

【B5】C++テクニカルセッション



EMBARCADERO
TECHNOLOGIES®

DEVELOPER CAMP

今さら聞けない(?!)C/C++ポインター再入門

株式会社 日本情報システム
筑木真志

- “ポインタ”って何よ?
- メモリ&ポインタとの付き合いかた
- 解決法:Cの場合 ~ デバッグ用malloc
- 解決法:C++の場合 ~ スマートポインタ

- 抽象的
 - 「メモリを確保」するってなに？
- プログラムが落ちる
 - ポインタが絡むとプログラムが落ちる
 - バッファ・オーバーフロー
- 宣言の意味がわかりづらい

```
char **argv;  
const char* const str;  
char *ptr[20];  
char (*ptr)[20];  
int (*func)(const void*, const void*);
```



“ポインタ”って何よ？

- ポインタは、他の変数のアドレスを持つ変数であり、Cで頻繁に使用される。

B.W.Kernighan,D.M.Ritchie著／石田晴久訳
プログラミング言語C 第2版 P113より

```
#include <stdio.h>
#include <tchar.h>
int add(int a, int b);

int _tmain(int argc, _TCHAR* argv[])
{
    int a = 100;
    int b = 200;
    int c;

    int *d = &a;    // 変数aがメモリのどこにあるか
    int *e = &b;    // 変数bがメモリのどこにあるか
    int *f = &c;    // 変数cがメモリのどこにあるか

    c = add(a, b);
    printf(_T("%d + %d = %d\n"), a, b, c);

    *d = 300;      // なぜか、変数aの値が変わる
    *e = 400;      // なぜか、変数bの値が変わる
    c = add(a, b);
    printf(_T("%d + %d = %d\n"), a, b, c);

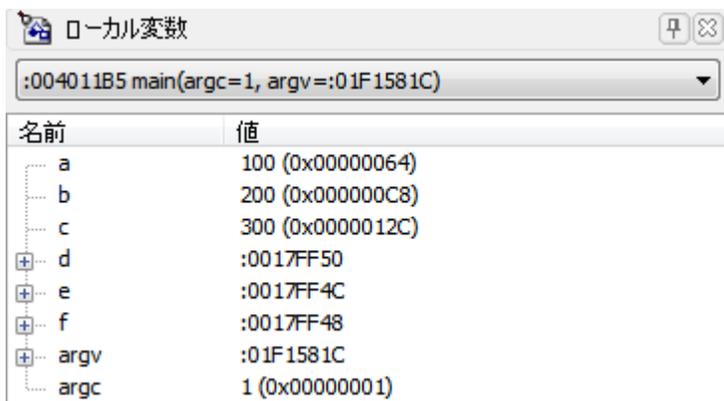
    return 0;
}

int add(int a, int b)
{
    int ret;
    ret = a + b;
    return ret;
}
```

アドレス	変数(シンボル)	中身
0012FF3C	f	0x0012FF48
0012FF40	e	0x0012FF4C
0012FF44	d	0x0012FF50
0012FF48	c	300
0012FF4C	b	200
0012FF50	a	100

変数aのアドレス
0x 0012FF50

変数dの値
0x 0012FF50



ローカル変数

:004011B5 main(argc=1, argv=:01F1581C)

名前	値
a	100 (0x00000064)
b	200 (0x000000C8)
c	300 (0x0000012C)
d	:0017FF50
e	:0017FF4C
f	:0017FF48
argv	:01F1581C
argc	1 (0x00000001)

メモリ先頭	OS予約領域	
	データ領域	スタック領域 (ローカル変数や関数の引数) ヒープ領域 (mallocやoperator newが「確保」する領域)
	テキスト領域	実際に実行されるマシン語 (main関数、各種ライブラリなど)
メモリ終端	BSS領域 (Block started by symbol)	定数
		初期化済み変数(静的／共通)
		未初期化変数(静的／共通)

- スタック領域
 - ローカル変数が格納される
 - 関数の引数が格納される
 - 関数が終了すれば自動的に解放する
- ヒープ領域
 - 関数malloc()が動的に確保する領域
 - operator newが動的に確保する領域
 - プログラマが自分で解放しなければならない

- 関数ポインタとは、メモリ(テキスト領域)に割り当てられた関数の先頭アドレス

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

```
int add(int a, int b) {  
    return a + b;  
}
```

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

```
    int a = 100;  
    int b = 200;  
    int c;
```

```
    int (*func)(int a, int b);
```

```
    // 変数funcは関数addの先頭アドレス
```

```
    func = add;
```

```
    c = (*func)(a, b); // 関数addの呼び出し
```

```
    printf("c = %d\n", c);
```

