インテル® oneAPI DPC++/C++ コンパイラーでサニタイザーを使用してバグを素早く検出

この記事は、インテルのサイトで公開されている「Find Bugs Quickly Using Sanitizers with the Intel® oneAPI DPC++/C++ Compiler」の日本語参考訳です。原文は更新される可能性があります。原文と翻訳文の内容が異なる場合は原文を優先してください。

このチュートリアルでは、インテル® oneAPI DPC++/C++ コンパイラーでサニタイザーを使用して、C/C++ コードの一般的な問題を効率良く正確に検出する方法を説明します。

注: ホスト CPU とコンピュート・アクセラレーター・ターゲット・デバイス上で実行するように設計された C++ with SYCL* プログラムは、現在サポートされていません。

サニタイザーは、コード内の望ましくない動作や未定義の動作を特定し、ピンポイントで検出するのに役立ちます。サニタイザーは、プログラムをインストルメントするコンパイラー・オプションで有効になり、バイナリーの安全チェックを提供します。

副作用として、パフォーマンスとファイルサイズのオーバーヘッドが発生します。パフォーマンスと実行可能ファイルサイズへの影響は、使用するサニタイザー・ツールと解析するプログラムの特性によって異なります。プログラムのサイズ、割り当てられたメモリー量、使用されるスレッド数などの要因が影響します。したがって、サニタイザーはデバッグとコード検証にのみ使用し、製品レベルのコードには使用すべきではありません。

サニタイザーを使用する個々のビルド手順は、バグやセキュリティーの脆弱性の検出と防止に役立つため、大きなメリットをもたらします。これらの使用は、CI/CD DevOps 環境での定期的なソフトウェア・テストに不可欠です。

また、ソフトウェア開発者がコード変更をリポジトリー・ブランチに送信する前に検証する便利な方法も用意されています。

実際、Clang やインテル® oneAPI DPC++/C++ コンパイラーなどの LLVM ベースのコンパイラーで使用されるサニタイザーは、かなり軽量です。これは、Valgrind* (英語) のようなオープンソース・ソフトウェア・テスト・ソリューションや、Parasoft の Insure++*、PVS Studio*、AbsInt Astrée*、QA Systems Cantata* などの機能テストおよびコーディング標準準拠のための商用コード・アナライザー・ソリューションと比較した場合に特に顕著です。通常、サニタイザーは実行時間を 2～3 倍に増やしますが、Valgrind* では最大 100 倍のオーバーヘッドが発生する可能性があります。

そのため、サニタイザーは、通常のソフトウェア開発フローの一部としてプログラムをテストまたはデバッグする場合や、大規模なアプリケーションの実行の後半で発生するランタイムの問題を特定する場合に有効です。

しかし、GDB* など、より伝統的な対話型デバッグアプローチと比較すると、1 つの欠点があります。サニタイザーを使用するには、プログラムの再コンパイルが必要です。プログラムがほかの共有ライブラリーに依存している場合、これらもサニタイザーを有効にして再コンパイルするのが理想的です (もちろん、標準の libc/libc++ は除きます)。そうすることで、コード・インストルメンテーションが代わりにバグを探してくれます。

このチュートリアルでは、次のサニタイザーについて詳しく説明します。

1. **AddressSanitizer** - メモリーの安全性に関するバグを検出します。
2. **UndefinedBehaviourSanitizer** - 未定義の動作に関するバグを検出します。
3. **MemorySanitizer** - 初期化されていないメモリーの使用に関するバグを検出します。

このチュートリアルで使用するサンプル・ソースコードは、sanitizers-tutorial.tgz (英語) アーカイブファイルにあります。

1. AddressSanitizer によるメモリーの安全性に関するバグの検出

サニタイザーのさまざまな機能を実証するため、フィボナッチ数列を出力する小さなプログラムを使用します。この数列では、0、1、1、2、3、5、8、13、21、34 … のように、各数値は前の 2 つの数値の合計になります。

