第4章

クラスの中で静的/動的な配列を使う

FIRフィルタのプログラムの作成

本章では，クラスを利用してFIRフィルタのプログラムを作成する．FIRフィルタを実現するためには配列が必要になるので，作成するクラスではこの配列をメンバにする．ここでは，コンパイル時にサイズが決定される静的な配列を使うクラスと，実行時にサイズが決定される動的な配列を使うクラスの，二つのケースについて紹介する．

1 FIRフィルタについて

最初に，FIRフィルタとその構成方法について簡単に説明する．

FIRフィルタとは，インパルス応答の継続時間が有限なフィルタのことである．FIRフィルタの入力信号を$x[n]$，出力信号を$y[n]$，フィルタの係数をh_m($m=0, 1, \cdots, M$)とすると，入出力の関係は次の式[注1]で表すことができる．

$$y[n]=\sum_{m=0}^{M} h_m\, x[n-m] \quad \cdots\cdots (1)$$

この式の中で使っているMの値を，このフィルタの次数という．このフィルタの伝達関数は，次のようになる[注2]．

$$H(z)=\sum_{m=0}^{M} h_m\, z^{-m} \quad \cdots\cdots (2)$$

この伝達関数で，$z=\exp(j\omega T)$[注3]とおいたものが，フィルタの周波数特性を表す関数である周波数応答になる．したがって，このフィルタの周波数応答は次のようになる．

$$H(z)=\sum_{m=0}^{M} h_m \exp(-j\omega T m) \quad \cdots\cdots (3)$$

このフィルタを実現する場合，いくつかの構成方法がある．代表的な構成法として，直接形(direct form)と転置形(transposed form)がある[注4]．それらのブロック図を**図1**(**a**)，(**b**)に示す．また，**図1**(**c**)にはブロック図を描くときの要素を示す．

直接形のブロック図は，式(1)の差分方程式と1対1に対応している．一方，転置形のブロック図は，式(1)を次のように変形したものに対応する．

$$\begin{cases} u_M[n]=h_M x[n] \\ u_{M-1}[n]=h_{M-1}x[n]+u_M[n-1] \\ \quad\vdots \\ u_1[n]=h_1 x[n]+u_2[n-1] \\ u_0[n]=h_0 x[n]+u_1[n-1] \\ y[n]=u_0[n] \end{cases} \quad \text{———} (4)$$

2 FIRフィルタのクラス ——静的な配列を使う場合

最初に，FIRフィルタで使用する配列を静的に，つまりコンパイル時にサイズが決まるようなクラスを作成する．

注1：このような式を差分方程式(difference equation)と呼ぶ．

注2：信号$x[n]$のz変換を$X(z)$，信号$y[n]$のZ変換を$Y(z)$とし，扱う信号は$n \geqq 0$の区間で定義されるものとする．このとき伝達関数$H(z)$は，次のように定義される．

$H(z)=Y(z)/X(z)$

一方，z変換は線形な変換であり，$f[n]$のz変換を$F(z)$とすると，次の関係が成り立つ．

$f[n-1]$ のz変換は$F(z)z^{-1}$

$f[n-2]$ のz変換は$F(z)z^{-2}$

⋮

$f[n-m]$のZ変換は$F(z)z^{-m}$

したがって，式(1)の両辺をz変換すると次のようになる．

$$Y(z)=\sum_{m=0}^{M} h_m X(z) z^{-m}$$

この式から式(2)を導くことができる．

注3：jは虚数単位($j=\sqrt{-1}$)を，ωは角周波数を，Tは標本化間隔を表す．

注4：そのほかに，縦続形や格子形などがあるが，これらは特別な場合を除いてあまり使われない．

見本

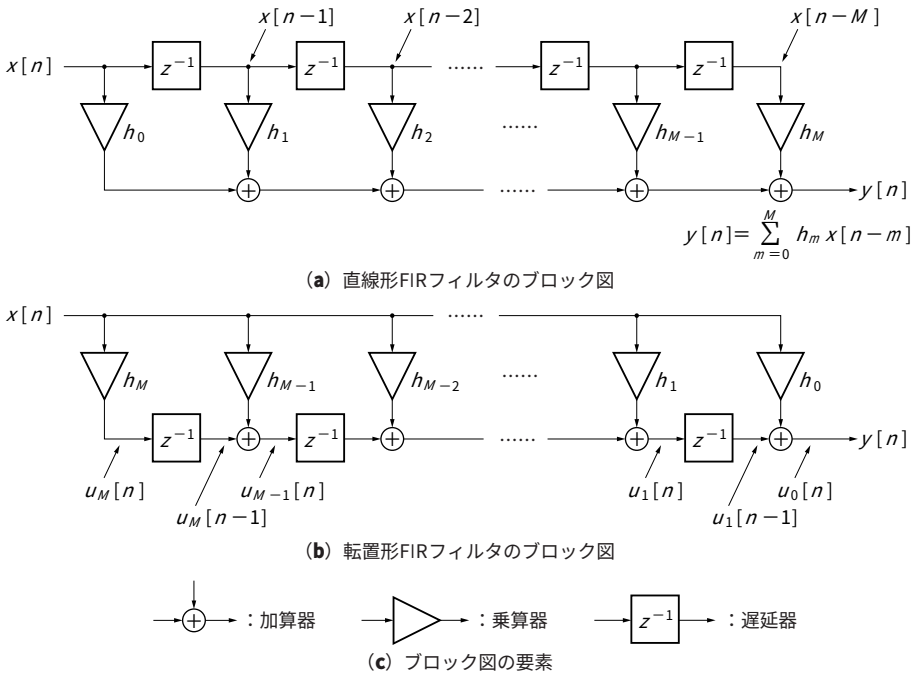


図1
FIRフィルタの代表的な構成方法

● FIRフィルタ用のクラス

▶ 直接形の場合

直接形FIRフィルタでは，過去の入力信号を保存しておくための配列が必要になる．この配列は，**図1(a)**の遅延器に対応している．FIRフィルタの次数をMとすると，$M+1$のサイズの配列が必要になる．