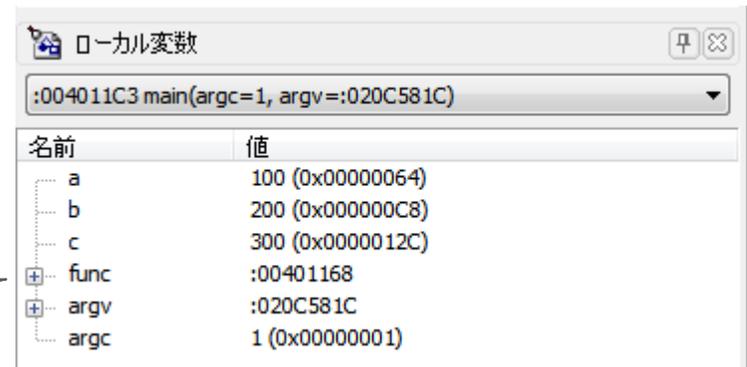
```
    return 0;
```

```
}
```

関数addの先頭アドレス
0x00401168

```
FuncPtr.c.3: int add(int a, int b) {  
00401168 55          push ebp  
00401169 8BEC       mov  ebp,esp  
FuncPtr.c.4: return a + b;  
0040116B 8B4508     mov  eax,[ebp+$08]  
0040116E 03450C     add  eax,[ebp+$0c]  
FuncPtr.c.5: }  
00401171 5D        pop  ebp  
00401172 C3        ret
```

変数funcの値
0x00401168



ローカル変数

:004011C3 main(argc=1, argv=:020C581C)

名前	値
a	100 (0x00000064)
b	200 (0x000000C8)
c	300 (0x0000012C)
func	:00401168
argv	:020C581C
argc	1 (0x00000001)



EMBARCADERO
TECHNOLOGIES®

DEVELOPER CAMP

メモリ&ポインタとの付き合いかた

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

int main(int argc, char* argv[])
{
    int i;
    char buff[9];
    char ch;

    memset(buff, 0, sizeof(buff));

    srand(time(NULL));

    for (i = 0; i < 16; ++i) {
        buff[i] = ((double)rand() / RAND_MAX) * ('z' - ' ') + '!';
    }

    printf("passwd = %s\n", buff);

    return 0;
}
```

- 無効なメモリ領域へのアクセス
 - Segmentation fault
 - Bus error
 - NULLポインタへのアクセス
- バッファオーバーフロー／バッファオーバーラン
 - 想定したメモリ領域の前後を書き換えてしまう
 - 別のメモリ領域が書き換わる
 - メモリ領域に悪意のあるコードが書き込まれる
- メモリリーク
 - 使用済みメモリ領域が確保されたまま、再利用されない
 - 別の処理でメモリが確保できなくなり、アプリケーションやOSがクラッシュする

- C/C++の変数宣言は内から外へと解釈する
- C/C++の変数宣言は「置き換え」である

例1) `int *val;`

→ *が付いている。すなわち、`val`はアドレスである。

→ *`val`は`int`型である。

変数`val`は`int`型へのポインタである。

例2) `const char *const str = "ABCDEFGH";`

→ `str`は`const`である。すなわち、`str`の「中身」は変更出来ない。

→ *が付いている。すなわち、`str`はアドレスを表している。

→ *`str`は`const char`型である。

変数`str`はReadOnlyな文字列定数でかつ、変更不可である。

- 例3) `char *ptr[20];`
 - []が付いている。すなわち、ptrは配列である。
 - *が付いている。すなわち、ptr[n] で表現するものはアドレスである。
 - *ptr[n] で表現するものはchar型である。変数ptrは、char型へのポインタの配列である。

- 例4) `char (*ptr)[20];`
 - *が付いている。すなわち、ptrの「中身」はアドレスである。
 - []が付いている。すなわち、*ptrは配列である
 - (*ptr)[n]はchar型である。変数ptrはcharの配列へのポインタである。

- typedef宣言の使用

```
typedef char[20] MYBUFFER;  
MYBUFFER* buffer;  
buffer = (MYBUFFER *)malloc(sizeof(MYBUFFER)* count);
```

- 配列で置き換えられるのであれば、置き換える

