2018 年 5 月版

Kodatuno R3.2 改造レポート

舞鶴工業高等専門学校　眞柄賢一

0　はじめに

本レポートは、Kodatuno を開発された金沢大学マンマシン研究室に敬意を表するとともに、Kodatuno のさらなる発展のため、現状での問題点と改善方法をまとめました。ただし、ここに示したことが必ずしも正解というわけではありませんし、それを強要するものでもありません。あくまで一個人の考察として捉えていただければ幸いです。なお、Kodatuno ライブラリの全部を見たわけではなく、IGES ファイルの読み込みに関連する部分だけを抜粋して見ていますので、全体として見たときに修正によって不都合が生ずるかもしれません。

1　malloc の廃止

C++では、new 演算子によって言語レベルでメモリの動的確保が可能となっています。C 言語の標準ライブラリ関数である malloc() を使う必要性は全くありません [1]。逆に後述するポリモーフィズム（多態性）には new 演算子による型指定が非常に重要です。よって、オリジナルコードの malloc() 関数は全て new 演算子に置き換えました。

もう 1 つのメリットとして、Visual Studio の場合 DEBUG_NEW を使うとメモリリークが発生したとき、どこの new 演算子が解放されていないかがわかるようになります。したがって、各 cpp の先頭に

```
#if defined(_DEBUG) && defined(_MSC_VER)
#define new DEBUG_NEW
#endif
```

と記述しました。これにより IGES ファイルを読み込んだ時に起こるメモリリークは、NURBS_Func.cpp の GenNurbsC() 関数 238 行目からの

```
238     Nurbs->T = (double *)malloc(sizeof(double)*Nurbs->N);   // 実際にはnew演算子に置換済み
239     Nurbs->W = (double *)malloc(sizeof(double)*Nurbs->K);
240     Nurbs->cp = (Coord *)malloc(sizeof(Coord)*Nurbs->K);
```

が判明しています。正確には読み込んだ時ではなくプログラムを終了したときなので、おそらく NCVC 本体側で消し方のルールが間違っているものと推測されますが、ライブラリ内で確保したメモリはライブラリ側で後始末をつけるべきと考えます。現在原因と消すタイミング（NURBSC 構造体が union に属しておりデストラクタを置けない）を調査中です。

2　セキュアコード

Visual Studio に限った話ですが、strcpy() 関数などセキュリティが弱い古いバージョンの関数は推奨されないバージョンとしてマークされ、新しいバージョンには _s（サフィックス）が付いています [2]。現在は他の修正作業を優先して行っているため、stdafx.h に

```
#define _CRT_SECURE_NO_WARNINGS
```

を定義し、応急処置を行っています。とくにポインタ操作による構文解析などは、str 系関数を使わず、boost::tokenizer や boost::spirit を使ってスマートに書けないかを含め検討中です。

文字列を扱うstr系関数に関連するところで言うともう1つ、例えばBODY名を保持するchar Name[FNAMEMAX]も、下記のように書き換えた方が良いでしょう。これでstr系関数を少しでも減らせますし、なにより直感的にわかりやすいと思います。

```
    #include <string>
    ...
    // Variable: Name[FNAMEMAX]
    // BODY名
//  char Name[FNAMEMAX];
    std::string Name;       // STLで用意されている文字列型
    ...
    Name = BodyName;        // strcpy(Name, BodyName);
```

STL（Standard Template Library）は、C++で用意されている標準ライブラリの1つです。

3 クラスの設計

C++では、データ（オブジェクト）とそれを操作する処理を1つの構造体（クラス）にまとめることができます。これはカプセル化の基本概念です。さらにprivateやpublicを使い分けることで、データの不正な書き換えなどのバグを抑えることができ、信頼性の高いプログラムを書くことができます[1)]。オリジナルコードは、基本コンセプトとしてC言語寄りのC++で書いてあるとのことですが、中途半端なC言語の手続き型パラダイムは逆に混乱の元となります。できる限りC++オブジェクト指向パラダイムに沿うよう書き換えました。

3.1 メンバ変数に関係のない関数は通常の関数へ

まずクラス設計をスリム化するために、メンバ変数を直接的に扱わないサブルーチン的な関数を通常の関数扱いに変更しました。通常関数においてもstatic宣言することで外部ソースからのアクセスができなくなります。privateメンバ関数にする必要はありません。

