# C 言語への静的なバインド機構の実装

荻　原　剛　志

（平成 22 年 9 月 22 日提出
平成 22 年 12 月 6 日修正）

**要　旨**

オブジェクト指向のスタイルによらない手続き型言語を用いた開発では，抽象度の高い記述が難しいためにモジュール間の依存度が高い．これまでの研究で，共有変数を使って複数のモジュールを連携させる静的な言語機構，タップを提案した．タップを利用することによって，具体的な手続き名を互いのモジュールに記述する必要がなくなり，独立性を高めることが可能である．また，オブジェクト指向におけるデザインパターンであるオブザーバパターンを手続き型言語でも利用できるようになる．本研究では，タップの機構を C 言語で利用できるような処理系を試作し，実際にソフトウェアを記述してタップの効果を確かめた．その結果，関数間の静的な連携に関しては問題がなく，想定通りに動作することが確認できた．しかし，動的な結合が制約されているため，実際の応用に対する記述力が十分ではないことが明らかとなった．

**キーワード：** ソフトウェアモジュール，グルーコード，バインド機構，手続き型言語，デザインパターン

## 1.　はじめに

C 言語をはじめとする手続き型のプログラミング言語では，複数のモジュールを相互に関連づけようとする場合，一方のソースコードに他方の手続き名などの固有の情報を記述するか，双方の情報を利用して仲介役を果たすルーチンを付け加えるのが普通である．以下では，ソフトウェアを構成する主要なモジュールやルーチン，GUI 部品などを「部品」と総称し，これらを相互に結合するために記述されるコードをグルーコードと呼ぶ．

グルーコードはソフトウェアを構成する要素として必要ではあるが，特定のグルーコードの存在を前提に作成されたモジュールは独立性，再利用性が低くなりやすい．ソフトウェアの改変や既存のソースコードの再利用の際には，不要なグルーコードを削除したり，新しい用途向けの別のグルーコードを書き足したりする作業に多くの労力が必要となる．

一方，オブジェクト指向言語では，クラスの継承機構などを利用した抽象度の高い記述が行えることから，独立性の高い記述が可能なデザインパターンや，コンポーネント指向の開発手法が利用されている．しかし，これらの方法を手続き型言語にそのまま適用することは難しい．

本研究では，手続き型言語（本稿では非オブジェクト指向の言語の意味で使う）のプログラム

部品を対象とし，グルーコードの存在なしに相互を関連づけることができる言語機構，タップを提案している [1]．タップは，それぞれのモジュール内では静的な変数として記述することができる．モジュールをリンクして実行ファイルを構築する際，あるモジュールのタップと，別のモジュールのタップを結合することを指定できる．結合されたタップは，1 つの共有変数を介して情報を交換することができるほか，共有変数が他のモジュールによって更新された場合には，自らのモジュール内に記述しておいた関数を自動的に呼び出して通知してもらうことができる．

本稿では，タップ機能に関する実験を行うために作成した試作処理系について報告する．この処理系は，C 言語のソースコード内に特定の記法で書かれたタップの構文を前処理し，C の変数と関数の形に再構成すると同時に，タップの識別子や型に関する情報を抽出する．また，リンクに先だって，どのタップ同士を結合させるのかという情報をもとに，共有変数部分を実現するための C 言語，およびアセンブリ言語のコードを生成する．

以下，タップ機構を提案するに至った背景について述べ，タップの構文と今回の試作で用いた記法，実装方法について説明する．また，この処理系を利用して作成したソフトウェアの例を示し，現状の機能について議論を行う．

## 2.　タップ機構の提案

Mac OS X の Cocoa 開発環境 [2] では，MVC モデルのモデルに相当するオブジェクトをメッセージ通知の中心として扱い，ソースコードへの記述なしにオブジェクトの相互接続を実現できる．この技術を Cocoa バインディングと呼ぶ．本研究はこの機能を非オブジェクト指向の言語で実現する方法として着想されたものである．

