4-1 派生と継承

本節では、既存クラスの資産を継承することによって新しいクラスを作り出す派生について学習します。

会員クラスの実現

あるスポーツクラブの会員を表すクラスを考えます。**List 4-1** と **List 4-2** に示すのが、一般会員クラスのヘッダ部とソース部です。

▶ 本来ならば『会員クラス』と呼ぶべきですが、この後で作成する『優待会員クラス』と区別しやすくするために、あえて『一般会員クラス』と呼んでいます。

List 4-1 Member1/Member.h

```
// 一般会員クラス（第１版：ヘッダ部）

#ifndef ___Member
#define ___Member

#include <string>

//===== 一般会員クラス =====//
class Member {
    std::string full_name;  // 氏名
    int     number;         // 会員番号
    double weight;          // 体重

public:
    //--- コンストラクタ ---//
    Member(const std::string& name, int no, double w);

    //--- 氏名取得（full_nameのゲッタ）---//
    std::string name() const { return full_name; }

    //--- 会員番号取得（numberのゲッタ）---//
    int no() const { return number; }

    //--- 体重取得（weightのゲッタ）---//
    double get_weight() const { return weight; }

    //--- 体重設定（weightのセッタ）---//
    void set_weight(double w) { weight = (w > 0) ? w : 0; }
};

#endif
```

List 4-2 Member1/Member.cpp

```
//--- 一般会員クラス（第１版：ソース部）---//

#include <iostream>
#include "Member.h"

using namespace std;

//--- コンストラクタ ---//
Member::Member(const string& name, int no, double w)
                : full_name(name), number(no)
{
    set_weight(w);                   // 体重を設定
}
```

実際には数多くのデータメンバが存在するはずでしょうが、ここに示している一般会員クラス *Member* では、以下の三つだけとしています。

- 氏名 *full_name*
- 会員番号 *number*
- 体重 *weight*

コンストラクタは、仮引数 *name*, *no*, *w* に受け取った三つの値で、対応するデータメンバ *full_name*, *number*, *weight* を初期化します。

コンストラクタの他に、四つのメンバ関数が定義されています。氏名を取得するゲッタ *name*、会員番号を取得するゲッタ *no*、体重を取得・設定するゲッタ *get_weight* とセッタ *set_weight* です。

▶ 体重（データメンバ *weight*）のセッタである *set_weight* は、*weight* が負値になるのを避けるために、仮引数 *w* に負値を受け取ったときは *weight* に *0* を代入しています。

関数本体がソース部で定義されているのは、コンストラクタだけです。それ以外のメンバ関数の本体は、ヘッダ部のクラス定義の中で定義されていますので、内部結合をもつインライン関数となります。

▶ コンストラクタでは、名前 *full_name* と会員番号 *number* の初期化を青網部のコンストラクタ初期化子で行っています。また、体重の設定をメンバ関数 *set_weight* に委ねているため（黒網部）、*weight* が負値になることはありません。

List 4-3 に示すのが、一般会員クラスを利用するプログラム例です。

▶ クラス *Member* 型のオブジェクトを１個生成して、コンストラクタを含む全メンバ関数を適用するだけの単純なプログラムです。

List 4-3 Member1/MemberTest.cpp

```
//--- 一般会員クラス（第１版）の利用例 ---//

#include <iostream>
#include "Member.h"

using namespace std;

int main()
{
    Member kaneko("金子健太", 15, 75.2);

    double weight = kaneko.get_weight();    // 金子君の体重を取得
    kaneko.set_weight(weight - 3.7);        // 金子君の体重を更新（3.7kg減量）

    cout << "No." << kaneko.no() << "：" << kaneko.name()
         << "（" << kaneko.get_weight() << "kg）\n";
}
```

実行結果

```
No.15：金子健太（71.5kg）
```

一般会員である金子健太君を表すのが、クラス *Member* 型のオブジェクト *kaneko* です。金子君の会員番号は 15 で、体重は 75.2kg です。

ただし、金子君は3.7kgの減量を達成したため、メンバ関数 *get_weight* と *set_weight* を利用して体重を更新しています。体重が 71.5kg へと更新されていることは、実行結果からも確認できます。

