit-conversion). Both of these two issues are handled by -fsanitize=implicit-conversion group of checks.

-fsanitize=unreachable: If control flow reaches an unreachable program point.

-fsanitize=unsigned-integer-overflow: Unsigned integer overflow, where the result of an unsigned integer computation cannot be represented in its type. Unlike signed integer overflow, this is not undefined behavior, but it is often unintentional. This sanitizer does not check for lossy implicit conversions performed before such a computation (see -fsanitize=implicit-conversion).

-fsanitize=vla-bound: A variable-length array whose bound does not evaluate to a positive value.

-fsanitize=vptr: Use of an object whose vptr indicates that it is of the wrong dynamic type, or that its lifetime has not begun or has ended. Incompatible with -fno-rtti. Link must be performed by clang++, not clang, to make sure C++-specific parts of the runtime library and C++ standard libraries are present.

You can also use the following check groups:

-fsanitize=undefined: All of the checks listed above other than float-divide-by-zero, unsigned-integer-overflow, implicit-conversion, local-bounds and the nullability-* group of checks.

-fsanitize=undefined-trap: Deprecated alias of -fsanitize=undefined.

-fsanitize=implicit-integer-truncation: Catches lossy integral conversions. Enables implicit-signed-integer-truncation and implicit-unsigned-integer-truncation.

-fsanitize=implicit-integer-arithmetic-value-change: Catches implicit conversions that change the arithmetic value of the integer. Enables implicit-signed-integer-truncation and implicit-integer-sign-change.

-fsanitize=implicit-conversion: Checks for suspicious behavior of implicit conversions. Enables implicit-unsigned-integer-truncation, implicit-signed-integer-truncation, and implicit-integer-sign-change.

-fsanitize=integer: Checks for undefined or suspicious integer behavior (e.g. unsigned integer overflow). Enables signed-integer-overflow, unsigned-integer-overflow, shift, integer-divide-by-zero, implicit-unsigned-integer-truncation, implicit-signed-integer-truncation, and implicit-integer-sign-change.

-fsanitize=nullability: Enables nullability-arg, nullability-assign, and nullability-return. While violating nullability does not have undefined behavior, it is often unintentional, so UBSan offers to catch it.

https://clang.llvm.org/docs/UsersManual.html#controlling-static-analyzer-diagnostics (clang reference)

Mitigation

https://learn.microsoft.com/en-us/cpp/safeint/safeint-class?view=msvc-170 (SafeInt reference)

Write a program to implement a calculator using SafeInt operations..