

SOLIDのアドレスサニタイザを  
使ってみよう！

2018.06.07

京都マイクロコンピュータ

# アドレスサニタイザの機能

---

何ができるか

# アドレスサニタイザの機能

- 以下の実行時バグを動的に検出する機能です
  - 変数のサイズを超えたアクセス (コード例1)
  - 配列オーバーラン(コード例2)
  - malloc() した領域に対して... (コード例3)
    - 領域の境界を越えたアクセス
    - free()後に、その領域にアクセス

※動的検出機能であって、ソースの静的解析とは異なる機能で、検出されるバグも異なります

# 具体的に検出されるバグのコード例1

```
char var1;

void func()
{
    char var2;
    short * p_var;
    short var3;

    p_var = &var1;
    *p_var = 0;      <- 検出

    p_var = &var2;
    var3 = *p_var;   <- 検出
}
```

(例) char 型 8bit の変数のアドレスを short型16bitのポインタを介してアクセス

- 外部変数、内部変数問わず検出されます。
- Read/Write どちらでも検出されます。

# 具体的に検出されるバグのコード例2

```
func()
{
    int i;
    char buf[10];

    for (i=0; i <= 10; i++) {
        buf[i] = i;    <- i=10の時に検出
    }
}
```

(例) メンバーが10個の配列に対して、11番目のメンバーにアクセスしようとする

i=0～9の間は正常に実行。i=10の時、初めてエラー検出。

- 外部変数、内部変数問わず検出されます。
- Read/Write どちらでも検出されます。

# 具体的に検出されるバグのコード例3

```
static const int BUFFER_SIZE = 1024;

func()
{
    char *buffer;
    char var;

    buffer=(char *)malloc(BUFFER_SIZE);

    memset(buffer,0,BUFFER_SIZE*2); <-検出

    free(buffer);
    var = *buffer; <-検出
}
```

(例1) malloc() のサイズを越えてアクセス。

(例2) free() 後にアクセス

- Read/Write どちらでも検出されます。

# アドレスサニタイザを 使うための準備

使うために前もって決めたり設定したりする内容

# アドレスサニタイザを使うための準備

## ・システム設計チームのやるべきこと

1. 適用するメモリアドレスの範囲を決めて、その範囲のサイズに対応した「アドレスサニタイザのワークメモリ(シャドウメモリ)」を確保する。  
※メモリマップエディタ

## ・各エンジニアのやるべきこと (P.15までスキップ)

1. アドレスサニタイザ有効モードでビルド  
※ソリューション構成 “Debug\_tasan” を選択
2. 除外するプロジェクトがある場合には、除外の設定  
※プロジェクトのプロパティ
3. 除外するソースファイルがある場合には、除外の設定  
※ソースファイルのプロパティ
4. 除外する関数がある場合には、除外の設定  
※関数に NOSANITIZE をつける

# 検出対象領域とシャドウメモリの設定

The screenshot shows the KMC(SOLID\_V7A\_ARM) IDE interface. The main window displays the **Memory Configurations Table** for the **memory\_map.smm** file. The table lists memory regions with their attributes, physical addresses, sizes, virtual addresses, and descriptions. Below the table, the **Address Sanitizer** section shows the **PA/VA mapping** between physical and virtual addresses.

Name	Attribute	Physical Address	Size	Virtual Address	Description
<input checked="" type="checkbox"/> VRAM	SOLID_IO	0x80600000	52.0 MByte	0x80600000	VRAMMgrの管理対象
<input checked="" type="checkbox"/> SOLID	SOLID_CORE	0x80000000	4.0 MByte	0xf0c00000	SOLID CORE
<input checked="" type="checkbox"/> H264	SOLID_IO	0x83f00000	64.0 KByte	0x83f00000	H264 work
<input checked="" type="checkbox"/> OSSTACK	SOLID_RESERVE	NO_ADDRESS	512.0 KByte	0xf0b00000	

The **Address Sanitizer** section shows the **PA/VA mapping** between physical and virtual addresses. The table is divided into two columns: **Physical Address** and **Virtual Address**.

Physical Address	Virtual Address
0xfd1f ffff	0xfd1f ffff
0xfd10 0000	0xfd10 0000
0x83ff ffff	0xfceff ffff
0x83ff f000	0xf100 0000
0x83ff efff	0xf0ff ffff
0x83f1 0000	0xf0c0 0000
0x83f0 ffff	0xf0b7 ffff
0x83f0 0000	0xf0b0 0000
0x83ef ffff	0xf0a7 ffff
0x83b0 0000	0xf0a0 0000
0x83ab ffff	0x83ff ffff
0x83a0 0000	0x839f ffff
0x839f ffff	0x8390 0000

The **Solution Explorer** on the right shows the project structure. The file **memory\_map.smm** is highlighted with a red circle. A blue speech bubble points to this file with the text: "ソリューションエクスプローラーからmemory\_map.smmを選んでダブルクリックし、メモリマップエディタを起動".

# 検出対象領域とシャドウメモリの設定

(V) をクリックし、アドレスサニタイザ用の設定画面を開く

The screenshot displays the SolidMemoryMapDesigner Explorer interface. The main window shows a table of memory regions with columns for Name, Virtual Address, and Description. A blue callout points to the 'OSSTACK' entry, indicating that clicking the '(V)' icon opens the address sanitizer settings. Below the table, the 'Address Sanitizer' section is highlighted with a red circle. The right pane shows the 'Solution Explorer' with a list of files, including 'memory\_map.smm' and 'solid\_axell\_FileSys.c'. The bottom pane shows the 'Properties' window for 'KMC.SolidMemoryMap.MemoryMapView.ViewModels.SizeUnitType'.