リスト1(FIRfixedOrderDirect.hpp)には，直接形FIRフィルタのためのクラスFIR_Directを示す．このクラスは，FIRフィルタで使用する配列のサイズを静的に決めるようになっているので，サイズをコンストラクタの引き数で与えることはできない．そのため，このクラスFIR_Directはテンプレート・クラスとして作成し，テンプレート引き数によって配列のサイズを決定するための数値を与えるようにする．

ところで，テンプレート・クラスを定義する際は，クラス名の前にtemplateというキーワードと<と>で囲まれた中に引き数のリストを配置する．引き数のリストの書き方には2通りの方法がある．

(1) template <class T1, class T2, …>

よく見るのはこの形式で，classはキーワードである．T1，T2の部分は，任意の名前を書くことができる．

(2) template <Type1 constant1, Type2 constant2, …>

Type1，Type2はデータの型を表すもので，intやfloatのような組み込みのデータ型名のほかに，クラス名を書くこともできる．

constant1やconstant2としては定数を書く．ここに変数を書くとエラーとなる．

＊　＊　＊　＊

ここでは，class FIR_Directの前のテンプレート引き数は<int ORDER>となっており，2番目の形式を使っている．コンストラクタの引き数は定数ではないので，コンストラクタの内部でその引き数に基づいて配列のサイズを静的に決定するということはできない．しかし，テンプレートの引き数は定数なので，コンストラクタの内部で，テンプレート引き数に基づいてサイズを静的に決定する，つまりコンパイル時に決定することが可能になる．

このクラスの非公開部では，二つのメンバが宣言されている．

ポインタhmはFIRフィルタの係数に対応するポインタで，ポインタ自身がconstであるとともに，その内容もconstになっている．

配列unはFIRフィルタの遅延器に対応するもので，そのサイズはテンプレート引き数に基づいてコンパイル時に決定される．

公開部では，コンストラクタとFIRフィルタの処理を行うメンバ関数Execute()が宣言されている．これらの処理は，クラスの外部で定義されている．

見本

リスト1　直接形FIRフィルタのクラス――次数を静的に設定(FIRfixedOrderDirect.hpp)

```
//-----------------------------------------------------
//    Class for direct-form FIR filter
//-----------------------------------------------------
#ifndef MK_FIR_Direct

#include "AIC23.hpp"

template<int ORDER> class FIR_Direct
{
private:
    const float *const hm;
    float un[ORDER+1];
public:
    FIR_Direct(const float hk[]);   // Constructor
    inline float Execute(const float xin);
};

template<int ORDER>
FIR_Direct<ORDER>::FIR_Direct(const float hk[]): hm(hk)
{
    for (int k=0; k<=ORDER; k++) un[k] = 0.0;
}

template<int ORDER>
inline float FIR_Direct<ORDER>::Execute(const float xin)
{
    float acc = 0.0;
    un[0] = xin;
    for (int k=0; k<=ORDER; k++) acc = acc + hm[k]*un[k];
    for (int k=ORDER; k>0; k--) un[k] = un[k-1];
    return acc;
}

#define MK_FIR_Direct
#endif
```

FIRフィルタの係数に対応するポインタ，ポインタ自身およびその指す内容がconst

FIRフィルタの遅延器に対応する配列

FIRフィルタの係数に対応するポインタの初期設定

遅延器に対応する配置をクリア

直接形FIRフィルタを実行

積和の計算

遅延器のデータの移動

コンストラクタの定義で，：hm(hk)という記述はメンバの初期設定を行うための構文で，hmにコンストラクタの引き数として与えられるhkを設定する．ポインタhmはconstなので，これ以外の方法でhmを初期化することはできない．

コンストラクタでは，FIRフィルタの遅延器に対応する配列をクリアするという処理が行われる．

メンバ関数Execute()は，FIRフィルタの処理を実行する．このメンバ関数は，二つのforループをもっている．一つ目のforループは式(1)の積和の計算に相当するもので，二つ目のforループは遅延器のデータの移動に相当する．実行結果は，戻り値として呼び出し側へ返される．

▶ 転置形の場合

転置形FIRフィルタでは，**図1(b)**に示すように計算結果を保存するための配列が必要になる．FIRフィルタの次数をMとすると，この配列のサイズは$M+1$になる．

リスト2(FIRfixedOrderTransposed.hpp)に転置形FIRフィルタのためのクラスFIR_Transposedを示す．このクラスはメンバ関数Execute()が異なるほかは，**リスト1**のクラスFIR_Directと同じである．メンバ関数Execute()はFIR_Directと異なり，配列に格納されているデータの移動は必要ないので，forループは一つだけになっている．