優待会員クラスの実現

さて、スポーツクラブでは、特典の付いた『優待会員』という制度が導入されました。会員一人一人で内容が異なる特典を ***string*** 型のデータメンバで表すことにして、優待会員クラスを作ることにしましょう。

一般会員クラス *Member* をもとに優待会員クラスを作る作業は、単純です。**ヘッダ部とソース部の各ファイルをコピーして、部分的な追加と変更を施すだけです。**

この手順で作成した優待会員クラスのヘッダ部とソース部を、**List 4-4** と **List 4-5** に示します（試作版であるため、便宜的にクラス名を *VipMember0* としています）。

▶ プログラムの追加箇所が青網部で、変更箇所が黒網部です。

List 4-4 Member1/VipMember0.h

```
// 試作版・優待会員クラス（ヘッダ部）

#ifndef ___VipMember0
#define ___VipMember0

#include <string>

//===== 試作版・優待会員クラス =====//
class VipMember0 {
    std::string full_name;   // 氏名
    int    number;           // 会員番号
    double weight;           // 体重
    std::string privilege;   // 特典

public:
    //--- コンストラクタ ---//
    VipMember0(const std::string& name, int no, double w, const std::string& prv);

    //--- 氏名取得（full_nameのゲッタ）---//
    std::string name() const { return full_name; }

    //--- 会員番号取得（numberのゲッタ）---//
    int no() const { return number; }

    //--- 体重取得（weightのゲッタ）---//
    double get_weight() const { return weight; }

    //--- 体重設定（weightのセッタ）---//
    void set_weight(double w) { weight = (w > 0) ? w : 0; }

    //--- 特典取得（privilegeのゲッタ）---//
    std::string get_privilege() const { return privilege; }

    //--- 特典設定（privilegeのセッタ）---//
    void set_privilege(const std::string& prv) {
        privilege = (prv != "") ? prv : "未登録";
    }
};

#endif
```

特典を表すのが、***string*** 型のデータメンバ *privilege* です。このデータメンバの追加に伴って、特典を取得・設定するためのゲッタ *get_privilege* とセッタ *set_privilege* とが追加されています。

▶ データメンバ *privilege* のセッタであるメンバ関数 *set_privilege* では、仮引数 *prv* に空文字列を受け取った場合は、文字列 "未登録" を *privilege* に代入します。

List 4-5　Member1/VipMember0.cpp

```
// 試作版・優待会員クラス（ソース部）

#include <string>
#include <iostream>
#include "VipMember0.h"

using namespace std;

//--- コンストラクタ ---//
VipMember0::VipMember0(const string& name, int no, double w, const string& prv)
          : full_name(name), number(no)
{
    set_weight(w);                // 体重を設定
    set_privilege(prv);           // 特典を設定
}
```


コンストラクタの仕様も変更されています。特典用の文字列を受け取るための仮引数 *prv* が増えるとともに、その値を設定する処理が追加されています。

▶　データメンバ *privilege* への値の設定は、メンバ関数 *set_privilege* によって行っています。そのため、仮引数 *prv* に空文字列を受け取った場合は、*privilege* に "未登録" が代入されます。

試作版・優待会員クラス *VipMember0* を利用するプログラム例を **List 4-6** に示します。これは、クラス *VipMember0* 型のオブジェクトを 1 個生成して、コンストラクタを含む各メンバ関数を適用するだけの単純なプログラムです。

List 4-6　Member1/VipMember0Test.cpp

```
// 試作版・優待会員クラスの利用例

#include <iostream>
#include "VipMember0.h"

using namespace std;

int main()
{
    VipMember0 mineya("峰屋龍次", 17, 89.2, "会費全額免除");

    double weight = mineya.get_weight();     // 峰屋君の体重を取得
    mineya.set_weight(weight - 15.3);        // 峰屋君の体重を更新（15.3kg減量）

    cout << "No." << mineya.no() << "：" << mineya.name()
         << "（" << mineya.get_weight() << "kg）"
         << "特典＝" << mineya.get_privilege() << '\n';
}
```