Name	Virtual Address	Description
VRAM	0x80600000	VRAMMgrの管理対象
SOLID	0x80000000	SOLID CORE
H264	0x83f00000	H264 work
OSSTACK	NO_ADDRESS	

Physical Address	Virtual Address
0xfd1f ffff	0xfd1f ffff
0xfd10 0000	0xfd10 0000
0x83ff ffff	0x83ff ffff
0x83ff f000	0x83ff f000
0x83ff efff	0x83ff efff
0x83f1 0000	0x83f1 0000
0x83f0 ffff	0x83f0 ffff
0x83f0 0000	0x83f0 0000
0x83ef ffff	0x83ef ffff
0x83b0 0000	0x83b0 0000
0x83ab ffff	0x83ab ffff
0x83a0 0000	0x83a0 0000
0x839f ffff	0x839f ffff
0x8000 0000	0x8000 0000

# 検出対象領域とシャドウメモリの設定

The screenshot shows the 'Address Sanitizer' configuration window. The top section lists memory regions to be monitored: SOLID, H264, and OSSTACK. Below this, the 'Address Sanitizer' section contains the following fields:

- Monitoring address: 0xf0a00000
- Monitoring size: 0x00600000
- Shadow memory address(Physical): 0x83a00000
- Shadow memory address(Virtual): 0x40000000
- Shadow memory size: 0xc0000

Callouts point to these fields with the following descriptions:

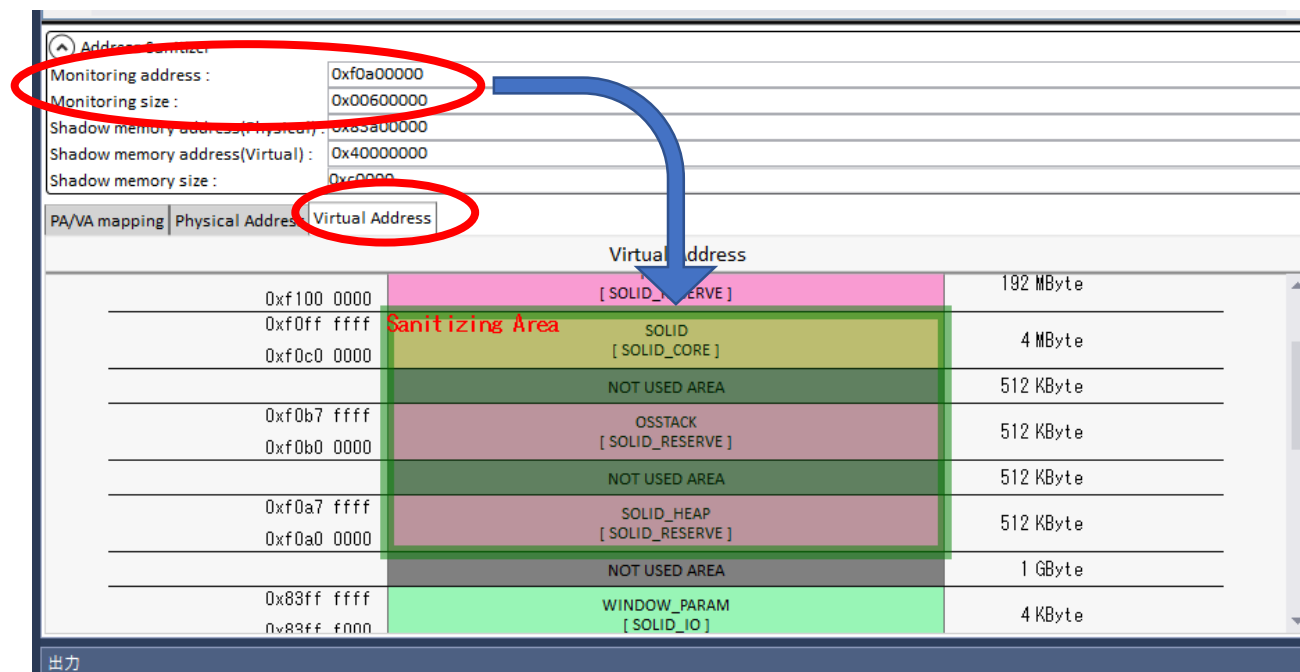
- 検出対象領域の開始番地※1 (Start address of the detection target area ※1) points to Monitoring address.
- 検出対象領域の領域サイズ※2 (Detection target area size ※2) points to Monitoring size.
- シャドウメモリとして使うメモリの物理アドレス (Physical address of memory used as shadow memory) points to Shadow memory address(Physical).
- シャドウメモリとして使うメモリの論理アドレス (Logical address of memory used as shadow memory) points to Shadow memory address(Virtual).
- シャドウメモリとして使うメモリのサイズ (※1, ※2から自動計算され入力不可) (Size of memory used as shadow memory (cannot be input as it is automatically calculated from ※1, ※2)) points to Shadow memory size.

Below the configuration fields is a table showing the mapping between Physical Address and Virtual Address:

PA/VA mapping	Physical Address	Virtual Address
	NOT USED AREA	NOT USED AREA
0xfd1f ffff	DMA_DSC [ SOLID_IO ]	0xfd1f ffff
0xfd10 0000		0xfd10 0000
	NOT USED AREA	NOT USED AREA
0x83ff ffff	WINDOW_PARAM [ SOLID_IO ]	0xfcff ffff
0x83ff f000		0xf100 0000
		IOAREA [ SOLID_RESERVE ]