表1　リスト3のFIRフィルタの設計時に与えたパラメータ

次 数	100	
標本化周波数(kHz)	48	
	帯域1(通過域)	帯域2(阻止域)
下側帯域端周波数(kHz)	0.0	1.6
上側帯域端周波数(kHz)	0.8	24.0
利 得	1	0
重 み	1	1

● FIRフィルタ用クラスの使用例

リスト3に，クラスFIR_Directの使用例(FIR_LPF.cpp)を示す．また，このフィルタの係数を求める際に与えたパラメータを**表1**に示す．係数は大部分を省略しているので，全体については本書付属のCD-ROMを参照してほしい．このフィルタの振幅特性を**図2**に示す．

このリストには二つのmain()関数があり，片方はコメントになっているが，こちらも正しいプログラムである．

最初のコメントになっていないmain()関数では，

見本

リスト2　転置形FIRフィルタのクラス――次数を静的に設定(FIRfixedOrderTransposed.hpp)

```
//-----------------------------------------------------
//    Class for transposed-form FIR filter
//-----------------------------------------------------
#ifndef MK_FIR_Transposed

#include "AIC23.hpp"

template<int ORDER> class FIR_Transposed
{
private:
    const float *const hm;
    float un[ORDER+1];
public:
    FIR_Transposed(const float hk[]);   // Constructor
    inline float Execute(const float xin);
};

template<int ORDER>
FIR_Transposed<ORDER>::FIR_Transposed(const float hk[]): hm(hk)
{
    for (int k=0; k<=ORDER; k++) un[k] = 0.0;
}

template<int ORDER>
inline float FIR_Transposed<ORDER>::Execute(const float xin)
{
    for (int k=0; k<ORDER; k++) un[k] = hm[k]*xin + un[k+1];
    un[ORDER] = hm[ORDER]*xin;
    return un[0];
}

#define MK_FIR_Transposed
#endif
```

この部分は，名前がFIR_Directから FIR_Transposedに変わることを除くと，**リスト1**と同じ

転置形FIRフィルタを実行

積和の計算

クラスFIR_Directのオブジェクトをもっとも単純な方法で宣言している．一方，コメントになっているほうのmain()関数では，クラスFIR_Directのオブジェクトを配列の要素として宣言している．

なお，クラスFIR_Transposedは，クラスFIR_Directと同じように使うことができる．つまり，FIR_Directの部分をFIR_Transposedに変更するだけで，そのほかはまったく変更する必要はない．そのため，使用例は割愛する．

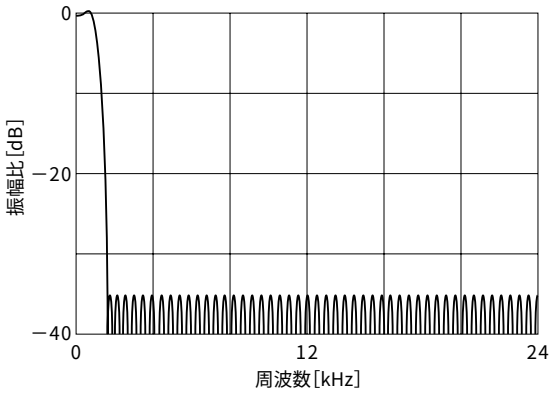


図2　リスト3のFIRフィルタの特性

3 FIRフィルタのクラス――動的な配列を使う場合(Ⅰ)

リスト1，**リスト2**のクラスは，いずれも使用する配列のサイズを静的に設定するため，このクラスを使ってプログラムを作る際には柔軟性に欠けている．たとえば，**リスト3**のコメントの部分のmain()関数のように，オブジェクトを配列の要素にする場合を考えてみよう．このリストのように，フィルタの次数が2チャネルともに同じであれば問題ないが，二つのチャネルで次数が違う場合は，このようなプログラムを作ることはできない．つまり，テンプレート引き数の値が異なれば，型名も異なっているものとみなされるので，配列として扱うことはできない．もっとも，次数の少ないほうのフィルタの係数の一部を0にして，次数を同じにすれば，プログラムを作ることは可能である．しかし，メモリと処理時間をむだに使うことになる．

このクラスに柔軟性をもたせるためには，次数の指定をテンプレートの引き数ではなく，コンストラクタの引き数で与えるようにすればよい．その場合は，クラスの中の配列のサイズを静的には決めることはできないので，配列のサイズを動的に決定するようなプロ