```
char **foo; → char *foo[];
```

- 複雑なポインタ宣言は使わない！
 - 可読性、保守性の低下
 - Keep it Simple, Stupid!の原則
 - ポインタへのポインタへのポインタなんて、もってのほか！
 - もっと、別のシンプルな方法があるはず
- でも、やっぱり、ポインタ／メモリ管理は重要で、どうしても逃げることは出来ない...



解決法:Cの場合 ~ デバッグ用malloc

- mallocハックとは、C標準のメモリ処理関数を「乗っ取る」
 - プリプロセッサでmalloc等を置換して、自前のmallocでメモリ領域を管理する。
 - リンク時にC標準ライブラリより前に、デバッグ用ライブラリをリンクする。
- mallocハックを使用したデバッグ用ライブラリ
 - mpatrol (<http://mpatrol.sourceforge.net/>)
 - ccmalloc(<http://cs.ecs.baylor.edu/~donahoo/tools/ccmalloc/>)
 - malloc等が呼び出された時のログを作成する
 - ただし、C++Builderでは使えない

- ガベージコレクションライブラリ Boehm GC
(http://www.hpl.hp.com/personal/Hans_Boehm/gc/)
 - ガベージコレクションを実装したmalloc
 - mallocで確保した領域が「不要」になれば自動的に解放
 - ポインタと見なせるメモリイメージより、メモリの使用状態を判別
 - mallocの代替として使用可能

```
#include <stdio.h>
#include "include/gc.h"

#pragma link "gc.lib"

typedef struct _Tree_tag {
    struct _Tree_tag* left;
    struct _Tree_tag* right;
} Tree;

Tree* generate_tree(int level) {
    if( level > 0 ){
        Tree* new_tree =
            (Tree*)GC_malloc(sizeof(Tree));

        new_tree->left = generate_tree(level-1);
        new_tree->right = generate_tree(level-1);
        return new_tree;
    } else {
        return (Tree*)0;
    }
}
```

```
int main(int argc, char** argv)
{
    int i;

    for(i = 0; i<100 ; i++){
        Tree* root ;
        printf("GC_get_heap_size: %ld\n",
            GC_get_heap_size() );
        printf("GC_get_free_bytes: %ld\n",
            GC_get_free_bytes() );
        printf("GC_get_bytes_since_gc: %ld\n",
            GC_get_bytes_since_gc() );
        printf("GC_get_total_bytes: %ld\n",
            GC_get_total_bytes() );
        root = generate_tree(20);
        printf("GC counts: %d\n", GC_gc_no );
    }

    return 0;
}
```



解決法:C++の場合 ~ スマートポインタ

- STL(Standard Template Library)の使用
 - 可変長配列(std:: vector)、リスト(std::list)、連想配列(std::map)など
 - STL内部でメモリ領域の管理を行っている
- スマートポインタの使用
 - 自動的にメモリ領域の解放を行う
 - 挙動によりいくつか種類がある
- その結果、ソースコード中でメモリ領域の管理が不要になる

```
//-----  
void __fastcall TForm1::Button1Click(TObject *Sender)  
{  
    // スマートポインタにしてみる  
    std::unique_ptr<TStringList> pList(new TStringList());  
  
    // アクセスは変わらない  
    pList->LoadFromFile("DATA.TXT");  
    for (int i = 0; i < pList->Count; ++i) {  
        // 何らかの処理  
        foo(pList->Strings[i]);  
    }  
  
    // スマートポインタなので、pListが持っていたメモリ領域は自動的に解放される  
  
}  
//-----  
void __fastcall TForm1::foo(UnicodeString us)  
{  
    throw Exception("何らかのエラー");  
}
```

- 自動的にメモリ領域の開放を行うポインタ
 - `std::unique_ptr` / `boost::scoped_ptr`
 - 参照されなくなったら、メモリ領域を解放する
 - `boost::shared_ptr` (`std::tr1::shared_ptr`)
 - 参照カウンタ付きポインタ
 - 参照カウンタが0になったらメモリ領域を解放する
 - `boost::weak_ptr` (`std::tr1::weak_ptr`)
 - `std::shared_ptr`の参照カウンタを変化させない
 - `boost::intrusive_ptr`
 - 自前で参照カウンタを管理をする

- unique_ptrは参照されなくなったら、自動的にメモリを解放する
 - 従来のauto_ptrは非推奨(deprecated)となる
 - 代入が出来ない
 - 想定する使い方
 - VCLクラスを使用する場合
 - pimplイディオムを実装する
 - ヘッダファイルにインターフェースだけ用意して、実装は別のクラスで行う
 - Singletonパターンの実装

- Singletonパターン: 1つのオブジェクトしか存在しない