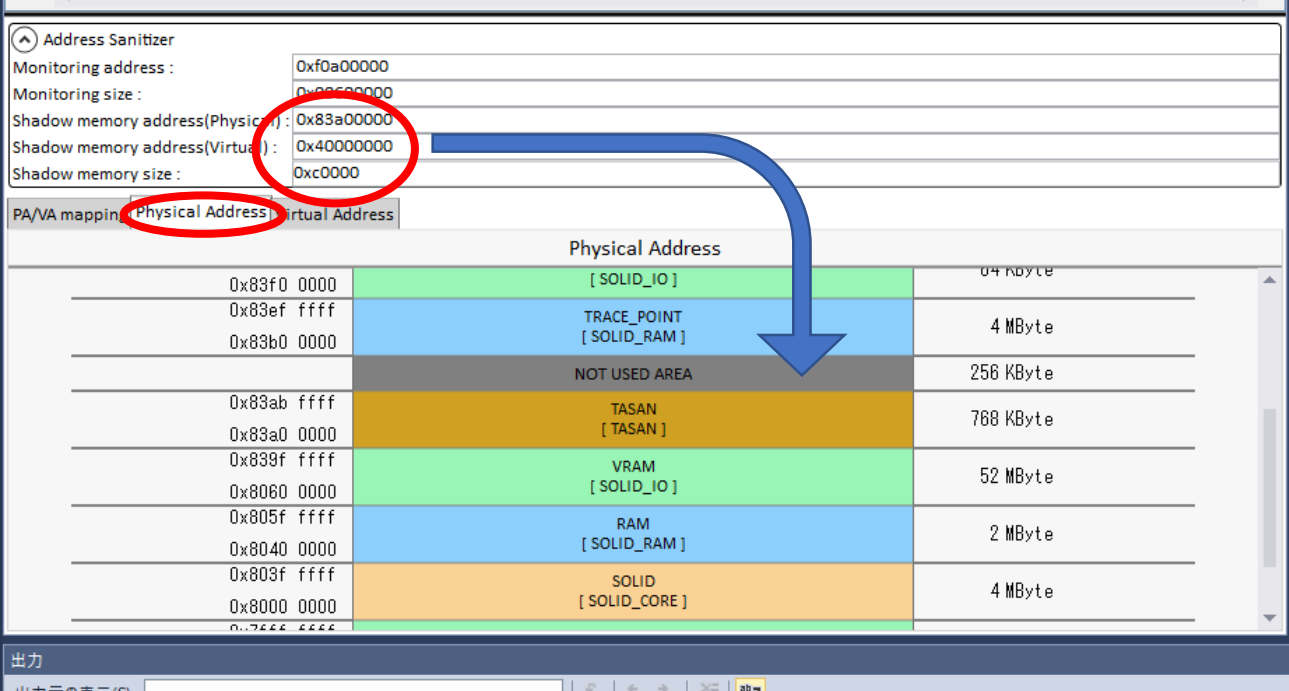
## 検出領域の設定結果

- メモリマップエディタの“Virtual Address”タブを選択すると、アドレスサニタイザーの検出対象領域をグラフィカルに確認できます。



## 検出領域の設定結果

- メモリマップエディタの“Physical Address”タブを選択すると、シャドウメモリに使われる領域が確認できます。
- 領域名“TASAN” 領域属性 “[TASAN]”



Address Sanitizer

Monitoring address : 0xf0a00000  
Monitoring size : 0x00000000  
Shadow memory address(Physical) : 0x83a00000  
Shadow memory address(Virtual) : 0x40000000  
Shadow memory size : 0xc0000

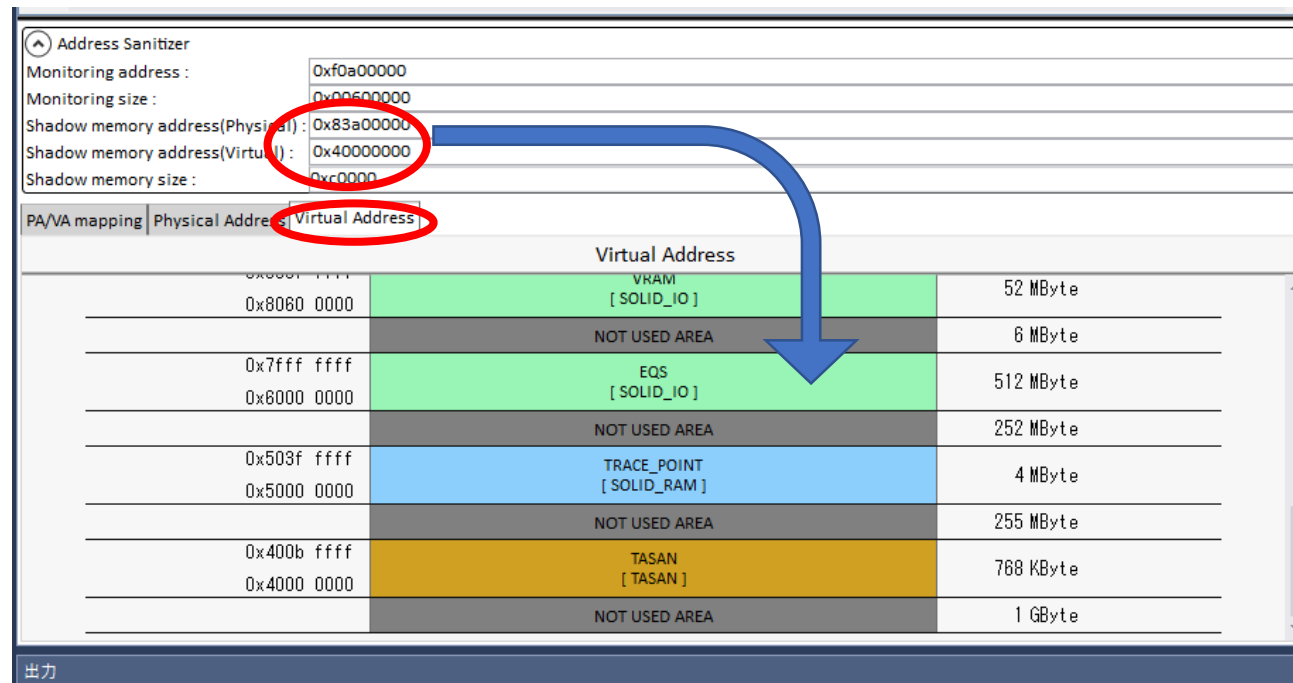
PA/VA mapping (Physical Address) Virtual Address