オブジェクト指向言語によって実装された多くの GUI 環境では，通常，GUI 部品などからのイベントを，イベント処理メソッドを実装したオブジェクト（イベントリスナ）が取得する仕組みとなっている．Cocoa 環境ではこれを，GUI 部品がオブジェクトに対してメッセージを送ることとみなし，ターゲット–アクション・メカニズムと呼ぶ．これらの仕組みを使って，図 1 (a) のように，スライダが動かされた時にその値をオブジェクトに知らせるといった構造は容易に実現できる．一方，スライダを動かしても，テキストフィールドに直接入力しても値が指定でき，しかも，値の変化が両者の表示に反映されるといったインタフェースも一般的によく用いられている．しかし，イベントリスナ，あるいはターゲット–アクション・メカニズムを使ってこの関係を実現するためには，図 1 (b) のように仲介役となる別のオブジェクトを記述する必要がある．

Cocoa バインディングでは，連携しあう部品が共有するプロパティを，モデルと呼ぶオブジェクトに持たせ，汎用的に利用できるコントローラオブジェクトで管理する．図 1 (c) のように，

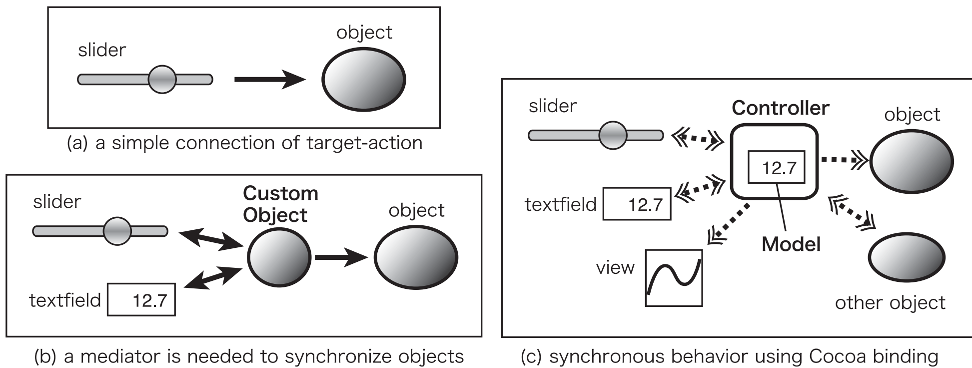


**図 1**　Cocoaバインディングの概念

モデルのプロパティには複数個の GUI 部品や他のオブジェクトを関連づける（バインドする）ことができる．バインドした個々の部品はプロパティを変化させることができ，逆にプロパティが変化した場合には，変更を知らせるメッセージを受け取ることもできる．

オブジェクト間のバインドのために既存のコードを変更したり，新たなコードを追加する必要はなく，通常は GUI 作成ツール上でバインドすることを指定するだけでよい．ただし，Cocoa バインディングは Objective-C 言語のランタイムを利用して実現 [3] しており，手続き型言語で同様な機能を実現するのは容易ではない．Mac OS X と同様に Objective-C をアプリケーション記述言語とする iOS でも，現状では Cocoa バインディングを利用することはできない．

本研究ではタップという機構を提案し，Cocoa バインディングにおけるオブジェクト間のバインドに類似した機能を，非オブジェクト指向言語のモジュール間で実現することを目指す．

タップは，それぞれのモジュール内では静的な変数として記述することができる．あるタップの静的変数は，モジュールのリンクの際，別のタップの静的変数と同一のアドレスに格納されるように指定できる．これを，タップを結合すると言う．タップは別のタップと結合しても，しなくてもよいし，2 個以上のタップが結合してもよい．結合されたタップは同じ静的変数を共有し，結合されていないタップはそれぞれに個別の静的変数を持つ．

結合されたタップは，共有する変数を介して情報を交換することができるほか，共有変数が他のモジュールによって更新された場合には，自らのモジュール内に記述しておいた関数を自動的に呼び出して通知してもらうことができる．

Cocoa バインディングは，典型的には MVC の要素オブジェクトを，コードを記述することなく結びつけることを目的としている．非オブジェクト指向の言語においても，タップ機構を用意しておくことによって，MVC のほか，オブザーバ，メディエータなどのデザインパターンのうち，静的な構造を持つシンプルなものであれば実装が容易になると期待される．