実行結果

```
No.17：峰屋龍次（73.9kg）特典＝会費全額免除
```

優待会員の峰屋龍次君を表すのが、クラス *VipMember0* 型のオブジェクト *mineya* です。峰屋君の会員番号は 17 で、体重は 89.2kg であり、特典は "会費全額免除" です。

なお、15.3kg の減量を達成したため、メンバ関数 *get_weight* と *set_weight* を利用して体重を更新しています。体重の更新が正しく行われていることは、実行結果からも確認できます。

それでは、一般会員クラス *Member* と優待会員クラス *VipMember0* の両方を利用するプログラムを作ってみましょう。**List 4-7** に示すのが、その一例です。

List 4-7 Member1/SlimOff0.cpp

```
// 一般会員クラス（第１版）と試作版・優待会員クラスの利用例

#include <iostream>
#include "Member.h"
#include "VipMember0.h"

using namespace std;

//--- 一般会員mの減量（体重がdw減る）---//
void slim_off(Member& m, double dw)
{
    double weight = m.get_weight();      // 現在の体重を取得
    if (weight > dw)
        m.set_weight(weight - dw);       // 体重を更新
}

//--- 優待会員mの減量（体重がdw減る）---//
void slim_off(VipMember0& m, double dw)
{
    double weight = m.get_weight();      // 現在の体重を取得
    if (weight > dw)
        m.set_weight(weight - dw);       // 体重を更新
}

int main()
{
    Member kaneko("金子健太", 15, 75.2);                          // 一般会員
    VipMember0 mineya("峰屋龍次", 17, 89.2, "会費全額免除");      // 優待会員

    slim_off(kaneko, 3.7);                    // 金子君が3.7kg減量
    cout << "No." << kaneko.no() << "：" << kaneko.name()
         << "（" << kaneko.get_weight() << "kg）\n";

    slim_off(mineya, 15.3);                   // 峰屋君が15.3kg減量
    cout << "No." << mineya.no() << "：" << mineya.name()
         << "（" << mineya.get_weight() << "kg）"
         << " 特典＝" << mineya.get_privilege() << '\n';
}
```

実行結果

```
No.15：金子健太（71.5kg）
No.17：峰屋龍次（73.9kg）特典＝会費全額免除
```

二つの *slim_off* は、会員の減量処理を行うための関数です。一般会員クラス用の関数と優待会員クラス用の関数が、個別に作られて多重定義されています。すなわち、**ほとんど同じ処理を行う関数 *slim_off* が、各クラスごとに作られている**わけです。このように実現しなければならないのは、関数の処理対象である変数の『型』が異なるからです。

一般会員クラスと優待会員クラスの“見かけ”は似ていますが、コンパイラにとって、これら二つのクラスは、何の関係もない、まったく別のクラスです。

同一あるいは類似したクラスを、仕様の異なる別のクラスとして実現すると、**プログラムのあちこちに“似て非なる”クラスが溢れかえってしまい**、プログラムの開発効率・保守性が低下します。

派生と継承

このような問題を解決する手段の一つが、クラスの**派生**（*derive*）です。派生とは、既存クラスのデータメンバやメンバ関数などの資産を**継承**（*inheritance*）したクラスを新しく作り出すことです。なお、派生の際は、資産を単純に継承するだけでなく、データメンバやメンバ関数を追加したり、上書き（うわがき）したりできます。

Fig.4-1 を見てください。これは、クラス*Base*と、その資産を継承するクラス*Derived*の定義です。クラス*Derived*の定義中の、クラス名*Derived*の後ろのコロン記号：に続いて書かれている網かけ部の*Base*が、継承元のクラス名です。