Physical Address	Physical Address	Size
0x83f0 0000	[ SOLID_IO ]	04 KByte
0x83ef ffff	TRACE_POINT [ SOLID_RAM ]	4 MByte
0x83b0 0000	NOT USED AREA	256 KByte
0x83ab ffff	TASAN [ TASAN ]	768 KByte
0x83a0 0000	VRAM [ SOLID_IO ]	52 MByte
0x8060 0000	RAM [ SOLID_RAM ]	2 MByte
0x805f ffff	SOLID [ SOLID_CORE ]	4 MByte

出力

## 検出領域の設定結果

- メモリマップエディタの“Virtual Address”タブを選択すると、シャドウメモリに使われる領域が確認できます。
- 領域名“TASAN” 領域属性 “[TASAN]”



Address Sanitizer

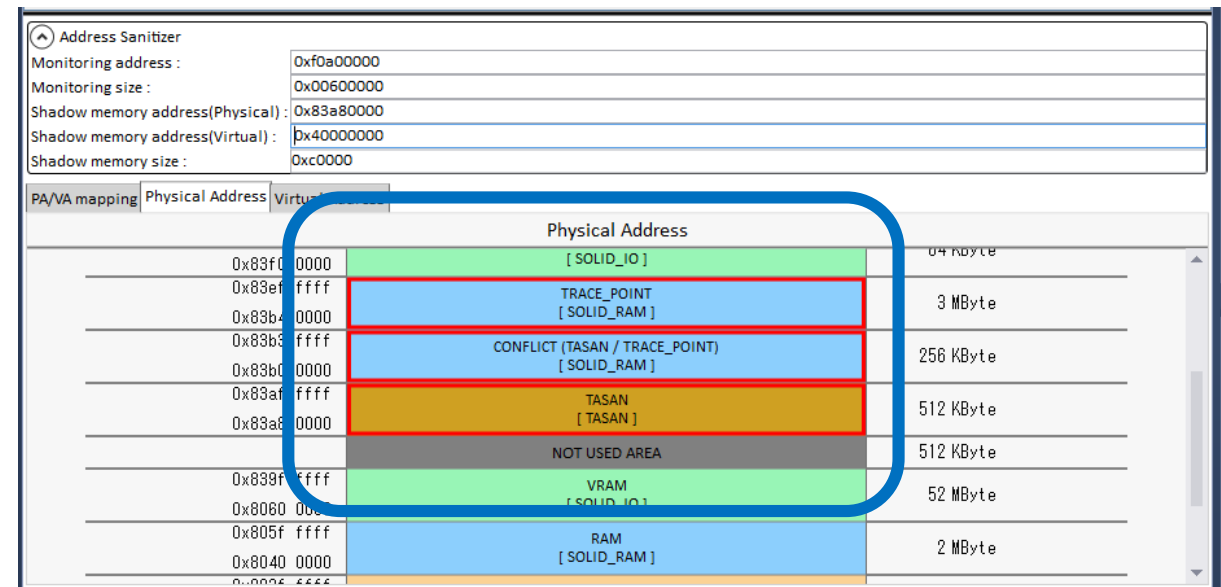
Monitoring address : 0xf0a00000  
Monitoring size : 0x00500000  
Shadow memory address(Physical) : 0x83a00000  
Shadow memory address(Virtual) : 0x40000000  
Shadow memory size : 0xc0000

PA/VA mapping	Physical Address	Virtual Address	
	0x0000 0000	VRAM [ SOLID_IO ]	52 MByte
	0x8060 0000	NOT USED AREA	6 MByte
	0x7fff ffff 0x6000 0000	EQS [ SOLID_IO ]	512 MByte
		NOT USED AREA	252 MByte
	0x503f ffff 0x5000 0000	TRACE_POINT [ SOLID_RAM ]	4 MByte
		NOT USED AREA	255 MByte
	0x400b ffff 0x4000 0000	TASAN [ TASAN ]	768 KByte
		NOT USED AREA	1 GByte

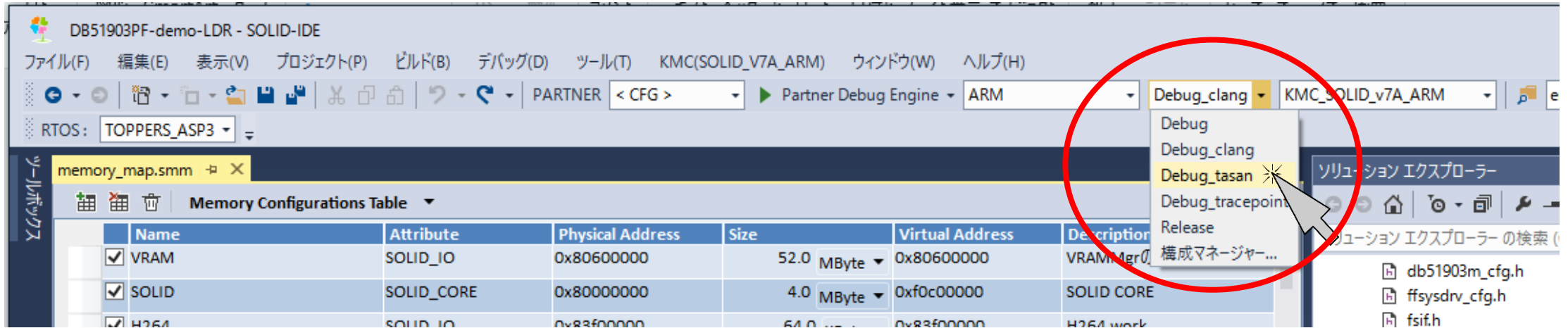
出力

# シャドウメモリ設定の注意点

- シャドウメモリが他の領域と重ならないようにしてください。右図は重なった場合の表示。
- シャドウメモリの論理アドレスは、極力低いアドレスにしてください (0x400000000推奨)。物理アドレスの配置には制限はありません。