3.2 コンストラクタによる初期化

BODY.hで宣言されているCIRAやCONAなどの構造体にコンストラクタを配置しました。データの初期化はコンストラクタで統一した方が良いでしょう。関連する事項としてBODY.cppの393行目に添え字不正のバグがあります。

```
393     CirA[i].cp[0] = CirA[i].cp[1] = CirA[i].cp[3] = SetCoord(0,0,0);    // [3]->[2]
```

ただし、unionに該当クラスが含まれるとコンストラクタ・デストラクタの効果がなくなる（とくに後者はコンパイルエラー）ので、unionの必要性を含め検討が必要です。

3.3 拡張 union 型の boost::variant（参考）

unionの欠点として、以下の項目があります。

- コンストラクタやデストラクタを必要とするオブジェクトを含むことが出来ない
- 共用体に入っているどの型が有効なのか判定する方法がない

この点を改善し、任意のオブジェクトを保持でき、かつ、現在有効な型の判別方法が提供されるunionのパワーアップ版にboost::variant[3)4)]があります。

```
union {
    int     n;
    double  d;
    char*   s;
} u;
```

例えば上記の例は、

```
boost::variant<int, double, char*> u;
```

と書くことができます。以下に例題を示します。

```
#include <iostream>
#include <string>
#include <vector>
#include "boost/variant.hpp"

using namespace std;
using namespace boost;

int main()
{
    // string, int配列, double の型が入る共用体
    variant<string, vector<int>, double>   v;

    v = 3.141592;                           // double型を代入
    assert( v.which() == 2 );               // which()メンバ関数でdouble型を確認
    cout << get<double>(v) * 2.0 << endl;   // double型として操作

    v = string("文字列");
    assert( v.which() == 0 );
    cout << "length=" << get<string>(v).length() << endl;

    v = vector<int>();
    assert( v.which() == 1 );
    get< vector<int> >(v).push_back(10);
    cout << "vector size=" << get< vector<int> >(v).size() << endl;

    return 0;
}
```

```
> ./variant1
6.28318
length=6
vector size=1
```

variant に対する操作は”which() で型を判定しその中身に応じて get() で取り出す” というパターンに落ち着きます。このパターンの切り口を変えてみると、きれいなオブジェクト指向のコードが書けます。

```
using namespace boost;  // まで上記サンプルと同じ

struct do_double : static_visitor<>
{
    template<typename T>
    void operator()(T& t) const {
        t = t + t;
    }
};

int main()
{
    variant<int, double, string>  v;

    v = -2;
    apply_visitor( do_double(), v );
    cout << v << endl;

    v = 3.14;
```

```
20      apply_visitor( do_double(), v );
21      cout << v << endl;
22
23      v = "hoge";
24      apply_visitor( do_double(), v );
25      cout << v << endl;
26
27      return 0;
28  }
```

```
> ./variant2
-4
6.28
hogehoge
```

apply_visitor() というサポート関数によってライブラリ側で自動的に振り分けされるため、抽象化された読みやすいコードになります。NCVC 本体でも使用していますが[4]、Kodatuno（改）に採用するかは現在検討中です。

3.4 COMPELEM と CURVE

この２つの共用体について、明らかに”ポインタ変数の共用体”のはずなので、union の中に実体を書くのはおかしいです。CURVE を例に以下のように書き換えました。

```
union CURVE
{
//  CIRA  CirA;
//  COMPC CompC;
//  CONA  ConA;
//  NURBSC NurbsC;
    void*   substitution;   // ここに代入
    CIRA*   CirA;
    COMPC*  CompC;
    CONA*   ConA;
    NURBSC* NurbsC;
};
// 本当はvariantで書きたい
//typedef CURVE boost::variant<CIRA*, COMPC*, CONA*, NURBSC*>

struct CONPS
{
    int crtn;
    int SType;
    int BType;
    int CType;
    NURBSS *pS;
//  CURVE *pB;
//  CURVE *pC;
    CURVE pB;   // ここは実体
    CURVE pC;
    int pref;
    int pD;

    CONPS() {
        crtn = 0;
        SType = 0;
        BType = 0;
        CType = 0;
        pS = NULL;
        pB.substitution = NULL;
        pC.substitution = NULL;
        pref = 0;
        pD = 0;
    }
};
```