タップの結合の指定はソースコードとしては記述しないため，モジュールの独立性を損なう

ことがなく，グルーコードが増大するという問題を回避することができる．また，モジュール間の関係は実行前に決定されているため，リソースに関する制限が厳しい，組込みシステムのソフトウェアにも適用可能であると考えられる．

# 3.　タップ機構の構文

## 3.1　タップの宣言

タップの記述は他のモジュールと共有する変数の宣言と，タップの変更に伴って動作する手続きから成る．手続きはなくてもよい．タップを利用するモジュールは，外部変数と同様にして以下の宣言をソースコード中に記述する（イタリック，日本語の部分は実際のコードによって異なる）．

```
#tap type nameOfTap;
```

または次のように記述する．タップと共に記述されるコードはそのモジュールに固有のものである．タップの値を共有しても，実行コードは共有されない．

```
#tap type nameOfTap {
    実行可能なコード
}
```

プログラム内の複数個のタップがひとつの変数を共有できるが，それらのタップの名前は同じである必要はない．ひとつのモジュールに複数個のタップがあってもよい．共有される変数の実体がどのモジュールにあるのかについて留意する必要はない．

複数のタップ名をカンマで区切って連続させることもできる．

```
#tap type nameOfTap1, nameOfTap2, nameOfTap3 { ... }
```

この場合，複数のタップが実行コードを共有することになるが，それぞれのタップの間に変数の共有の関係はない．つまり，同じデータ型と実行コードを持つ別名のタップを個別に宣言したことと同じである．

ただし，このように実行コードを共有した場合，どのタップの変更に伴ってコードが実行されたのかは，実行コード内では把握できない．そこで，実行コード内では「隠された引数」として tap という変数が利用できるとした．tap は実行コードに対応する関数の引数として実装され，その値はコードを実行するきっかけとなったタップを指すポインタである．このポインタを参照することによって，実行コード内でタップごとに記述を分けることも可能である．

タップ名は変数名，関数名と同じ名前空間に属する．従って，C 言語では異なるモジュール

間での名前の衝突の危険がある点に注意が必要である．

### 3.2　値の参照

プログラムのコード中で，タップが共有している変数の値を使いたい場合は式として以下のように記述できる．

%?*nameOfTap*

なお，タップの実行コード中では，タップ自体を参照するためにタップ名を直接使うことができる．

### 3.3　通知

他のモジュールに対して何かのイベントを知らせたい場合，タップに対して以下のようにして通知を発する．パラメータの値がタップの新しい値となる．

%^*nameOfTap*（**値**）;

タップの宣言で指定した型が構造体の場合，メンバを指定することができる．ただし，この機能は今回の試作では実装されていない．

%^*nameOfTap*.**メンバ**（ **値** ）;

通知が送られたら，そのタップを共有している複数のモジュールでは，タップとともに記述されているコードが実行される．ただし，タップに通知を発したモジュールではコードは実行されない．

### 3.4　タップの結合

複数のタップを結合し，値を共有するための指示はソースコードには記述しない．ソースコードに記述した時点でグルーコードと同様，モジュールの独立性を損なう可能性があると考えたからである．また，タップは静的な機構であるとしたため，プログラムの実行によって結合させる必要はなく，リンク時に決定されていればよい．

今回の実装では，拡張子 con を持つ結合指定ファイルに次のような記述を行うことにした．

*nameOfTap1* = *nameOfTap2*;

このように，2 つのタップ名を “=” で結ぶ．3 個以上のタップを結合させる場合には “=” をはさんで複数個のタップ名を記述する．

*nameOfTap1* = *nameOfTap2* = *nameOfTap3*;

タップ名は関数名と同様に大域名であるため，モジュール名を指定する必要はない．

# 4.　実装方法

## 4.1　タップ宣言を変数宣言と関数に置き換える

コンパイルの前に前処理を行い，節 3 (1)「タップの宣言」で示したタップの宣言は，共有される変数の宣言と，実行コードに対応する関数定義に置き換える．例として，次のようなタップ宣言があったとする．

```
#tap double sliderA, sliderB {
  int t = (tap == &sliderA) ? 0 :  1;
  calc(t, *(double *)tap);
}
```