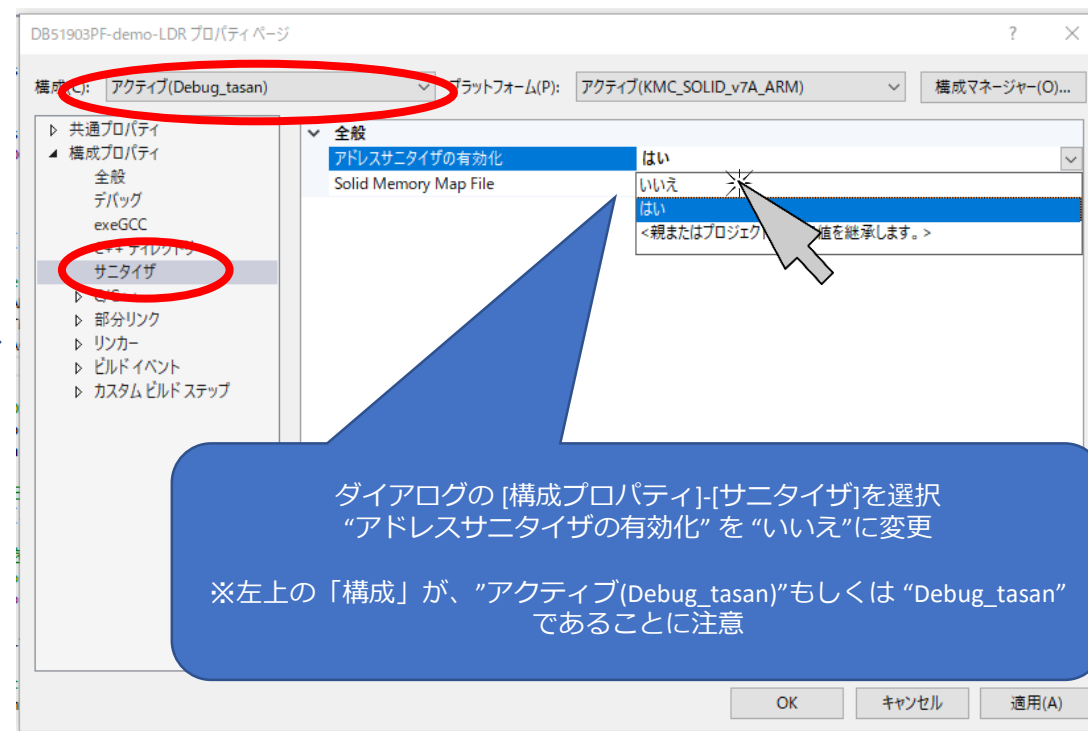
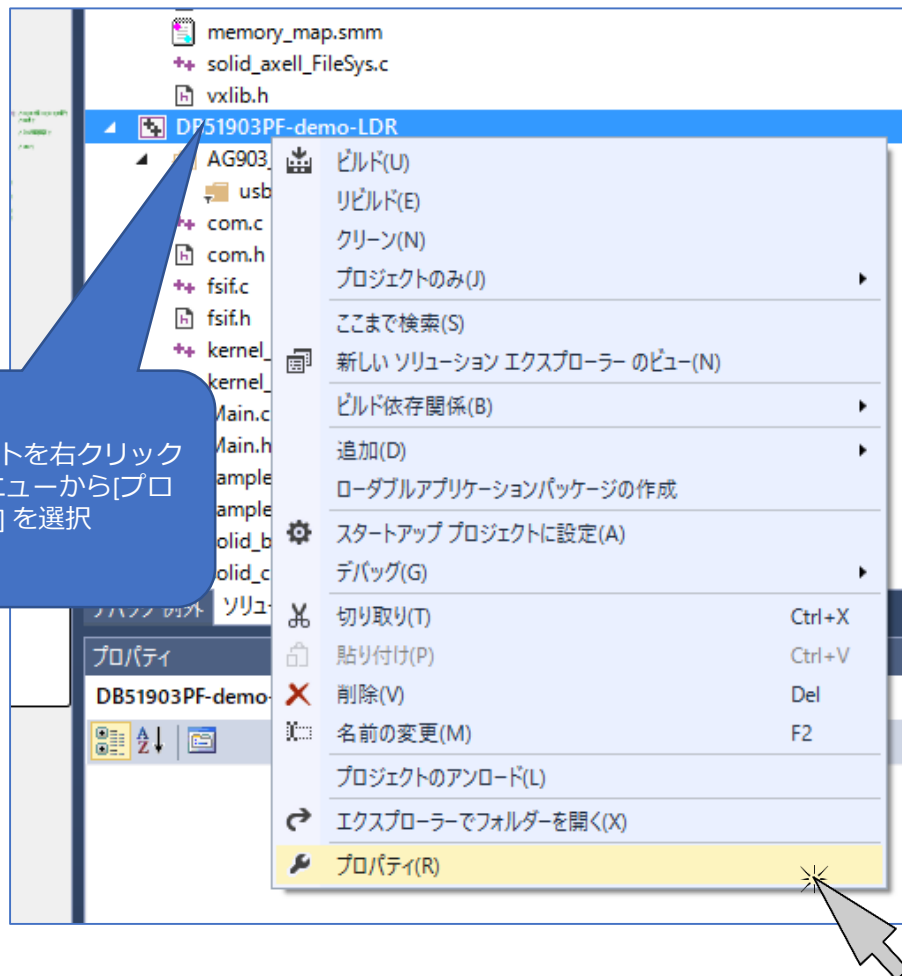