見本

リスト3　直接形FIRフィルタのクラスFIR_Direct<>の使用例(FIR_LPF.cpp)

```
//-----------------------------------------------------------------------
//  FIR low pass filter, using class of direct-form FIR filter
//      sampling frequency      48.0 kHz
//      passband edge frequency  0.8 kHz
//      stopband edge frequency  1.6 kHz
//      order                   100
//      deviation in passband    0.01733007 (  0.14923759 dB)
//      deviation in stopband    0.01733007 (-35.22399568 dB)
//  Designed by Parks-McClellan method
//-----------------------------------------------------------------------
#include "FIRfixedOrderDirect.hpp"

const int ORDER = 100;
const float hn[ORDER+1] = {
         9.61043138e-03,  1.95013017e-03,  2.05004161e-03,  2.08121602e-03,
         2.03403023e-03,  1.90021648e-03,  1.67303406e-03,  1.34854760e-03,

                          ～省略～

         2.03403023e-03,  2.08121602e-03,  2.05004161e-03,  1.95013017e-03,
         9.61043138e-03};

//---------------------------------------------
int main()
{
    float ch_in[2], ch_out[2];
    AIC23 codec;

    FIR_Direct<ORDER> firL(hn), firR(hn);

    while(true)
    {
        codec.Read(ch_in);

        ch_out[0] = firL.Execute(ch_in[0]);
        ch_out[1] = firR.Execute(ch_in[1]);

        codec.Write(ch_out);
    }
}
//---------------------------------------------

/*
// Alternative example (using array of FIR_Direct objects)
int main()
{
    float ch_in[2], ch_out[2];
    AIC23 codec;

    FIR_Direct<ORDER> fir[2] =
            {FIR_Direct<ORDER>(hn), FIR_Direct<ORDER>(hn)};

    while(true)
    {
        codec.Read(ch_in);

        for (int n=0; n<2; n++)
            ch_out[n] = fir[n].Execute(ch_in[n]);

        codec.Write(ch_out);
    }
}
*/
```

FIRフィルタの係数

左チャネル用のFIR_Direct<>オブジェクト

右チャネル用のFIR_Direct<>オブジェクト

FIR_Direct<>オブジェクトの宣言

FIR_Direct<>オブジェクトの配列

これらは同じ値でなければならない

FIR_Direct<>のオブジェクトを配列の要素にした場合

グラムに変更する必要がある．

● **直接形FIRフィルタのクラスの例**

リスト4に，配列のサイズを動的に，つまりプログラムの実行時に決定するような直接形FIRフィルタのクラスFIRdynamicBeta(FIRvariableOrderBeta.hpp)[注5]を示す．

配列のサイズを動的に決定する場合，C++言語では演算子newを使う．演算子newはヒープ領域にメモリを確保する．このときに，メモリの確保に失敗し

注5：このクラスは完成したものではないので，名前の後ろにBetaを付けた．

見本

リスト4　直接形FIRフィルタのクラス――次数を動的に設定(FIRvariableOrderBeta.hpp)

```
//-------------------------------------------------------------
// Class for FIR filter using dynamic memory allocation
//    without default constructor and assign operator
//-------------------------------------------------------------
#ifndef MK_FIRdynamicBeta

#include <cassert>      // for assert()
#include <new>          // for new, delete, and set_new_handler()
#include "AIC23.hpp"

class FIRdynamicBeta
{
private:
    const float *const hm;  // constant pointer for constant array
                            //     of coefficients
    const int _order;       // order of filter
    float *un;              // pointer for delay elements
public:
    FIRdynamicBeta(const float hk[], int order);    // Constructor
    ~FIRdynamicBeta() { delete[] un; }              // Destructor
    inline float Execute(const float xin);          // Filtering
};

//-------------------------------------------------------------
// Contructor
FIRdynamicBeta::FIRdynamicBeta(const float hk[], int order)
          : hm(hk), _order(order)
{
    std::set_new_handler(0);
    un = new float[order+1];
    assert(un);             // abort if failure of new
    for (int k=0; k<=order; k++) un[k] = 0.0;
}
//-------------------------------------------------------------
// Execution of filtering
inline float FIRdynamicBeta::Execute(const float xin)
{
    float acc = 0.0;
    un[0] = xin;
    for (int k=0; k<=_order; k++) acc = acc + hm[k]*un[k];
    for (int k=_order; k>0; k--) un[k] = un[k-1];
    return acc;
}

#define MK_FIRdynamicBeta
#endif
```

FIRフィルタの係数に対応するポインタ，ポインタ自身およびその指す内容がconst

デストラクタ：FIRフィルタの遅延器に対応する配列の領域を解放

フィルタの次数の初期設定

フィルタの係数に対応するポインタの初期設定

この関数は名前空間stdで定義されているのでstd::を付ける必要がある

new演算子が処理に失敗したとき，NULLポインタを返すように設定

FIRフィルタの遅延器に対応する配列の領域を動的に確保

new演算子が処理に失敗したとき，NULLポインタが返されるので，その場合，ここでプログラムを強制終了する

遅延器に対応する配列をクリア

直接形FIRフィルタを実行

積和の計算

遅延器のデータの移動

た場合の対応を，本来であれば例外処理(try，catchを使う構文)を使って記述するべきである．

しかし，TMS320C6000シリーズ用のコンパイラは現在のところこの機能をサポートしていない[注6]．そこで，このクラスでは関数set_new_handler()[注7]とassert()を使い，領域が確保できない場合には，メッセージを出して異常終了するようにしている．

ところで，本章の第2節で示したFIRフィルタ用のクラスFIR_Direct，FIR_Transposedでは，静的な配列を使っているので，とくにデストラクタは必要としなかった．しかし，クラスFIRdynamicBetaはコンストラクタで演算子newを使って，配列を動的に確保しているので，デストラクタが必要になる．

このクラスのメンバ関数Execute()のソース・リストは，クラスFIR_Directのメンバ関数Execute()のソース・リストとまったく同じである．

● 直接形FIRフィルタのクラスの使用例

リスト5に，直接形FIRフィルタのクラスであるFIRdynamicBetaの使用例(FIR_LPFHPF1.cpp)を示す．この例では，二つのチャネルに，それぞれ異なる次数のフィルタを使っている．フィルタの係数を設計する際に与えたパラメータを**表2**に示す[注8]．これ

注6：筆者の使用しているCCS ver. 3.1に付属のコンパイラはVersion 5.1.0.