代入はコンストラクタにもあるように substitution へ行えばよいでしょう。void*型なので、例えばIGES_Parser.cppの GeConpSPara() 関数 952 行目や 956 行目では、GetDEPointer() のキャストが不要になります。
実際の使用例では、下記のサンプルのとおりドット演算子とアロー演算子の位置が若干変わる程度です。しかし、型保障がないので本当は boost::variant を使いたいところです。

```
//      for(int j=1;j<body->TrmS[i].pTO->pB->CompC.N;j++){  // 旧
        for(int j=1;j<body->TrmS[i].pTO->pB.CompC->N;j++){  // 新
```

実はこの修正で、NURBSC クラスにデストラクタが配置できるようになるので、メモリリークの解決になるかと思いましたが、デストラクタを配置すると Invalid address specified to RtlValidateHeap で本体終了時にプログラムが異常停止してしまいました（泣）。消し方の問題か、根本解決するか、あとは Java のようなガーベジコレクションが使える boost::shared_ptr に切り替えるか、検討する必要があります。
参考までに、NCVC 本体での該当箇所を以下に抜粋します。

```
class CNCDoc : public CDocBase
{
    ...snip
    BODY*       m_kBody;        // Kodatuno Body
    BODYList*   m_kbList;       // Kodatuno Body List
    ...
};

//  IGESファイルの読み込み関数
void CNCDoc::ReadWorkFile(LPCTSTR strFile)
{
    CString     strPath, strName;
    IGES_PARSER iges;

    // ファイル名の検査
    ...snip

    m_kBody = new BODY;
    if ( iges.IGES_Parser_Main(m_kBody, strPath) != KOD_TRUE ) {
        delete  m_kBody;
        return;
    }
    m_kbList = new BODYList;
    iges.Optimize4OpenGL(m_kBody);
    m_kBody->RegistBody(m_kbList, strFile);
}

//  あとしまつ
CNCDoc::~CNCDoc()
{
    ...snip
    // IGES Body （こちらでnewしたものは責任もって消す）
    if ( m_kBody ) {
        m_kBody->DelBodyElem(); // これでNURBSC消えない？？
        delete  m_kBody;
    }
    if ( m_kbList ) {
        m_kbList->clear();
        delete  m_kbList;
    }
}
```

消し方が間違っていればぜひアドバイスをいただきたいところです。

3.5 メンバ関数の再配置

以下に抜粋した例題を示します。

```
struct A
{
    int foo;
};
class A_Func
{
public:
    void Func1(A* a) {
        a->foo = 1;
    }
    int Func2(A* a) {
        a->foo = 2;
        return a->foo;
    }
};
...
int func(A* a)      // 実際にFunc1()を呼び出したい場合
{
    A_Func  func;   // A_Funcインスタンスを生成し
    func.Func1(a);  // A_Funcクラス経由でFunc1()を呼び出す？？
}
```

文法的に間違っていなくても、これはやらないほうが良いでしょう。混乱を招く要因です。Func1() や Func2() が A 構造体のメンバ変数をさわるのなら、それは A 構造体のメンバ関数にすべきです。上記の例は、以下のように書き換えます。

```
class A         // struct Aと同義語ですが
{
    int foo;    // 本当はできる限りメンバ変数は隠蔽したい
public:
    void Func1(void) {
        foo = 1;
    }
    int Func2(void) {
        foo = 2;
        return foo;
    }
};
//class A_Funcは廃止
...
int func(A* a)  // 実際にFunc1()を呼び出したい場合
{
    a->Func1(); // aインスタンスに対して関数呼び出し（メッセージを送る）
}
```

オリジナルコードで言うと、NURBS_Func クラスや Describe_BODY クラスのメンバ関数が該当します。これらは NURBSC や NURBSS、BODY 等の該当クラスのメンバ関数へと移動させました。

4 動的配列（リスト）構造

KODlistData クラスは、STL の vector または list に置き換える予定ですが、影響が大きすぎてラッパーとして残すかもしれません。

```
std::list<Data*> list;
```

と宣言すれば、Data*型を保持する動的リスト構造が構築でき、面倒な追加や削除の機能は標準で用意されています。vector や list に限らず、STL で用意されている機能は積極的に活用すべきでしょう。

他にクォータニオンを表すクラスなども boost で用意されています。こちらは標準ライブラリではないですが、boost は別格なので情報収集くらいはしておくことをオススメします。