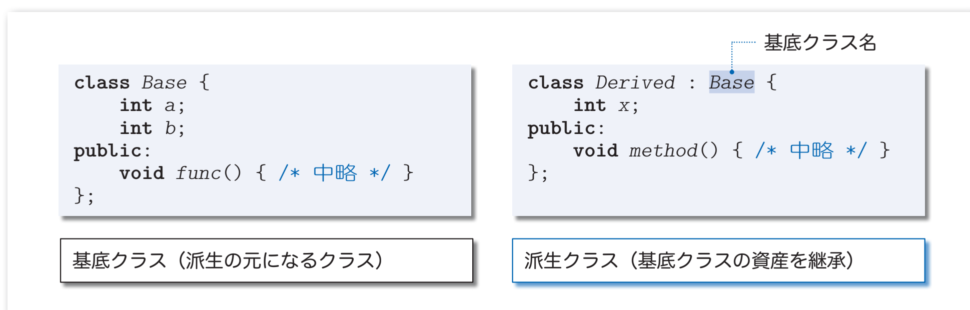

Fig.4-1 基底クラスと派生クラス

これら二つのクラスの関係は、以下のように表現します。

> **クラス*Derived*はクラス*Base*から派生した。**

なお、派生の元になるクラスと、派生によって作られたクラスには、以下に示す呼び名があります。

- 派生元のクラス … 基底クラス／上位クラス／親クラス／スーパークラス
- 派生したクラス … 派生クラス／下位クラス／子クラス／サブクラス

数多くの呼び方が存在するのですが、C++の世界で最もポピュラーなのは、先頭に書かれた**基底クラス**（*base class*）と**派生クラス**（*derived class*）です。

そのため、『クラス*Base*からクラス*Derived*が派生している』ことは、以下のようにも表現できます。

- クラス*Derived*にとって、クラス*Base*は基底クラスである。
- クラス*Base*にとって、クラス*Derived*は派生クラスである。

派生クラスは、基底クラスのデータメンバやメンバ関数などの資産を継承しますので、基底クラスのメンバは、派生クラスの中に“部分”として含まれることになります。

▶ 基底クラスで『型』や『列挙定数』などが定義されていれば、それらも資産として継承されます。

基底クラス *Base* と派生クラス *Derived* がもつ資産の概略を図で示すと、**Fig.4-2** のようになります。

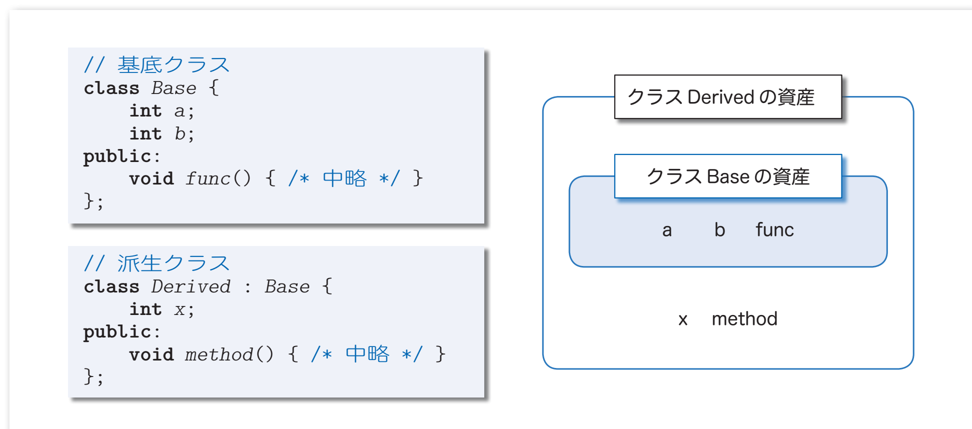

Fig.4-2 基底クラスと派生クラスの資産

この図からも明らかですが、念のために各クラスの資産を確認しましょう。

▪ 基底クラス *Base*

データメンバは *a* と *b* の 2 個で、メンバ関数は *func* の 1 個のみです。

▪ 派生クラス *Derived*

クラス定義の中では、データメンバ *x* とメンバ関数 *method* だけが宣言・定義されています。とはいえ、クラス *Base* のデータメンバとメンバ関数を継承しているのですから、それらを合わせると、データメンバは 3 個、メンバ関数は 2 個となります。