```
#include <memory>
#include <vcl.h>

class COption
{
public:
    static COption& COption::getInstance();
private:
    static std::unique_ptr<COption> s_pInstance;
};

std::unique_ptr<COption> COption::s_pInstance(NULL);

COption& COption::getInstance()
{
    if (s_pInstance.get() == NULL) {
        // Double-Checkd Lockingイディオム
        std::unique_ptr<TCriticalSection> pCriticalSection(new TCriticalSection);
        pCriticalSection->Enter();

        // CriticalSectionの生成中に別スレッドで初期化されているかもしれないので、再チェック
        if (s_pInstance.get() == NULL) {
            s_pInstance.reset(new COption());
        }
        pCriticalSection->Release();
    }
    return *s_pInstance;
}
```

- shared_ptrはポインタの参照カウントを数える
 - 代入で参照カウントを増やす
 - 破棄で参照カウントを減らす
 - shared_ptr同士で循環参照した場合は正しくメモリが解放されない。
 - その場合は、weak_ptrを使用する
- weak_ptrはshared_ptrの参照カウントを変化させない
 - shared_ptrの「本体」が有効か無効かがチェックできる

- ファクトリパターン(仮想コンストラクタ)
 - 基本となる共通の手続き(インターフェース)を基底クラスに用意
 - 基底クラスを継承したクラスで、おのものの振る舞いを実装する

```
#include <vector>
#include <boost/shared_ptr.hpp>
#include <boost/foreach.hpp>
#include <tchar.h>

// 図形要素基底クラス
class CPrimitiveBase
{
protected:
    CPrimitiveBase(){}
    CPrimitiveBase(const CPrimitiveBase& Primitive);
public:
    virtual ~CPrimitiveBase() {}
    virtual void draw() const = 0;           // 描画
};
typedef boost::shared_ptr<CPrimitiveBase> CPrimitive;
typedef std::vector<CPrimitive> CPrimitiveArray;
```

```
// 図形要素:直線
class CPrimitiveLine : public CPrimitiveBase
{
protected:
    // 通常のコストラクタは隠蔽する
    CPrimitiveLine() {}
    CPrimitiveLine(const CPrimitiveLine& Primitive) {}
public:
    virtual ~CPrimitiveLine() {}
    static CPrimitive create() {
        return CPrimitive(new CPrimitiveLine());
    }
    virtual void draw() const {printf("line¥n");}
};

// 図形要素:文字列
class CPrimitiveText : public CPrimitiveBase
{
protected:
    // 通常のコストラクタは隠蔽する
    CPrimitiveText() {}
    CPrimitiveText(const CPrimitiveText& Primitive) {}
public:
    virtual ~CPrimitiveText() {}
    static CPrimitive create() {
        return CPrimitive(new CPrimitiveText());
    }
    virtual void draw() const {printf("text¥n");}
};
```

```
// 図形要素の追加
void addPrimitive(CPrimitiveArray& arr)
{
    // 直線オブジェクトの生成
    arr.push_back(CPrimitiveLine::create());
    // 文字列オブジェクトの生成
    arr.push_back(CPrimitiveText::create());
}

// 図形要素の描画
void drawPrimitive(CPrimitiveArray& arr)
{
    BOOST_FOREACH(CPrimitive& e, arr) {
        // 格納されている図形オブジェクトを描画する
        e->draw();
    }
}

int _tmain(int argc, _TCHAR* argv[])
{
    CPrimitiveArray arr;
    addPrimitive(arr);
    drawPrimitive(arr);

    return 0;
}
```

- intrusive_ptrは自前でポインタの参照回数を管理する
 - 代入で関数intrusive_ptr_add_ref()が呼ばれる
 - 破棄で関数intrusive_ptr_release()が呼ばれる
 - COMオブジェクトの管理に有用

```
class MyClass
{
public:
    MyClass();
    virtual ~MyClass();
public:
    void addref();
    void release();
};

void intrusive_ptr_add_ref(MyClass* p) { p->addref(); }
void intrusive_ptr_release(MyClass* p) { p->release(); }

int _tmain(int argc, _TCHAR* argv[])
{
    boost::intrusive_ptr<MyClass> ptr(new MyClass());
    return 0;
}
```



EMBARCADERO
TECHNOLOGIES®

DEVELOPER CAMP

まとめ

- はっきり言って、ポインタは「怖く」ない！
 - 積極的にプログラムを「クラッシュ」させてみてください。
 - デバッガでプログラムを追っかけてみてください。

- でも、「生ポインタ」の使用は控えめに...

ご静聴ありがとうございました!!