まず，タップの値を保持するための変数を宣言する．これはタップ名そのもので，大域名となるが，変数の実体はここでは定義しない．

```
extern double sliderA, sliderB;
```

さらに，それぞれのタップの値を変更した場合に呼び出される関数を指すポインタ変数を宣言する．この関数の本体はここでは定義しない．以下ではこの変数が指す関数を通知関数，変数を通知関数ポインタと呼ぶ．変数名はモジュール（ファイル）名とタップ名から生成する．

```
extern tap_notify_fp モジュール名_pretap_notify_sliderA;
extern tap_notify_fp モジュール名_pretap_notify_sliderB;
```

ここで，型 tap_notify_fp は関数を指すポインタ型であり，次のように宣言される．

```
typedef void (*tap_callback_fp)(void *);
typedef void (*tap_notify_fp)(tap_callback_fp, void *);
```

実行コードは次のような関数に置き換える．上で示した型 tap_callback_fp が，これらの実行コードの関数を指す．以下ではこの関数をコールバック関数と呼ぶ．関数名はモジュール名とタップ名から生成する．

```
void モジュール名_pretap_cback_sliderA(void *tap) {
  int t = (tap == &sliderA) ? 0 :  1;
```

```
  calc(t, *(double *)tap);
}
```

### 4.2　値の参照をタップ変数の参照に置き換える

3.2「値の参照」で述べたように，タップの保持する値を参照するには，式の中に次のように記述する．

```
%?sliderA
```

今回の試作システムでは，これをタップ名に置き換える．

```
sliderA
```

ただしこの方法では，参照だけではなく，値に変更を加えることも可能となってしまう．例えば次のような記述が可能であり，値が実際に変更されてしまう．

```
double p = %?sliderA = 0.0;
```

このため，実用レベルのシステムでは値の変更を禁止するようにする必要がある．例えば次のようにすれば，コンパイラがエラーを検出できる．

```
double p = *(const double *)&sliderA = 0.0;
```

### 4.3　通知をマクロ定義で置き換える

3.3「通知」で示したように，タップの値を変更し，必要に応じて通知を発するには次のように記述する．

%^**タップ名**（value);

試作システムでは，これを次のようなマクロの呼び出しで置き換える．

TAP_CALL（**タップ名**，**通知関数ポインタ**，**コールバック関数名**，value);

マクロは次のように定義されている．

```
#define TAP_CALL(t, ntfy, cbk, val) \
  if (t = (val), (ntfy == NULL)) ; else ntfy(cbk, &(t))
```

このマクロでは，まず値をタップの共有変数に代入する．次に，通知関数ポインタが空ポイ

ンタでなければ，通知関数を呼び出す．通知関数の引数はコールバック関数と，タップの共有変数へのポインタである．なお，わざわざ else 節に実行文を記述しているのは，マクロ全体がひとつの式として扱われて，末尾に “;” が付けられても問題が生じないようにするためである．

タップを結合せず，変数が共有されない場合には，通知関数を呼び出す必要はない．この場合には，通知関数ポインタには空ポインタが入れられており，通常行われる変数への値の代入に加えて比較が 1 回行われるだけである．この実装によって，不要な関数呼び出しを取り除くことができる．

タップの値を変更した場合，結合された複数のタップに付随するコールバック関数が順番に起動される．ただし，その呼び出しを行ったモジュールの関数は呼び出さないようにする必要がある．そこで，通知関数にコールバック関数を引数として渡し，その関数を呼び出さないことを伝える．コールバック関数自体がない場合には空ポインタを渡す．

### 4.4　タップの情報と実体のコード

モジュール内で宣言されたタップは，上記のように関数や変数の宣言に置き換えるだけではなく，対応する通知関数と共有変数の実体を用意する必要がある．

上記の 4.1 の例のようなタップが宣言されていた場合，試作システムはコードから次のような情報を生成し，拡張子 “tap” を持つファイルとして書き出す．含まれる情報は，タップ名，タップの保持するデータの型，コールバック関数名，通知関数ポインタの変数名である．

```
{ "sliderA", "double", "モジュールX_pretap_cback_sliderA",
  "モジュールX_pretap_notify_sliderA" },
{ "sliderB", "double", "モジュールX_pretap_cback_sliderA",
  "モジュールX_pretap_notify_sliderB" },
```