# アドレスサニタイザ有効モードでビルド



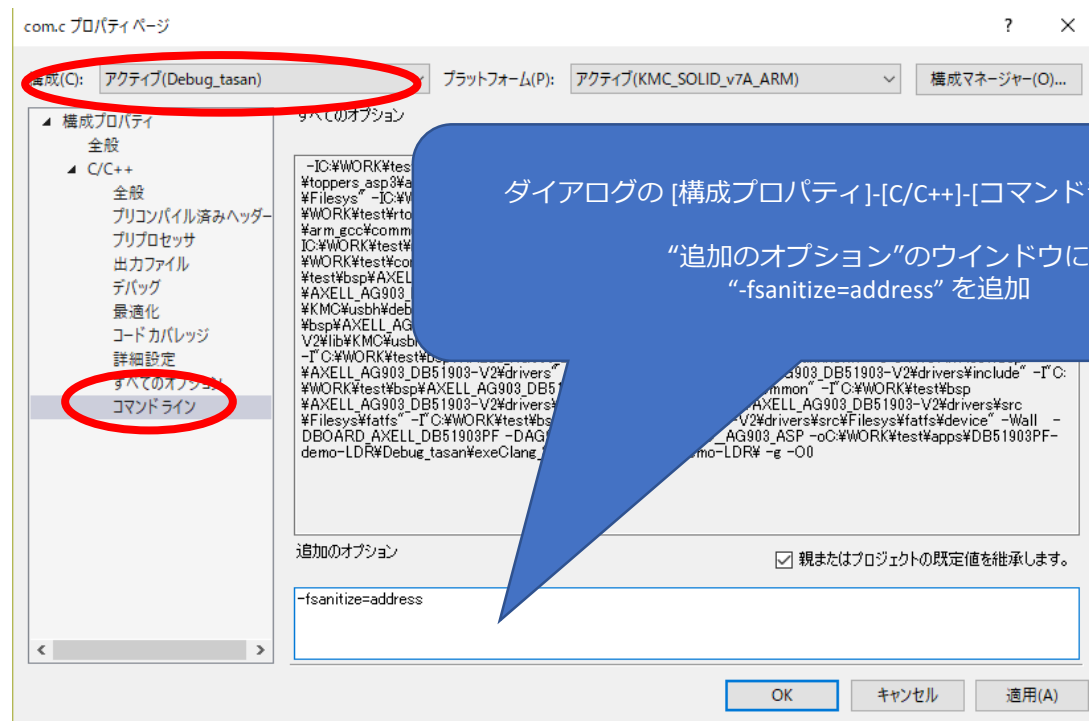
IDEの「ソリューション構成」選択リストボックスから  
“Debug\_tasan” を選択。

# プロジェクト単位での対象除外方法



# ソースコード単位での対象除外方法

- 選択的に除外する方法はありません。
- プロジェクト単位で対象除外をしたうえで、対象にするソースコードについて以下の設定をします



# 関数単位の対象除外方法

- 関数単位の対象除外方法
  1. `#include "solid_asan.h"` を追加
  2. 関数の定義の前に `"NO_SANITIZE"` を追加

```
#include "solid_asan.h"
. . . . .
NO_SANITIZE
void function1()
{
. . . . .
```