注7：関数set_new_handler()については，コラムを参照．

注8：本書付属のCD-ROMに収録されているFIRフィルタの設計プログラムでは，フィルタの種類を“通過域/阻止域型”とし，次数を奇数とした場合，高域通過フィルタは原理的に設計できない．そこで，周波数0は阻止し，標本化周波数の1/2は通過するようなヒルベルト変換器として設計した．

見本

リスト５　直接形 FIR フィルタのクラス FIRdynamicBeta の使用例(FIR_LPFHPF1.cpp)

```
//-----------------------------------------------------------------------
//  FIR low-pass and high-pass filter, using class of FIR filter
//      include file: "FIRvariableOrderBeta.hpp"
//      sampling frequency      48.0 kHz
//  LPF:
//      order                   14
//      passband edge frequency  1.0 kHz
//      stopband edge frequency  2.4 kHz
//      deviation in passband    0.20878513 (  1.64698221 dB)
//      deviation in stopband    0.20878513 (-13.60600852 dB)
//  HPF
//      order                    9
//      passband edge frequency  4.0 kHz
//      stopband edge frequency 24.0 kHz
//      deviation in passband    0.04065072 (  0.34609982 dB)
//  Designed by Parks-McClellan method
//-----------------------------------------------------------------------
#include "FIRvariableOrderBeta.hpp"

const int ORDER_L = 14;
const float hnL[ORDER_L+1] = {
         1.24200012e-01,  4.25554449e-02,  4.82532060e-02,  5.33194295e-02,
         5.75207089e-02,  6.06731866e-02,  6.26022288e-02,  6.32710556e-02,
         6.26022288e-02,  6.06731866e-02,  5.75207089e-02,  5.33194295e-02,
         4.82532060e-02,  4.25554449e-02,  1.24200012e-01};
const int ORDER_H = 9;
const float hnH[ORDER_H+1] = {
        -4.02139877e-02, -5.54550244e-02, -1.00071820e-01, -1.94805387e-01,
        -6.30801526e-01,  6.30801526e-01,  1.94805387e-01,  1.00071820e-01,
         5.54550244e-02,  4.02139877e-02};

//-----------------------------------------------------------------------
//          Example 1: legal
//-----------------------------------------------------------------------
int main()
{
    float ch_in[2], ch_out[2];
    AIC23 codec;
    FIRdynamicBeta firL(hnL, ORDER_L), firR(hnH, ORDER_H);

    while(true)
    {
        codec.Read(ch_in);
        ch_out[0] = firL.Execute(ch_in[0]);
        ch_out[1] = firR.Execute(ch_in[1]);
        codec.Write(ch_out);
    }
}

/*
//-----------------------------------------------------------------------
//          Example 2: legal
//-----------------------------------------------------------------------
int main()
{
    float ch_in[2], ch_out[2];
    AIC23 codec;
    FIRdynamicBeta fir[2] = { FIRdynamicBeta(hnL, ORDER_L),
                              FIRdynamicBeta(hnH, ORDER_H)};

    while(true)
    {
        codec.Read(ch_in);
        for (int n=0; n<2; n++) ch_out[n] = fir[n].Execute(ch_in[n]);
        codec.Write(ch_out);
    }
}
*/

/*
//-----------------------------------------------------------------------
//          Example 3: legal
//-----------------------------------------------------------------------
int main()
{
    float ch_in[2], ch_out[2];
    AIC23 codec;
```

左チャネル用低域通過フィルタの係数

右チャネル用高域通過フィルタの係数

左チャネル用のFIRdynamicBetaオブジェクト

右チャネル用のFIRdynamicBetaオブジェクト

FIRdynamicBetaのオブジェクトを配列の要素にした場合

FIRdynamicBetaオブジェクトの配列

見本

リスト5　直接形FIRフィルタのクラスFIRdynamicBetaの使用例(FIR_LPFHPF1.cpp)(つづき)

```
    FIRdynamicBeta *fir[2];

    fir[0] = new FIRdynamicBeta(hnL, ORDER_L);
    fir[1] = new FIRdynamicBeta(hnH, ORDER_H);

    while(true)
    {
        codec.Read(ch_in);
        for (int n=0; n<2; n++) ch_out[n] = fir[n]->Execute(ch_in[n]);
        codec.Write(ch_out);
    }
}
*/
-----------------------------------------------------------------------------
/*
//-----------------------------------------------------------------------
//          Example 4: illegal
//-----------------------------------------------------------------------
int main()
{
    float ch_in[2], ch_out[2];
    AIC23 codec;
    FIRdynamicBeta fir[2], firL(hnL, ORDER_L), firR(hnH, ORDER_H);

    fir[0] = firL;
    fir[1] = firR;

    while(true)
    {
        codec.Read(ch_in);
        for (int n=0; n<2; n++) ch_out[n] = fir[n].Execute(ch_in[n]);
        codec.Write(ch_out);
    }
}
*/

/*
//-----------------------------------------------------------------------
//          Example 5: illegal
//-----------------------------------------------------------------------
int main()
{
    float ch_in[2], ch_out[2];
    AIC23 codec;
    FIRdynamicBeta fir[2];

    fir[0] = FIRdynamicBeta(hnL, ORDER_L);
    fir[1] = FIRdynamicBeta(hnH, ORDER_H);

    while(true)
    {
        codec.Read(ch_in);
        for (int n=0; n<2; n++) ch_out[n] = fir[n].Execute(ch_in[n]);
        codec.Write(ch_out);
    }
}
*/
```