結合されるタップは 3.4「タップの結合」で示した方法で記述される．例として，タップ `sliderA` と `coeffa` が結合されることになっているとすると，これらの情報から，通知関数の実体を構成するために次のような関数定義が生成される．

```
double sliderA;
void モジュールX_pretap_cback_sliderA(void *);
void モジュールY_pretap_cback_coeffa(void *);
void tap_notify0000_f(tap_callback_fp _cb, void *_vp) {
  static tap_callback_fp _tb[] = {
    モジュールX_pretap_cback_sliderA,
    モジュールY_pretap_cback_coeffa,
```

```
    (tap_callback_fp)0
  };
  int i;
  for (i = 0; _tb[i]; i++)
    if (_tb[i] != _cb) _tb[i](_vp);
}
tap_notify_fp tap_notify0000 = tap_notify0000_f;
```

1 行目の変数宣言は，タップ sliderA と coeffa が共有することになる変数 sliderA の宣言である．続く 2 行は，それぞれのモジュールで定義されているコールバック関数の宣言である．4 行目からが通知関数の定義で，結合されたタップのコールバック関数を順番に呼び出すようになっている．上の例では共有しているのは 2 つのタップだが，より多数のタップが結合されてもよい．通知関数には，呼び出し元のモジュールのコールバック関数のポインタと，共有変数のポインタが引数として渡される．ここで，引数として渡されたポインタに一致するコールバック関数は呼び出さない．このことは 4.3 でも述べた通りである．また，結合されるどのタップにもコールバック関数が定義されていない場合，通知関数は生成されない．

最後の行は，各モジュールから通知関数を呼び出すため通知関数ポインタのための変数の定義である．

## 4.5　共有変数と通知関数ポインタのアドレス

ここまでで，各タップごとに，共有変数の宣言，通知関数ポインタの宣言，およびコールバック関数の定義が記述でき，さらに，結合されるタップの集合ごとに，共有変数の実体と通知関数の実体が定義できた．

次に，各タップにそれぞれ異なる名前で宣言されている共有変数と通知関数ポインタが，実際に存在する共有変数の実体，通知関数の実体を参照するようにしなければならない．上の例で言えば，タップ sliderA と coeffa を結合することを宣言しているため，共有変数の sliderA と coeffa はメモリ上の同じ変数を参照しなければならない．また，通知関数ポインタ（"**モジュール**X_pretap_notify_sliderA" など）も同じ通知関数を参照しなければならない．

C 言語には，異なる識別子に同一のアドレスを割り当てる方法が用意されていない．そこで，この部分だけはアセンブリ言語の記述を用いる．タップ sliderA と coeffa を結合する例では，以下のようなコードを生成する．

```
.globl _FunctionCtrl_pretap_notify_sliderA
    .set   _FunctionCtrl_pretap_notify_sliderA, _tap_notify0000
.globl _QuadraticView_pretap_notify_coeffa
```

```
        .set   _QuadraticView_pretap_notify_coeffa, _tap_notify0000
    .globl _coeffa
        .set   _coeffa, _sliderA
```

ここに示したのは gas (GNU Assembler) によるコードである．1 行目では，C 言語の識別子である FunctionCtrl_pretap_notify_sliderA が大域名であること，2 行目ではこの名前が tap_notify0000 と同じであることを指示している．これによって，結合されたタップの共有変数と通知関数ポインタがメモリ上の同じアドレスを指すように指定できる．

ただし，実装を行った Mac OS X ではリンカ ld の仕様に制限があり，テキスト領域に関しては上記の通りでよいが，データ領域についてはデータ境界（バウンダリ）の計算を行う必要がある．コード生成に関しては，この部分についてのみ，自動的に行うことができていない．

## 5.　使用例

タップを用いて部品を組み合わせたプログラムの例を示す．

このプログラムは文献 [3] で，Cocoa バインディングを用いた例題として掲載しているものである．ここでは Cocoa バインディングで実装されている部分をタップの記述に書き換え，同様に動作することを確認する．ソースコードは Objective-C で記述されているが，Objective-C は C 言語の上位互換の言語であることから，本研究で提案しているタップを用いたコードを含めることが可能である．ただし，タップは静的に定まるものであるため，本来はインスタンス変数同士の結合として記述されていた部分を大域変数で記述しなおす必要があった．