5 ポリモーフィズム

BODY.h で宣言されている各種図形情報へのポインタですが、派生と仮想関数を使えば、もっとスマートに記述できるかもしれません。以下に例題を示します。

```
1  #include <vector>
2  #include <iostream>
3
4  using namespace std;
5
6  class CZukei
7  {
8  public:
9      virtual void OnDraw() {  // 最初はvirtualなしで実行
10         cout << “ CZukei::OnDraw() ” << endl;
11     }
12 };
13 class CTen : public CZukei
14 {
15     CPoint  pts;  // CPointクラスはMFCにしかないのでサンプル実行のときはコメントアウトでOK
16 public:
17     virtual void OnDraw() {   // 派生クラスにvirtualは書かなくてもよいが慣例で書いた方が良い
18         cout << “ CTen::OnDraw() ” << endl;
19     }
20 };
21 class CSen : public CTen
22 {
23     CPoint  pte;
24 public:
25     virtual void OnDraw() {
26         cout << “ CSen::OnDraw() ” << endl;
27     }
28 };
29
30 int main()
31 {
32     vector<CZukei*>  v;
33     vector<CZukei*>::iterator  it;
34
35     v.push_back( new CTen() );
36     v.push_back( new CSen() );
37     v.push_back( new CSen() );
38
39     for ( it=v.begin(); it!=v.end(); ++it )
40         (*it)->OnDraw();
41
42     for ( auto a : v )  // C++11ではこんな風にも書けます
43         delete  a;
44
45     return 0;
46 }
```

CZukei クラスを基底に、CTen と CSen クラスを派生させています。main() では、それらを格納する動的配列を 32 行目で宣言し、35〜37 行目で点を 1 つ、線を 2 つ、動的配列に追加しています。9 行目、17 行目、25 行目の virtual がない状態で実行すると

```
> ./hasei1
CZukei::OnDraw()
CZukei::OnDraw()
CZukei::OnDraw()
```

と出力されます。32 行目の宣言は CZukei*型の動的配列として宣言しており、その OnDraw() 関数が呼ばれるのは当たり前の話です。

次に、9 行目、17 行目、25 行目の virtual を入れて実行してみましょう。

```
> ./hasei2
CTen::OnDraw()
CSen::OnDraw()
CSen::OnDraw()
```

C 言語の場合は、点データの場合どうるす、線データの場合こうする、という場合分けが必要でしたが、C++では（クラスの設計次第ですが）その必要がなくなります。派生クラスと仮想関数がポリモーフィズムの真髄ですが、これが C 言語しか知らない人（学生）が苦労する部分かも知れません。”C と C++は似て非なるもの ”とはこういうことですね。

6 おわりに

私の当初の目的はメモリリークを改善することだけでした。C++プログラミングにおいての処理記述は、自由度のある書き方や C 言語風に書いてもコンパイルが通ることから、他人の書いたコードはなかなか理解しづらいのは承知しているつもりです。しかし、Kodatuno のソースコードはクラスの設計思想すら読めない Why？の連続でした。民間企業が公開しているライブラリなら、このレポートを書くことはなかったでしょう。単にそのライブラリは使わないという結論に至ったはずです。大学で開発されているからこそ、あえて苦言を呈した次第です。C++で書く以上は C++の流儀で書かないと逆に混乱すると思います。学生の目に触れるならなおさらで、そこは機械系だろうが情報系だろうが関係ありません。正しい知識を身に付けさせてこその教育研究機関だと思います。

冒頭でも述べましたが、決してこれが正解というわけではありませんし、ましてやそれを強要することもありません。オープンソースということを理解しつつ、こちらで細々と修正を加えていきたいと思います。

Kodatuno の今後益々の発展に期待するとともに、本レポートが少しでもその一助になれば幸いです。

参考文献

1) 柴田望洋：C プログラマのための C++入門，ソフトバンクパブリッシング，2000 年 9 月
2) MSDN CRT のセキュリティ機能：https://msdn.microsoft.com/ja-jp/library/8ef0s5kh.aspx
3) 稲葉一浩：Boost C++ Library プログラミング，秀和システム，2004 年 6 月
4) 眞柄賢一：Boost C++ Library の紹介，舞鶴工業高等専門学校情報科学センター年報，第 40 号，2012 年 3 月