これはポインタの宣言であり，ここではオブジェクトが生成されないので，コンパイル・エラーは出ない．FIRdynamicBeta FIR[2]と宣言するとコンパイル・エラーが発生

FIRdynamicBetaのオブジェクトを動的に生成

fir[]はポインタなので，. ではなく -> を使う

これ以前は正しいプログラムの例

以下の例はコンパイル・エラーが発生する

FIRdynamicBetaはデフォルト・コンストラクタをもっていないので，コンパイル・エラーを発生

FIRdynamicBetaではオブジェクトを代入するための演算子 = が定義されていないので，正しい代入が行われない

FIRdynamicBetaはデフォルト・コンストラクタをもっていないので，コンパイル・エラーを発生

FIRdynamicBetaではオブジェクトを代入するための演算子 = が定義されていないので，正しい代入が行われない

らのフィルタの振幅特性を**図3**に示す．

このリスト5には五つのmain()関数があり，四つはコメントになっている．コメントになっていないmain()関数(Example 1)と，コメントになっているmain()関数の初めの二つ(Example 2，Example 3)は正しいプログラムである．しかし，後の二つ(Example 4，Example 5)はコンパイル・エラーが発生する．

その理由を考えてみよう．クラスFIRdynamicBetaではデフォルト・コンストラクタ[注9]が定義されていない．そのため，Example 4，Example 5のように初期化を伴わない配列の宣言，つまり，

```
FIRdynamicBeta fir[2]
```

のような宣言では必要な引き数を与えていないことになるため，コンパイル・エラーとなる．

さらに，FIRdynamicBetaクラスでは，代入演算

注9：デフォルト・コンストラクタとは，引き数をもたない，またはすべての引き数にデフォルト値が指定されているコンストラクタのことである．

見本

Column
set_new_handler()について

new演算子で領域確保を失敗した場合の動作は，C++言語が最初に開発されたときから何度か変更されている．最新のC++言語(標準C++：ISO/IEC 14882, Standard for C++ Programming Language)では，基本的に例外処理(try，catchを使う構文)で対処するような言語仕様になっている．

ところが，Code Composer Studio(CCS)のコンパイラは現在のところ例外処理をサポートしておらず[注A]，new演算子で領域確保を失敗した場合に，デフォルトでは直ちにプログラムを異常終了するようなコードを生成する．その結果，デバッグする際にどこで領域確保に失敗したのか調べようとしても，その場所がわからなくなる．そこで，領域確保に失敗した場所がわかるようにするためにはnew演算子で領域確保を失敗した場合にデフォルトの動作とは別の動作を行わせる必要がある．

そのために使うのが，関数set_new_handler()である．new演算子を使う前に，関数set_new_handler()を使って必要な設定を行っておくと，それに応じた動作を行う．そこで，関数set_new_handler()の使い方について簡単に紹介しておく．

関数set_new_handler()を使うためには，次のインクルード文が必要になる．

```
#include <new>
```

関数set_new_handler()を使った設定には，以下に示すように2通りの引き数の与え方があり，それぞれについてnew演算子で領域確保に失敗した場合の動作は異なる．

● 引き数に0を与える場合

領域確保に失敗した場合にNULLポインタが返される．**リスト4**では，この方法を採用している．

● 引き数に関数名を与える場合

引き数に，すでに宣言されている関数名を与えると，領域確保に失敗した場合にその関数が実行される[注B]．

注A：本書執筆時点における最新のC++コンパイラのマニュアル(TMS320C6000 Optimizing Compiler v6.0 Beta, 文献番号：spru187n)によると，例外処理もサポートされているようである．しかし，詳しい記述がないうえに，このマニュアル自体がpreliminaryとなっているので，今回は使用用を見送った．

注B：ただし，このときの動作を確認するプログラムを作成してみたが，領域の確保に失敗した場合に，引き数で指定された関数は実行されずに，異常終了の状態になった．

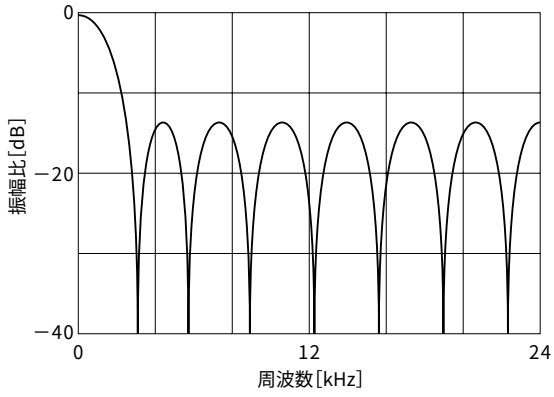
表2　リスト5，リスト7のFIRフィルタの設計時に与えたパラメータ

次 数	14	
標本化周波数(kHz)	48	
フィルタの種類	通過域/阻止域型	
	帯域1(通過域)	帯域2(阻止域)
下側帯域端周波数(kHz)	0.0	2.4
上側帯域端周波数(kHz)	1.0	24.0
利 得	1	0
重 み	1	1