プログラムの外観を図 2 に示す．このプログラムは二次関数

$$y = ax^2 + bx + c$$

のグラフを描画するもので，式中の係数 a，b，c はウィンドウ下部のスライダを用いて指定できる．また，描画領域をドラッグしてグラフを上下左右に移動させることができる．グラフの移動に伴って係数 a，b，c の値が変化するが，その変化した値はスライダの値に反映される．つまり，係数はスライダによっても，グラフのドラッグによっても変更される．

図 3 はこのプログラムの，描画部分のオブジェクトの関係を表したものである．オブジェクト FunctionController はスライダが動かされたら係数の値を変化させるが，一方，係数の値が変更された場合はスライダの表示を更新する．同様に，QuadraticView はグラフの描画とマウス操作による係数の変更を管理している．この 2 つのオブジェクトがそれぞれ保持している二次関数の係数をタップとして表し，結合することによって，スライダの変化をグラフの描画に，また逆に，グラフのドラッグをスライダの値の変化に反映させることができる（図中の点線部分）．

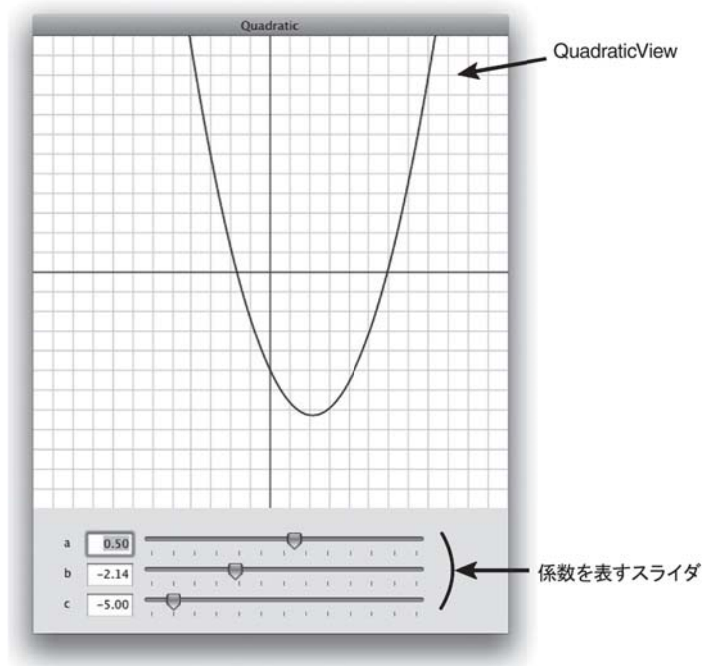


**図 2**　サンプルプログラムのインタフェース

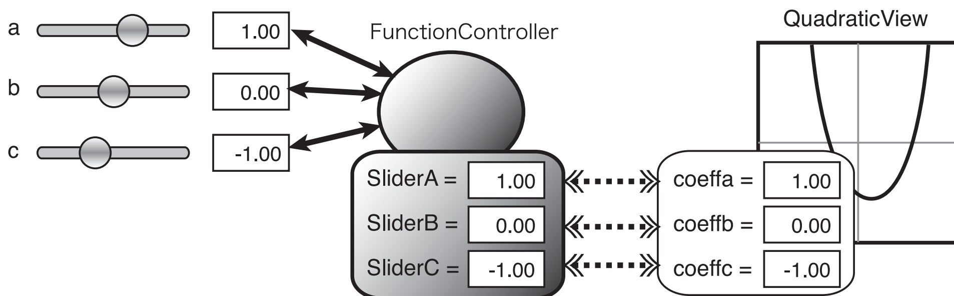


**図 3**　サンプルプログラムの概要

しかも，それぞれのオブジェクトのコードには，どのようなオブジェクトと連携しあっているのかという記述を行う必要が一切ない．

リスト 1 に FunctionController 内のメソッドと関数の記述の一部を示す．タップは sliderA, sliderB，sliderC の 3 つで，二次関数の係数に対応している．タップのコールバック関数は共有されており，値が変更された場合，それがどの係数に対する変更なのかを調べ，対応するスライダの値を変更するためにメソッド `sliderChanged:value:` を呼び出す．スライダが動かされた場合は，メソッド `sliderChanged:` が呼び出され，タップの値も変更される．

```
- (void)sliderChanged:(int)t value:(double)v
{
    [sl[t] setDoubleValue: v]; // スライダの値を変更
    [tx[t] setDoubleValue: v]; // テキストの値を変更
}

#tap double sliderA, sliderB, sliderC {
    int t = 0;
    if (tap == &sliderB) t = 1;  // tap は変更されたタップを示す
    else if (tap == &sliderC) t = 2;
    [sharedCtrl sliderChanged:t value:*(double *)tap];
   // この関数は大域関数なので、唯一のインスタンスを呼び出す必要がある
}

static void sliderValueChanged(int t, double v) { // タップを変更
    if (t == 0) %^sliderA(v);
    else if (t == 1) %^sliderB(v);
    else %^sliderC(v);
}

- (void)sliderChanged:(id)sender { // スライダが変更されたら呼び出される
    int t = [sender tag];
    double v = [sender doubleValue];