サンプルコード

チュートリアルのソースアーカイブにある `fibonacci_v1.c` サンプルコードから開始します。

```
#include <stdlib.h>
#include <stdio.h>

/**
 * 長さ n の配列 arr に最初の n 個の
 * フィボナッチ数を格納
 */
void set_fibonacci_list(int *arr, int n) {
  arr[0] = 0;
  arr[1] = 1;

  for (int i = 2; i < n; i++) {
    arr[i] = arr[i-1] + arr[i-2];
  }
}

/**
 * 最初の n 個のフィボナッチ数を出力
 * */
void print_fibonacci(int n) {
  int fibos[n];

  set_fibonacci_list(fibos, n);

  printf("Fibonacci Sequence\n");
  printf("==================\n");
  for (int i = 0; i < n; i++) {
    printf("%d\n", fibos[i]);
  }
  printf("==================\n");
  if (n > 1 && fibos[n-2] != 0) {
    printf("Golden ratio approximation: %g\n", ((double)fibos[n-1])/fibos[n-2]);
  }
}

int main(int argc, char *argv[]) {
```

```
  if (argc != 2) {
    printf("Usage: %s NUM\n", argv[0]);
    return 1;
  }

  print_fibonacci(atoi(argv[1]));

  return 0;
}
```

図 1. 初期のフィボナッチ数列のソースコード例

このプログラムは表示するフィボナッチ数をコマンドライン・パラメーターとして受け取ります。そして、`set_fibonacci_list` 関数でフィボナッチ数列を計算し、`print_fibonacci` 関数で結果を画面に出力します。

サニタイザーの実行

次に、AddressSantizer を使用して、このプログラム内の潜在的なメモリー関連のバグを検出します。AddressSanitizer は、スタックとヒープ上の境界外アクセスや解放後の使用など、メモリーの安全性に関する複数のバグを検出できます。

AddressSantizer を使用してプログラムをコンパイルするには、次のコマンドを使用します。

```
$ icx src/fibonacci_v1.c -O0 -g -fsanitize=address -fno-omit-frame-pointer -o
fibonacci_v1_with_asan
```

`-fsanitize=address` コンパイラー・オプションは、サニタイザーを有効にします。

`-O0 -g -fno-omit-frame-pointer` コンパイラー・オプションは、実際にコーディングの問題が見つかった場合に適切な診断出力が得られるように追加します。これらのオプションは必須ではありません。

`-g` は、暗黙的に `-O0` と `-fno-omit-frame-pointer` を設定することに注意してください。ここでは、パラメーターの完全なセットを明示するために列挙しています。

コマンドラインに渡すことができるサニタイザー関連のオプションはほかにもあります。一覧は、「コンパイラー・ユーザーズ・マニュアル」(英語) を参照してください。

比較のため、サニタイザーなしのバージョンをコンパイルすることもできます。

```
$ icx src/fibonacci_v1.c -O0 -g -o fibonacci_v1
```

N に任意の値を設定して両方の実行可能ファイルを実行すると、同じ出力になるはずです。

```
$ ./fibonacci_v1 10
Fibonacci Sequence
====================================
0
1
1
2
3
5
8
13
21
34
====================================
Golden ratio approximation: 1.61905
$ ./fibonacci_v1_with_asan 10
Fibonacci Sequence
====================================
0
1
1
2
3
5
8
13
21
34
====================================
Golden ratio approximation: 1.61905
```

しかし、このプログラムにはバグがあります。

`n < 2` の場合、`set_fibonacci_list` 関数で、フィボナッチの初期値が範囲外のインデックスに割り当てられます。

引数に 0 を指定してプログラムを実行し、何が起こるか見てみましょう。

```
$ ./fibonacci_v1 0
Fibonacci Sequence
==================
$ ./fibonacci_v1_with_asan 0
=================================================
==9006==ERROR: AddressSanitizer: dynamic-stack-buffer-overflow on address
0x7ffd21da5a20 at pc 0x000000506601 bp 0x7ffd21da5990 sp 0x7ffd21da5988
...
```

これは、AddressSanitizer がいかに強力であるかを示しています。この例では、通常のプログラム実行は失敗しなかったため、バグを見逃してしまったかもしれません。ほかの構成では、プログラムがクラッシュしたり、もっと後の時点でクラッシュが発生する可能性もあります。最悪の場合、プログラムはクラッシュせずに、間違った結果が出力されるかもしれません。