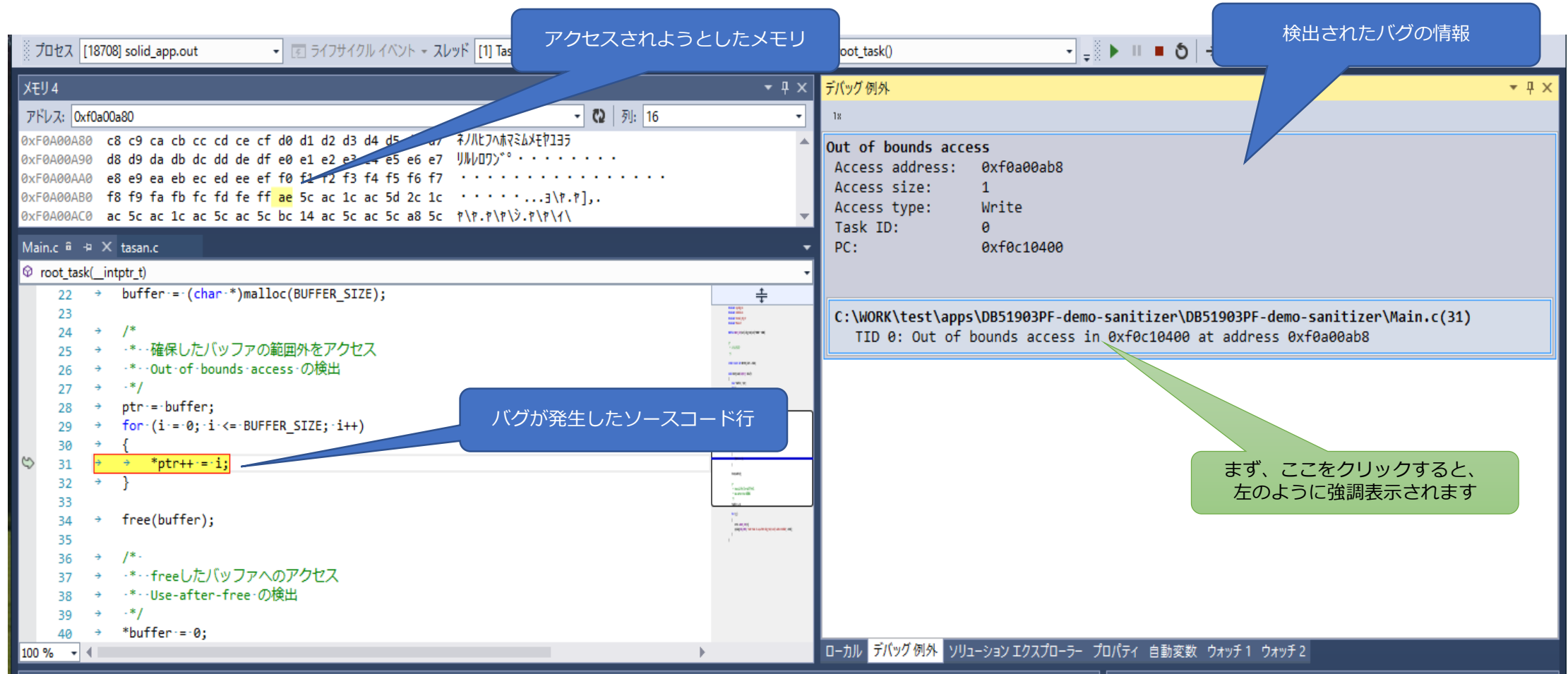
# アドレスサニタイザの使い方

実行する際の手順

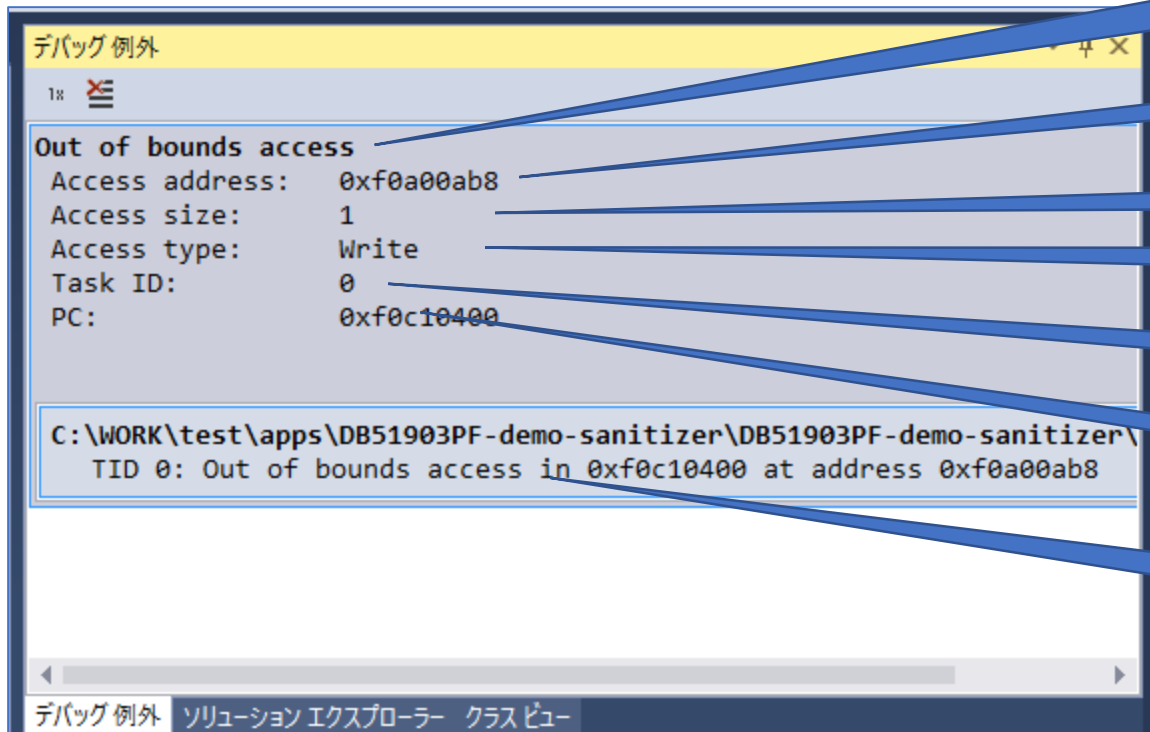
# アドレスサニタイザの使い方

- 前述の「アドレスサニタイザ有効モード」でビルドされ、ビルドが正常終了していることを確認。  
※ビルドがうまく行かない場合は、P.24 以降を確認
- 通常のデバッグと同じように「デバッグの開始」を行います
- 何もなければ、いつも通り動きます。

# アドレスサニタイザがバグを検出した時の画面



# アドレスサニタイザがバグを検出した時の画面



検出されたバグの内容

- Out of bounds access (変数、配列等の領域がアクセス)
- Access after free (malloc 領域を free後にアクセス)

不正にアクセスしようとしたアドレス

アクセスのサイズ

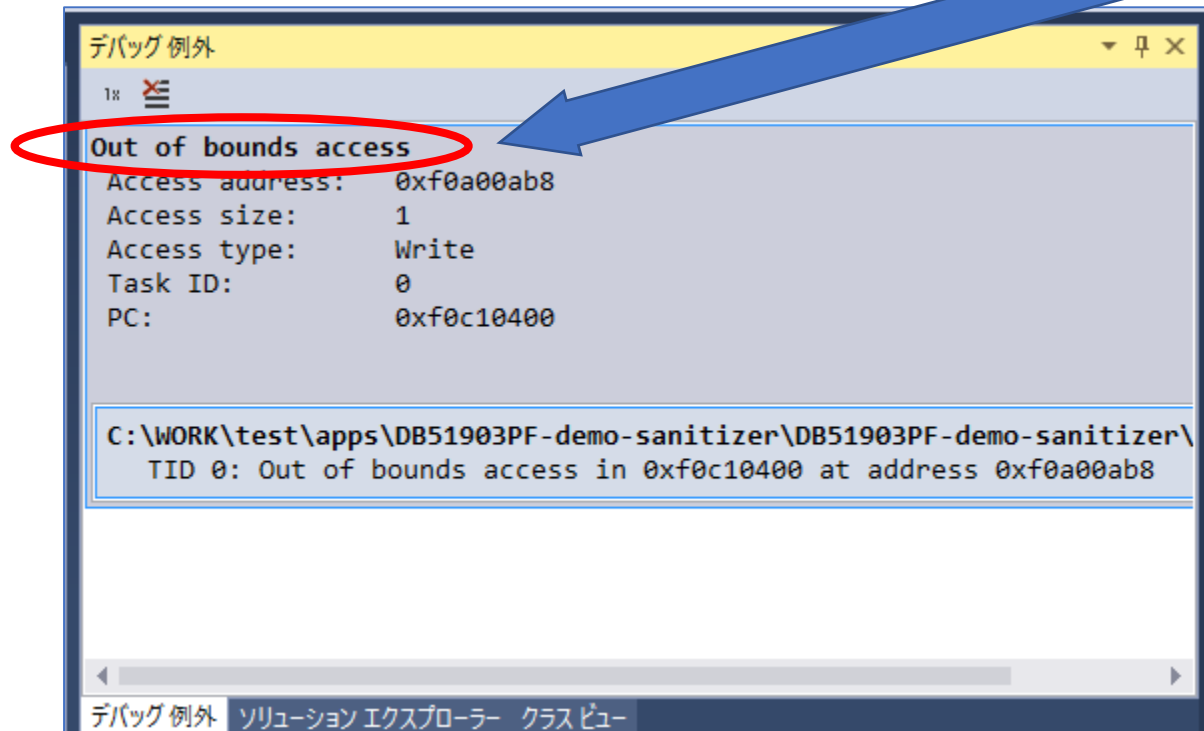
アクセスの種別 (Read/Write)