    [tx[t] setDoubleValue:v];
    sliderValueChanged(t, v);
}
```

**リスト 1**　タップの記述例 (FunctionController)

```
sliderA = coeffa;
sliderB = coeffb;
sliderC = coeffc;
```

**リスト 2**　タップの結合の指定

リスト 2 はタップの結合を指定する記述である.

前処理が済んだ後，リスト 1 のタップの宣言部分と，関数 `sliderValueChanged()` がどのように書き換えられているかをリスト 3 に示す.

## 6.　議論

### 6.1　現状のタップの問題点

前処理系の実装を行い，タップ機構を使ったプログラミングが可能であることを確認した．しかし，実際のアプリケーションを記述した経験から，いくつかの問題点が明らかとなってきた.

まず第一に，当初想定していた静的なタップだけでは，柔軟なプログラミングが困難である

```
extern double sliderA, sliderB, sliderC;  /* tap */
extern tap_notify_fp FunctionCtrl_pretap_notify_sliderA;  /* tap */
extern tap_notify_fp FunctionCtrl_pretap_notify_sliderB;  /* tap */
extern tap_notify_fp FunctionCtrl_pretap_notify_sliderC;  /* tap */
void FunctionCtrl_pretap_cback_sliderA(void *tap) /* tap callback */ {
    int t = 0;
    if (tap == &sliderB) t = 1;
    else if (tap == &sliderC) t = 2;
    [sharedCtrl sliderChanged:t value:*(double *)tap];
}

static void sliderValueChanged(int t, double v) {
  if (t == 0) TAP_CALL(sliderA, FunctionCtrl_pretap_notify_sliderA,
      FunctionCtrl_pretap_cback_sliderA, v);
  else if (t == 1) TAP_CALL(sliderB, FunctionCtrl_pretap_notify_sliderB,
      FunctionCtrl_pretap_cback_sliderA, v);
  else TAP_CALL(sliderC, FunctionCtrl_pretap_notify_sliderC,
      FunctionCtrl_pretap_cback_sliderA, v);
}
```

**リスト 3**　前処理が終わったときのタップの記述例

という点である．静的なタップの利点として，実行中に動的に結合させる必要がないため，結合のためのコードが新たなグルーコードとなることがないということと，特に実装上の制約が厳しい組み込みシステムで，システム起動時の速度向上とメモリ管理の簡略化に寄与するということを挙げることができる．しかし，C 言語のような非オブジェクト指向言語であっても，構造体などを動的に複数個生成して，それらの間の連携によってシステムを動作させることがある．このような場合に対応できないのでは，記述能力の点で問題があると言わざるを得ない．