重要 派生クラス（下位クラス）は、基底クラス（上位クラス）の資産を継承するとともに、それを部分として含むクラスである。

▶ コンパイラによって自動的に定義されるデフォルトコンストラクタ・デフォルトデストラクタ・代入演算子も、各クラスの資産として含まれます（この図では省略しています）。
なお、フレンド関係が派生によって継承されることはありません。

Column 4-1 基底クラスと派生クラス

オブジェクト指向プログラミング言語 Java では、派生クラスを**サブクラス**（*sub class*）と呼び、基底クラスを**スーパークラス**（*super class*）と呼ぶのが一般的です。

sub とは《部分》という意味で、super とは《部分を含んだ全体・完全》という意味です。「資産」の量という観点では、基底クラスは派生クラスの《部分》であり、sub や super のニュアンスとは反対です。このような紛らわしさを避けるため、C++ では、サブクラス／スーパークラスと呼ばずに、派生クラス／基底クラスと呼んでいます。

クラス階層図

派生クラスは、基底クラスから生まれた“子供”にたとえられます。この親子関係を表すのが、**Fig.4-3** に示すクラス階層図です。クラス階層図では、派生クラスから基底クラスに向かって矢印を結びます。矢印の向きは、資産の継承とは**逆向き**です。

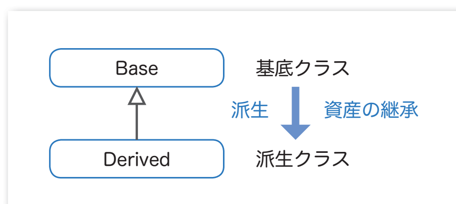

Fig.4-3 クラス階層図と資産の継承

クラス *Derived* の定義での“ : *Base* ”は、

『このクラスの親は *Base* です。』

という表明です。すなわち、親であるクラス *Base* の知らないところで、勝手に子供が作られるのです。

子供（派生クラス）は親（基底クラス）を知っているのですが、親（基底クラス）は子供（派生クラス）のことを知りません。そもそも子供がいるのか、いるのであれば何人いるのか、といった情報を、親がもつことはできません。基底クラス側で『このクラスを私の子供にします。』といった宣言は不可能なのです。

矢印の向きが《派生クラス→基底クラス》であるのは、このような事情によります。

＊

さて、派生は、1 回に制限されるものではありません。そのことを **Fig.4-4** の例で考えましょう。クラス *A* からクラス *B* を派生し、クラス *B* からクラス *C* とクラス *D* を派生しています。クラス *B* はクラス *A* の子であり、クラス *C* とクラス *D* はクラス *B* の子です。したがって、クラス *C* とクラス *D* はクラス *A* の孫となります。

直接の親となるクラスのことを**直接基底クラス**（*direct base class*）と呼び、直接の親ではないものの先祖に当たるクラスのことを**間接基底クラス**（*indirect base class*）と呼びます。

たとえば、クラス *D* にとって、クラス *B* は直接基底クラスであり、クラス *A* は間接基底クラスということになります。

さらに、このことは、『クラス *D* はクラス *A* から**間接派生**している』、あるいは、『クラス *B* から**直接派生**している』と表現されます。

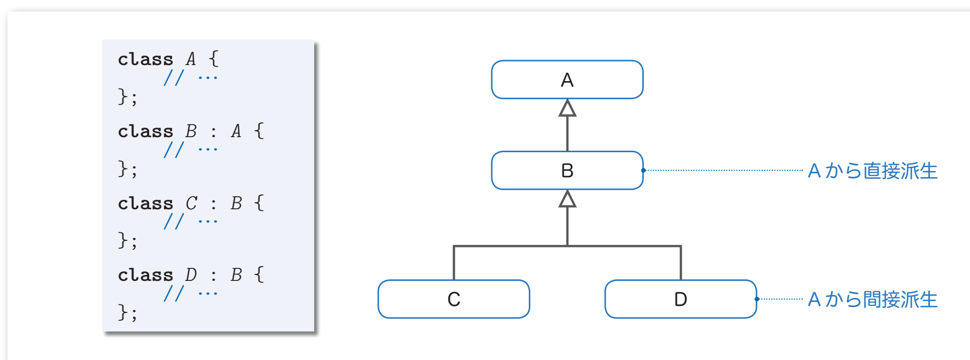

Fig.4-4 クラス階層図の一例

派生の形態

外部に公開すべきデータや手続きのみを公開し、そうでないものは非公開とするのが、クラスを設計する際の大原則であることは、既に学習しました。**派生クラスは基底クラスの資産を継承するものの、それらをクラスの外部に公開するかどうかは別問題です。**