検出された時点の実行タスク ID

検出された時点のプログラムカウンタの値

検出された時点のプログラムカウンタがある場所のソースコード  
および、上記のサマリ

# 他の例外が発生した場合



- 例外の種類が下記以外の場合は、アドレスサニタイザによる検出ではありません
  - Out of bounds access
  - Access after free
- 例えば
  - Data abort
  - Prefetch abort
  - Translation fault

# 問題が起きたときに

アドレスサニタイザを有効にしたら動かなくなった

# アドレスサニタイザを有効にしたら動かなくなった

- 原因が判らないが動かない  
メモリマップの影響かもしれません。構成を”Debug\_clang” にしてビルドしなおし、正しく動作するか確認してください。
- リンク時に “SOLID AREA Full” エラーが出てビルドが失敗する  
アドレスサニタイザの対象のファイル、関数を減らしてみてください。
- スタックオーバーフローが起きた (Data abort caused by stack overflow)  
アドレスサニタイザを有効にすると、通常よりタスクのスタック消費量が増えます。スタックオーバーフローが起きた場合には、スタックサイズを通常2倍程度に増やしてください。

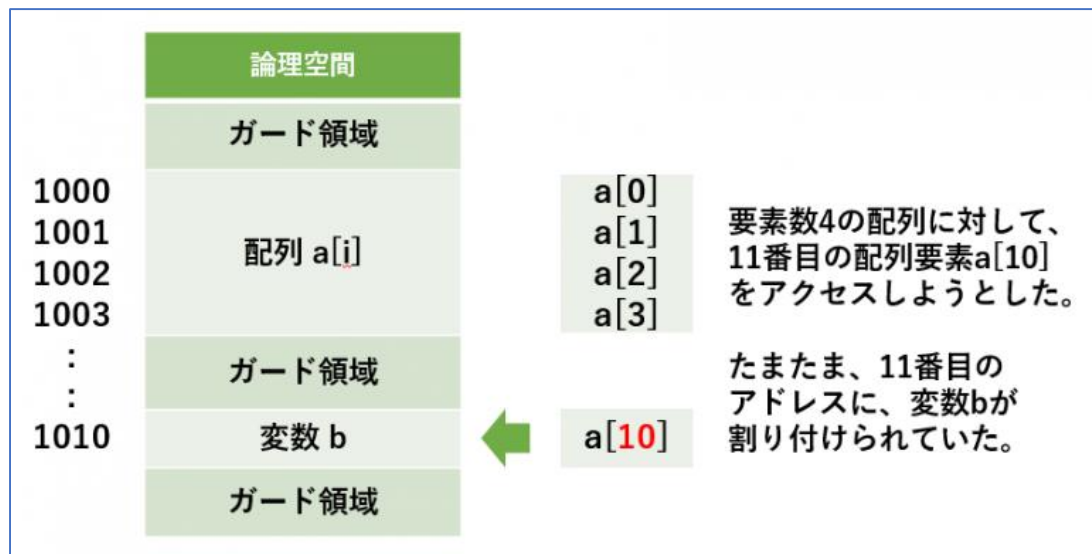
# アドレスサニタイザを有効にしたら動かなくなった

- リアルタイム性が必要な処理が間に合わなくなった
  - その部分の処理を行っているファイル、関数をアドレスサニタイザの対象から外してみてください。
  - データの複写や初期値設定を行っている処理があれば、SOLIDの標準関数(memcpy, memset 等)に置き換えてみてください。アドレスサニタイザの実行時間オーバーヘッドが減る可能性があります。

# アドレスサニタイザで 検出できないケース

アドレスサニタイザは万能ではありません

# 検出できないケース



- 配列オーバーランしても、偶然、そのアドレスに他の変数や配列があった場合には検出できません。

# 検出できないケース

```
task1()
{
    char *buffer;
    char var;

    buffer=(char *)malloc(BUFFER_SIZE);
    . . .
    free(buffer);
    . . . <- ここに task2 が割り込む
    var = *buffer;
    <- task_2がmallocした領域に含まれていた
}
```

```
task2()
{
    char *buffer2;
    buffer2=(char *)malloc(BUFFER_SIZE);
    . . .
}
```

- free()した領域が他のタスク等の malloc() によって割り当てられてしまった場合。
- そのタスクとしては free()した領域でも、不正アクセスとは判定されません。

# アドレスサニタイザの オーバーヘッド

メモリサイズ、実行スピードのオーバーヘッドについての注意

# オーバーヘッドに注意

- アドレスサニタイザは実現するための原理上、メモリサイズや実行スピードのオーバーヘッドがあります。
- アドレスサニタイザの仕組みについては以下の URL 参照  
<https://solid.kmckk.com/SOLID/archives/2859>

# オーバーヘッドの概算

- メモリサイズ
  - 前出の「シャドウメモリ」が必要です。
  - 前出のスタックサイズについても有効化時には増やす必要がある場合があります。
  - 変数のガード領域をつくるので、その分スタックや.data, .bss のサイズが増えてコードサイズが大きくなります。
  - チェックコードの呼び出し分、コードサイズが大きくなります。

シャドウメモリ分 : 検出対象エリアサイズ/8

スタックサイズ : 100% 増程度

その他を含め、ざっくり 1~3割程度多くのメモリが必要になります。

# オーバーヘッドの概算

- 実行スピードオーバーヘッド
  - バグ発生の可能性があるようなメモリアクセス毎に、チェックコードを呼び出す分、実行スピードのオーバーヘッドがあります。

コードの書き方次第なところもありますが、実行速度は、平均的には2倍程度に低下します。

以上です