第二に，複数のタップの値を同時に更新した時，対応する処理が必ず変更の回数分発生してしまうという点が問題になることがある．例えば，前節で挙げたグラフ描画のアプリケーションにおいて，表示を初期状態に戻すという操作をする場合には複数の係数の値が同時に変更される．これによって，3 つのタップのコールバック関数が繰り返し実行されてしまい，まったく同じグラフの描画が 3 回起きることになる．従って，必要に応じて，複数のタップをまとめて扱ったり，処理を省略したりすることができれば，より効率的な記述を行うことが可能となると考えられる．

### 6.2　Cocoaバインディングとの比較

第 2 章でも述べたように，本研究は Cocoa バインディングに類似する機能を非オブジェクト指向の言語で実現する方法として着想されたものである．

タップは，Cocoa バインディングと同様に，モデルに相当するデータを共有し，通知を発することで通信できるが，手続き型言語の静的なリンクで実現する．今回の実装実験で，タップ

を実現するために必要な実行コードは極めて小さく済むことが確認できた．また，アセンブリ言語の記述に関して改善すべき点は残されているものの，標準的な C 言語が動作する環境であればタップを動作させることができる見通しがついた．これらの点は，Mac OS X 上でのみ利用可能な Cocoa バインディングにはないメリットである．

今後は，組込みシステムのような制約の厳しい環境で有用かどうかについても，実際に確認を進める予定である．

### 6.3　並列プログラミングでの利用

マルチスレッドを用いる場合には，タップのコールバック関数を他のスレッドで実行させることが望ましい場合がある．タップの相互排除や同期についても検討しなければならないであろう．並列にプログラム同士が通信し合うためのプログラミング言語上の機構としては，Ada のランデヴや OCCAM [4] のチャネルなどがある．ただしこれらは，その通信路を使う相手同士が連携して動作することを前提として，最初からコーディングしておく必要がある．そのため，本研究で目的としている，グルーコードの軽減には結びつかない．

なお，組込み向けにも利用できることを念頭に置くのであれば，Lua [5] 言語のようにコルーチンとして動作するような拡張を考えるのも現実的であろう．

## 7.　まとめ

文献 [1] で提案した静的なタップ機構について，実際に動作可能な試作システムを作成し，タップを利用したプログラムの記述を試みた．本稿では，試作システムの概要について記した．

システムの試用の結果，当初想定していた静的なタップの機能は実現可能であることが確認できた．一方，静的なタップだけでは実用的なソフトウェアを開発するための記述力に問題があることが判明したため，現在は，動的な結合を許すタップへと仕様を拡張するための検討を行っている．

## 謝　辞

本研究は平成 21 年度京都産業大学総合研究支援制度（支援番号 442），および科学研究費補助金（基盤(C), 22500038）の支援を受けて行われた．

## 参　考　文　献

[1]　荻原剛志，“手続き型言語に適用可能なモジュールのバインド方式について”，情報処理学会研究報

告，2008-SE-162 (6), 2008.

[2] Apple Inc., “Cocoa Bindings Programming Topics”, http://developer.apple.com/library/mac/documentation/Cocoa/Conceptual/CocoaBindings/CocoaBindings.pdf (2009).

[3] 荻原剛志，“詳解 Objective-C 2.0”，ソフトバンククリエイティブ，2008.

[4] INMOS Limited, “OCCAM Programing Manual”, 1984.（「occam プログラミングマニュアル」，啓学出版，1984）

[5] R. Ierusalimschy, L. H. de Figueiredo, and W. Celes, “Lua 5.1 Reference Manual”, Lua.org, 2006.

# Implementation of a Static Binding Mechanism in C

Takeshi OGIHARA

**Abstract**

In procedural programming style, software modules tend to depend each other. Comparing to object-oriented programming, it is difcult to describe abstract modules. In the previous work, a new binding mechanism called “tap” was introduced. A tap can connect modules each other using a shared variable. With taps, the independency of modules can increase, because identiers of procedures or functions do not have to be written in other modules. Additionally, one of object-oriented design patterns, the observer pattern is available with taps. This paper reports the trial implementation of taps in C. Some programs written using taps showed that taps were effective to make static relationships between functions, and that, on the contrary, taps could not describe dynamic data structures enough as expected.

**Keywords:** Software Modules, Glue Code, Binding Mechanism, Procedural Programming Languages, Design Patterns