AddressSantitizer は、エラーを即座に検出して実行を中止し、詳細な診断レポートを表示します。レポートには、以下の情報が含まれます。

- バグの種類、場所、レジスター値

```
dynamic-stack-buffer-overflow on address 0x7ffd21da5a20 at pc
0x000000506601 bp 0x7ffd21da5990 sp 0x7ffd21da5988
```

- バグが発生した場所のトレースバック (これには、デバッグ・コンパイラー・オプションが役立ちます)

```
WRITE of size 8 at 0x7ffd21da5a20 thread T0
    #0 0x506600 in set_fibonacci_list
/home/user/sanitizer_tutorial/src/Fibonacci_v1.c:10:10
    #1 0x5067c1 in print_fibonacci
/home/user/sanitizer_tutorial/src/fibonacci_v1.c:24:3
    #2 0x50691e in main
/home/user/sanitizer_tutorial/src/fibonacci_v1.c:42:3
    #3 0x7feff05077b2 in __libc_start_main (/lib64/libc.so.6+0x237b2)
(BuildId: ade58d86662aceee2210a9ef12018705e978965d)
    #4 0x41eb2d in _start
(/home/user/sanitizer_tutorial/fibonacci_v1_with_asan+0x41eb2d)
```

この例では、Valgrind* はバグを検出しませんでした、これは、`Valgrind` がスタックベースのバッファー・オーバーフローを検出しないためです。

```
valgrind ./fibonacci_v1 0
==166581== Memcheck, a memory error detector
==166581== Copyright (C) 2002-2022, and GNU GPL&apos;d, by Julian Seward et al.
==166581== Using Valgrind-3.19.0 and LibVEX; rerun with -h for copyright info
==166581== Command: ./Fibonacci_v1 0
==166581==
Fibonacci Sequence
==================
==================
==166581==
==166581== HEAP SUMMARY:
==166581==     in use at exit: 0 bytes in 0 blocks
==166581==   total heap usage: 1 allocs, 1 frees, 1,024 bytes allocated
==166581==
==166581== All heap blocks were freed -- no leaks are possible
==166581==
==166581== For lists of detected and suppressed errors, rerun with: -s
==166581== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
```

問題の修正

`set_fibonacci_list` 関数を調整して、検出されたメモリーの安全性に関するバグを修正しましょう。

```
void set_fibonacci_list(int *arr, int n) {
  if (n > 1) {
    arr[0] = 0;
  }
  if (n > 2) {
    arr[1] = 1;
  }
  for (int i = 2; i < n; i++) {
    arr[i] = arr[i-1] + arr[i-2];
  }
}
```

図 2. n<2 の場合の set_fibonacci_list の修正

更新後のプログラムは、チュートリアル・ソース・アーカイブ内の `src/fibonacci_v2.c` にあります。

再コンパイル後、サニタイザーが有効なバージョンを再実行することでバグがなくなったことを確認できます。

```
$ ./fibonacci_v2_with_asan 0
Fibonacci Sequence
====================================
```

これで、プログラムの修正は完了です。

2. UndefinedBehaviorSanitizer (UBSan) による未定義の動作に関するバグの検出

フィボナッチ・プログラムのメモリー関連のバグを修正したら、プログラムを使って手動で基本的な機能テストを実行できます。

問題の観察

例えば、N に大きな値を使用してみます。

```
$ ./fibonacci_v2_with_asan 100
Fibonacci Sequence
==================
0
1
...
-889489150
==================
Golden ratio approximation: 9.81579
```

出力が間違っていることが分かります。黄金比 (Golden Ratio) が大きくずれており、負のフィボナッチ数が表示されています (フィボナッチ数が負になることはありません)。

サニタイザーの実行

原因を調べるため、プログラム内の未定義の動作の種類を検出できる `UndefinedBehaviorSanitizer (UBSan)` サニタイザーを使用します。

```
$ icx src/fibonacci_v2.c -O0 -g -fsanitize=undefined -fno-omit-frame-pointer -o
fibonacci_v2_with_ubsan
```