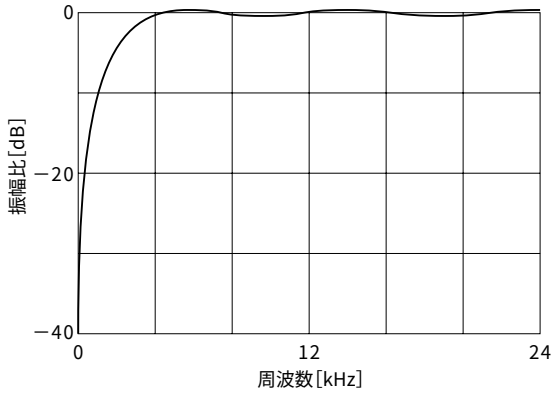
(a) 低域通過フィルタ

次 数	9
標本化周波数(kHz)	48
フィルタの種類	ヒルベルト変換器
	帯域1(通過域)
下側帯域端周波数(kHz)	4.0
上側帯域端周波数(kHz)	24

(b) 高域通過フィルタ



(a) 左チャネル用低域通過フィルタ



(b) 右チャネル用高域通過フィルタ

図3　リスト5，リスト7のFIRフィルタの特性

見本

リスト6　直接形FIRフィルタのクラス――次数を動的に設定(FIRvariableOrder.hpp)
デフォルト・コンストラクタと代入演算子=を定義したもの

```
//-----------------------------------------------------------------------
// Class for FIR filter using dynamic memory allocation
//    with default constructor and assign operator
//-----------------------------------------------------------------------
#ifndef MK_FIRdynamic

#include <cassert>      // for assert()
#include <new>          // for new, delete, and set_new_handler()
#include "AIC23.hpp"

class FIRdynamic
{
private:
    const float *hm;    // pointer for constant array of coefficients
    int   _order;       // order of filter
    float *un;          // pointer for delay elements
public:
    FIRdynamic(const float hk[] = NULL, int order = 0);  // Default constructor
    ~FIRdynamic() { delete[] un; }                       // Destructor
    FIRdynamic& operator=(const FIRdynamic& src);        // Assignment operator
    inline float Execute(const float xin);               // Filtering
};

                              ～省略～

//-----------------------------------------------------------------------
// Assignment of object
FIRdynamic& FIRdynamic::operator=(const FIRdynamic& src)
{
    _order = src._order;
    float *un_ = new float[_order+1];
    assert(un_);        // abort if failure of operator new
    delete[] un;        // free un
    un = un_;
    for (int k=0; k<=_order; k++) un[k] = 0.0;
    hm = src.hm;
    return *this;       // return object by reference
}

                              ～省略～
```


子operator=()が定義されていないので，Example 4，Example 5の次の代入文では，ポインタもそのまま代入されてしまう．

```
Example 4：fir[0] = firL;
Example 5：fir[0]
        = FIRdynamicTest(hnL, ORDER_L);
```

その結果，プログラムに不具合を発生させてしまう．この場合はコンパイル・エラーとはならないため余計にやっかいな問題である．

そのため，このクラスのオブジェクトを配列の要素として使いたい場合は，Example 3のように，つまり，

```
FIRdynamicBeta *fir[2]
```

のように，ポインタの配列として宣言しなければならない．このポインタの宣言では，クラスFIRdynamicBetaのオブジェクトが生成されるわけではないので，デフォルト・コンストラクタが定義されていなくてもコンパイル・エラーとはならない．

Example 3では，演算子newによりクラスFIRdynamicBetaのオブジェクトを生成している．また，演算子newは生成されたオブジェクトのポインタを返すので，そのまま，

```
fir[0]
    = new FIRdynamicTest(hnL, ORDER_L);
```

のように，ポインタの配列に代入することが可能になる．

4 FIRフィルタのクラス――動的な配列を使う場合(II)

第3節で作成した直接形FIRフィルタのクラスであるFIRdynamicBetaは，**リスト5**で見てきたように使い方が制限される．そこで，ここではその制限を外したクラスの作成と，その使用例について示す．

● 改良版FIRフィルタのクラス

クラスFIRdynamicBetaでその使い方が制限され

見本

リスト7　直接形FIRフィルタのクラスFIRdynamicの使用例(FIR_LPFHPF2.cpp)

```
//-----------------------------------------------------------------------
//  FIR low-pass and high-pass filter, using class of FIR filter
//      include file: "FIRvariableOrder.hpp"

                              ～省略～

#include "FIRvariableOrder.hpp"

                              ～省略～

//-----------------------------------------------------------------------
//          Example 1: legal
//-----------------------------------------------------------------------
int main()
{
    float ch_in[2], ch_out[2];
    AIC23 codec;
    FIRdynamic fir[2], firL(hnL, ORDER_L), firR(hnH, ORDER_H);

    fir[0] = firL;
    fir[1] = firR;

    while(true)
    {
        codec.Read(ch_in);
        for (int n=0; n<2; n++) ch_out[n] = fir[n].Execute(ch_in[n]);
        codec.Write(ch_out);
    }
}

/*
//-----------------------------------------------------------------------
//          Example 2: legal
//-----------------------------------------------------------------------
int main()
{
    float ch_in[2], ch_out[2];
    AIC23 codec;
    FIRdynamic fir[2];

    fir[0] = FIRdynamic(hnL, ORDER_L);
    fir[1] = FIRdynamic(hnH, ORDER_H);

    while(true)
    {
        codec.Read(ch_in);
        for (int n=0; n<2; n++) ch_out[n] = fir[n].Execute(ch_in[n]);
        codec.Write(ch_out);
    }
}
*/
```

