Visual C++コード分析を支えるSAL

H.28/11/05

Egtra

Boost.勉強会 #21

自分

- Egtra
  - Twitter: @egtra
  - http://dev.activebasic.com/egtra/
- 仕事: 主にVisual C++ 2005/2015 (Windows)
  - 最近はそれ以外も

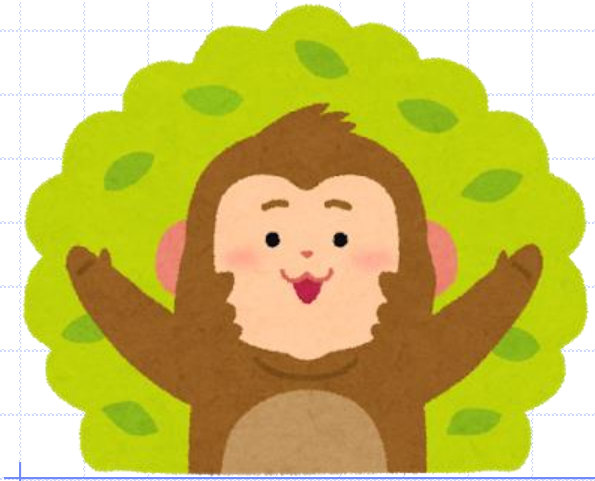
cl.exe /analyze

- Visual C++には
  静的コード解析の機能がある。
  - Visual C++ 2005から
- コンパイラオプション/analyze

SAL

- ソースコードに注釈を付ける専用言語。
- /analyzeに対する情報提供。
- 例: void f(_In_ int x);

https://msdn.microsoft.com/ja-jp/library/hh916383.aspx



使用箇所

- 以下で使われている。
  - VC++の標準Cライブラリのヘッダー
  - Windows SDKのヘッダー
- 自分のコードにも、SAL注釈を付ければ、さらに効果が得られる。
  - でもちょっと大変かも。

例

```
int main() {
        int x;
        x += 100;
}
```

例

```
int main() {
        int x;
        x += 100;
}
```

warning C6001: 初期化されていないメモリ 'x' を使用しています。

目次

- 読むか書くか
- nullptrの可否
- 要素数

仮引数の入出力の方向

読むか書くか

関数をまたいで検知したい

- このコードは警告出てほしい。

```
void f1(int* p) {
    *p += 100; // 読んで書く
}
int main() {
    int x;
    f1(&x); // 未初期化を渡している！
}
```

関数をまたいで検知したい

- 一方、このコードは警告出なくていい。

```
void f2(int* p) {
    *p = 100; // 代入！
}
int main() {
    int x;
    f1(&x); // 未初期化の変数だけどOK
}
```

アノテーション（注釈）

- 仮引数に追加情報。

```
void f1(_Inout_ int* p) {
        *p += 100;
}
int main() {
        int x;
        f1(&x);
}
```

← warning C6001: 初期化されていないメモリ 'x' を使用しています。

アノテーション（注釈）

- 元の値を読まないことの指定。

```
void f2(_Out_ int* p) {
        *p = 100; // 代入！
}
int main() {
        int x;
        f1(&x);
}
```

データの入出力方向

- _Inout_と_Out_があるなら、
  もちろん_In_もある。

```
void f3(_In_ const int* p) {
        std::cout << *p << std::endl;
}
int main() {
        int x;
        f3(&x);
} // warning C6001 初期化されていないメモリ 'x' ……
```

ここまでのまとめ

		呼び出し元で	呼び出し先（関数内）で
_In_	入力	要初期化	読み取りのみ
Inout	入出力	要初期化	読み書きする
Out	出力	初期化不要	書き込み(※)

- 3つともポインタ・参照に適用可
- _In_のみ、値渡しに適用可能
  - 例: void f(_In_ int x);

_Out_の意味

※ _Out_は書き込み限定ではない
元の値を読み取らないだけ

```
void my_rand(_Out_ int* p) {
        do {
                *p = rand();
        } while (*p >= 10000); // OK
}
```

ダメな例

前ページのコード、最初こう書いた。

```
void my_rand(_Out_ int* p) {
        while (*p >= 10000) { // C6001
                *p = rand();
        }
}
```

nullptrの可否

nullptrを渡して良いかの指定

- nullptrを渡しても良いならopt

nullptr不許可	nullptr許可
_In_	_In_opt_
Inout	_Inout_opt_
Out	_Out_opt_

呼び出す側の例

```
void f1(_In_ void* p);
void f2(_In_opt_ void* p);
int main() {
        f1(malloc(1)); // 警告
        f2(malloc(1)); // OK
}
```

warning C6387: '_Param_(1)' は '0' である可能性があります: この動作は、関数 'f1' の指定に従っていません。

注意その1

- あくまでコンパイル時警告の情報提供
- f2はこうでもいい

```
void f2(_In_opt_ void* p) {
        if (p == nullptr) {
                std::terminate();
        }
}
```

注意その2

- あくまでコンパイル時警告の情報提供
- _In_/_Inout_/_Out_仮引数でnullptrチェックしてもいい

```
void f1(_In_ void* p) {
        if (p == nullptr) {
                ……
```

呼び出される側の例 (1)

- _In_/_Inout_/_Out_ならnullptr
  チェックせず読み書きして良い

```
void print1(_In_ int* p) {
        std::cout << *p << std::endl;
}
```

（警告出ない）

呼び出される側の例 (2)

- _In_opt_/_Inout_opt_/_Out_opt_は nullptrかもしれないと想定される。

```
void print2(_In_opt_ int* p) {
        std::cout << *p << std::endl;
}
```

warning C6011: NULL ポインター 'p' を逆参照しています。

配列とヌル終端文字列

要素数

配列の注釈

- 配列へのポインタの仮引数では、要素数を指定する。

_In_reads_(n)	入力
_Inout_updates_(n)	入出力
_Out_writes_(n)	出力

- もちろんoptバージョンもある
  - 例: _In_reads_opt_(n)

例 その1

```
UINT WINAPI SendInput(
        _In_ UINT cInputs,
        _In_reads_(cInputs) LPINPUT pInputs,
        _In_ int cbSize);
```

仮引数pInputsは、
要素数cInputs個の配列を指すポインタ。

例 その2

```
int WINAPI GetWindowTextW(
        _In_ HWND hwnd,
        _Out_writes_(nMaxCount)
              LPWSTR lpString,
        _In_ int nMaxCount);
```

仮引数lpStringは、
最大で要素数nMaxCount個書き込まれる。

文字列 その1

<stdio.h>より

```
int __cdecl fputs(
        _In_z_ char const* _Buffer,
        _Inout_ FILE* _Stream
        );
```

文字列 その2

```
BOOL WINAPI DeleteFileA(
        _In_ LPCSTR lpFileName);
```

- lpFileNameはヌル終端文字列扱い。
- LPCSTRなど一部の型は、
  typedefに注釈が付与されている。

終わりに

終わりに

- ここまでの例では
  こういうのを検知できる（かも）。
  - 未初期化変数からの読み取り
  - nullptrへのアクセス
  - 配列の要素数を超えたアクセス
- 注意: /analyzeの機能はまだあるよ。