`-fsanitize=undefined` を使用して UBSan を有効にします。UBSan は、一般的な種類の未定義の動作を検出します。ほかの種類の未定義の動作の検出を有効にするには、UndefinedBehaviorSanitizer のドキュメント (英語) を参照してください。

サニタイザーが有効なバイナリーを実行します。

```
$ ./fibonacci_v2_with_ubsan 100
src/Fibonacci_v2.c:17:23: runtime error: signed integer overflow: 1836311903 +
1134903170 cannot be represented in type &apos;int&apos;
SUMMARY: UndefinedBehaviorSanitizer: undefined-behavior src/Fibonacci_v2.c:17:23
in
Fibonacci Sequence
==================
0
...
```

原因の特定

UBSan により問題が特定されました。フィボナッチ数列が大きくなり、C 標準では未定義の動作である、符号付き整数のオーバーフローが発生したことが原因です。

AddressSanitizer と同様に、詳細な診断が出力されます。

- 未定義の動作の種類 (`signed integer overflow` - 符号付き整数のオーバーフロー)
- 問題に関する追加情報 (`1836311903 + 1134903170 cannot be represented in type &apos;int&apos;` - 1836311903 + 1134903170 は 'int' 型で表現できません)
- 問題の場所 (`undefined-behavior src/Fibonacci_v2.c:17:23` - 未定義の動作 src/Fibonacci_v2.c:17:23)

AddressSantizer とは異なり、プログラムは未定義の動作を検出しても中止されないことに注意してください。

3. MemorySanitizer による初期化されていないメモリーの使用に関するバグの検出

MemorySanitizer を使用して、初期化されていないメモリーの使用によって引き起こされるバグを検出できます。`-fsanitize=memory` コンパイラー・オプションでサニタイザーを有効にできます。

MemorySanitizer の使用に関する重要な注意事項:

- サニタイザーは、初期化されていないメモリーの読み取りを検出してもすぐには失敗しません。分岐、システムコール、または動的呼び出しが初期化されていないメモリーに直接または間接的に依存する場合にのみ失敗します。
- すべてのプロジェクトの依存関係は、`MemorySanitizer` を使用して再コンパイルする必要があります。そうしないと、大量の誤検出につながる可能性があります。

MemorySanitizer がどのようにプログラムのコーディング問題を検出するか詳しく見てみましょう。

前のセクションの整数オーバーフローは、フィボナッチ数の計算量を最大 47 に制限することでも修正できます。

単純に、`set_fibonacci_list` 関数にこの制限を追加してみましょう。

```
void set_fibonacci_list(int *arr, int n) {
  if (n > 1) {
    arr[0] = 0;
  }
  if (n > 2) {
    arr[1] = 1;
  }
  if (n > 47) {
    n = 47;
  }
  for (int i = 2; i < n; i++) {
    arr[i] = arr[i-1] + arr[i-2];
  }
}
```

図 3. set_fibonacci_list 関数の項目数を n=47 に制限

更新後のプログラムは、チュートリアル・ソース・アーカイブ内の `src/fibonacci_v3.c` にあります。

問題の観察

プログラムを再コンパイルして再度実行します。

```
$ icx src/fibonacci_v3.c -O0 -g -fsanitize=undefined -fno-omit-frame-pointer -o
fibonacci_v3_with_ubsan

$ ./fibonacci_v3_with_ubsan 100
Fibonacci Sequence
==================
0
1
...
==================
Golden ratio approximation: -0.000100335
...
```

UBSan が問題を検出しなくなりました。つまり、プログラムに含まれていた符号付き整数のオーバーフローがなくなりました。しかし、リストにまだ負のフィボナッチ数があり、黄金比もまだずれています。

また、ツールの出力も非決定的に変化します。

これは、初期化されていないメモリー使用の可能性を示しています。

サニタイザーの実行

MemorySanitizer を使ってダブルチェックできます。以下のコマンドで、MemorySanitizer を有効にしてプログラムをコンパイルします。

```
$  icx  src/fibonacci_v3.c  -O0  -g  -fsanitize=memory  -fsanitize-memory-track-
origins=2 -fno-omit-frame-pointer -o fibonacci_v3_with_msan
```