正しいプログラム（その1）

FIRdynamicはデフォルト・コンストラクタをもっているので，コンパイル・エラーは発生しない

左チャネル用

右チャネル用

FIRdynamicではオブジェクトを代入するための演算子＝が定義されていないので，正しい代入が行われる

FIRdynamicBetaがFIRdynamicに変わった以外はリスト5のExample 4と同じ

正しいプログラム（その2）

FIRdynamicはデフォルト・コンストラクタを持っているので，コンパイル・エラーは発生しない

FIRdynamicではオブジェクトを代入するための演算子＝が定義されているので，正しい代入が行われる

FIRdynamicBetaがFIRdynamicに変わった以外はリスト5のExample 5と同じ

る点を改良したクラスを**リスト6**(FIRvariableOrder.hpp)に示す．新たなクラスFIRdynamicでは，コンストラクタをデフォルト・コンストラクタに変更した．さらにFIRdynamicクラスのオブジェクトの代入を，演算子＝を使って正しく行われるようにするために，代入演算子関数operator=()を追加した．

▶ 非公開部のメンバ

非公開部では，クラスFIRdynamicBetaの場合と2か所が異なっている．一つはポインタhmで，ポインタの指す内容はconstであるが，このポインタ自身はconstにはしていない．もう一つは，ポインタunを非constにしたことである．この二つの変更は，operator=()の処理で，これらのポインタを書き換える必要があることに対応するためである．

▶ デフォルト・コンストラクタ

クラスFIRdynamicBetaのコンストラクタは二つの引き数をもつので，新たなクラスFIRdynamicでは，これらの引き数にデフォルト値を割り当てた．つまり，hk[]にはNULLポインタを割り当て，orderには0を割り当てた．この変更により，このコンストラクタはデフォルト・コンストラクタになる．

▶ 代入演算子関数operator=()

ここで考えているFIRdynamicクラスのように，クラス内でメモリを動的に確保するような場合，そのクラスのオブジェクトを正しく代入するためには，演

見本

算子=をオーバロード(多重定義)する必要がある．

この演算子を実現するための代入演算子関数であるoperator=()はメンバ関数とし，その処理はクラスFIRdynamicの外部で定義した．

この場合の代入演算子関数は，operator=をメンバ関数の名前と考え，演算子=の右側に記述された要素を，この代入演算子関数の引き数とみなせばよい．

したがって，たとえば**リスト5**のExample 4の中で使っている，次の代入文，

```
fir[0] = firL;
```

は，演算子operator=()を定義することで，

```
fir[0].operator=(firL);
```

のように解釈されることになる．つまり，operator=という名前のメンバ関数を使って，fir[0]に対して処理を行うが，このときのメンバ関数の引き数がfirLになる．

代入演算子をオーバロードする際には，通常，自己代入[注10]でオブジェクトの内部の重要な情報が破壊されることを防ぐための処理が必要になる．そのため，通常は最初にif文などでチェックする必要がある．しかし，ここでは自己代入が生じても問題が起きないようにしているので，このチェックは行っていない[(1)]．

この代入演算子関数での処理は，最初に_orderの値を変更し，次にunに対応するメモリをnew演算子で確保して，そのポインタを一時的にun_に格納する．メモリの確保に失敗した場合は，関数assert()がメッセージを出し，プログラムを異常終了する．メモリの確保ができたら，演算子delete[]で今までunに対応していたメモリを解放し，一時的に格納してあったポインタをunに設定する．次に，unの指す配列をクリアし，最後にhmを変更する．

この代入演算子関数は戻り値として参照を返すようにしており，return文には戻り値として*this[注11]と書かれているが，これは，

```
a = b = c;
```

のような書き方を許すためである．このような書き方を使わない場合は，次のように変更してもよい．

```
FIRdynamic& operator
                =(const FIRdynamic& src);
```

を，

```
void operator
                =(const FIRdynamic& src);
```

に変更し，さらにoperator=()の実装部分からreturn *thisを削除する．

● 改良版FIRフィルタのクラス(II)の使用例

直接形FIRフィルタのクラスFIRdynamicの使用例(FIR_LPFHPF2.cpp)を**リスト7**に示す．作成するフィルタの特性は，**リスト5**と同じものである．

この**リスト7**には二つのmain()関数があり，一方はコメントになっている．コメントになっていないほう(Example 1)が**リスト5**のExample 4と同じものであり，コメントになっているほう(Example 2)が**リスト5**のExample 5と同じものである．**リスト7**のmain()関数は，どちらも正しいプログラムであり，コンパイル・エラーは発生しない．

注10：a = a;のような代入文．
注11：thisとは，操作対象としているオブジェクト，つまり自分自身を指す，明示されないポインタである．

参考文献

(1) M. クライン，G. ロモウ，M. ギルウ：C++ FAQ 第2版，pp.335-341，ピアソン・エデュケーション，2000年．

見本