`-fsanitize=memory` コンパイラー・オプションは MemorySanitizer を有効にします。初期化されていないメモリーがどの変数に関連するものか追跡するには、オプションで `-fsanitize-memory-tracks-origins=2` を指定します。

メモリー・サニタイザーが有効なプログラムを実行すると、次の結果が得られます。

```
$ ./fibonacci_v3_with_msan 100
Fibonacci Sequence
==================
0
1
1
...
==177412==WARNING: MemorySanitizer: use-of-uninitialized-value
    #0 0x4afd9e in print_fibonacci
/home/user/sanitizer_tutorial/src/fibonacci_v3.c:36:5
    #1 0x4b03ec in main /home/user/sanitizer_tutorial/src/Fibonacci_v3.c:53:3
    #2 0x7f1eac55a7b2 in __libc_start_main (/lib64/libc.so.6+0x237b2) (BuildId:
ade58d86662aceee2210a9ef12018705e978965d)
    #3 0x41f2dd in _start
(/home/user/sanitizer_tutorial/fibonacci_v3_with_msan+0x41f2dd)

  An uninitialized value was created by an allocation of &apos;vla&apos; in the
stack frame of function &apos;print_fibonacci&apos;
    #0 0x4af6c0 in print_fibonacci
/home/user/sanitizer_tutorial/src/Fibonacci_v3.c:29:3

SUMMARY: MemorySanitizer: use-of-uninitialized-value
/home/user/sanitizer_tutorial/src/Fibonacci_v3.c:36:5 in print_fibonacci
Exiting
```

原因の特定

MemorySanitizer が `use-of-uninitialized-value` を報告しています。これは、フィボナッチ配列の最初の 47 項目を格納する際に、割り当てられていない (つまり、初期化されていない) 配列値を出力して使用していることが原因です。

この問題を解決するには、メイン関数に以下のチェックを追加する必要があります。

```
if (atoi(argv[1]) > 47) {
    printf("Please provide a number of 47 or less.\n");
    return 1;
  }
```

図 4. フィボナッチ数列の入力パラメーターが n=47 を超えた場合のチェックを追加

更新後のプログラムは、チュートリアル・ソース・アーカイブ内の `src/fibonacci_v4.c` にあります。

これで、プログラムのすべてのバグ修正が完了しました。

まとめ

このチュートリアルでは、インテル® oneAPI DPC++/C++ コンパイラーでサニタイザーを使用する基礎を紹介しました。サニタイザーは、単純なプログラム内の複数のバグを検出するのに役立ちます。

サニタイザーを使用すると、開発プロセスの早い段階で効果的に問題を発見し、時間を節約し、製品版のコードでコストのかかるエラーが発生する可能性を軽減できます。

関連資料

以下のリソースでは、インテル® oneAPI DPC++/C++ コンパイラーについて詳しい情報を提供しています。

- インテル® oneAPI DPC++/C++ コンパイラー製品ページ
- インテル® oneAPI DPC++/C++ コンパイラー導入ガイド (英語)
- インテル® oneAPI DPC++/C++ コンパイラー・デベロッパー・ガイドおよびリファレンス (英語)
- Clang コンパイラー・ユーザーズ・マニュアル (英語)
- 次世代のコンパイラー・テクノロジー: アーキテクトによる Q&A (英語)

ソフトウェアの入手

インテル® oneAPI DPC++/C++ コンパイラーは、インテル® oneAPI ベース・ツールキットまたはインテル® HPC ツールキットの一部としてインストールできます。また、コンパイラーのスタンドアロン・バージョン (英語) をダウンロードしたり、インテル® デベロッパー・クラウド (英語) プラットフォーム上でインテルの各種 CPU および GPU でテストすることも可能です。

製品および性能に関する情報

[1] 性能は、使用状況、構成、その他の要因によって異なります。詳細については、http://www.intel.com/PerformanceIndex/ (英語) を参照してください。