基底クラスと派生クラス内のメンバのアクセス性の関係は、以下に示す3種類の派生の形態に依存します。

- **private** 派生
- **protected** 派生
- **public** 派生

どの形態で派生を行うのかの指定は、派生クラスの定義における基底クラス名の前に **private**, **protected**, **public** のいずれかのアクセス指定子を置くことによって行います。

たとえば、以下に示すクラス *Derived* は、クラス *Base* からの **public** 派生です。

```
class Derived : public Base { /*--- 中略 ---*/ };   // Baseからpublic派生
```

なお、アクセス指定子を省略した場合は、自動的に“**private** 派生”となります。

▶ ただし、派生クラスの定義に用いるキーワードとして **class** ではなく **struct** を用いて定義した場合は、“**public** 派生”となります。

ここでは、**List 4-8** に示すクラス *Super* からの派生を例に、三つの派生形態について学習していくことにします。

List 4-8　chap04/Super.h

```
// 基底クラスSuper
#ifndef ___Super
#define ___Super

class Super {
private:
    int pri;     // 非公開
protected:
    int pro;     // 限定公開
public:
    int pub;     // 公開
};

#endif
```

private 派生

クラス *Super* から **private** 派生を行う例を示したのが、**List 4-9** です。

▶ コンパイルエラーとなる行は、// によってコメントアウトしています。なお、プログラムを実行しても何も表示されません。

private 派生を行うことによって、派生クラス *Sub* にとってのクラス *Super* は、**非公開基底クラス**（*private base class*）となります。この派生における、基底クラスと派生クラスのメンバのアクセス性を示したのが **Fig.4-5** です。

まず、派生クラスの内部（派生クラスのメンバ関数とフレンド関数）から、基底クラスの非公開メンバⒶが**アクセス不可能である**ことが分かります。そうなる理由は単純です。基底クラスで『外部に対して公開しない』と宣言しているからです。

▶ もしも、派生クラス *Derived* のメンバ関数やフレンド関数から、基底クラス *Base* の非公開メンバ *pri* を自由にアクセスできるとしたらどうなるでしょう。クラスの派生を行うだけで、基底クラスの非公開部を扱えることになってしまいます。これでは情報隠蔽どころではありません。

　なお、アクセスできないとはいえ、基底クラスから継承したメンバが消滅したわけではありません（派生クラスの中に部分として存在しているのにアクセスできないというだけです）。

List 4-9 chap04/Private.cpp

```
// private派生とメンバのアクセス性（注：エラーとなる行はコメントアウト）
#include "Super.h"

class Sub : private Super {
    void f() {
//      pri = 1;    // クラス内部でもアクセスできない      ←1
        pro = 1;
        pub = 1;
    }
};

int main()
{
    Sub x;

//  x.pri = 1;      // クラス外部からアクセスできない
//  x.pro = 1;      // クラス外部からアクセスできない      ←2
//  x.pub = 1;      // クラス外部からアクセスできない
}
```

また、基底クラスの限定公開メンバⒷと公開メンバⒸが、派生クラス内で**非公開メンバ**として扱われ、派生クラスの利用者に対して公開されないことが図から分かります。

網かけ部がコンパイルエラーとなる理由を確認しましょう。

1 派生クラス *Sub* の内部（メンバ関数とフレンド関数：この場合はメンバ関数 *f* の本体）において、基底クラス *Super* の非公開メンバ *pri* のアクセスはできない。

2 基底クラス *Super* の全メンバは、派生クラス *Sub* の利用者に対して非公開である。

なお、プログラムと図は、以下のことを示しています。

重要 限定公開メンバは、外部に対しては存在を隠すが、自分の子供である派生クラスに対しては存在を隠さない。

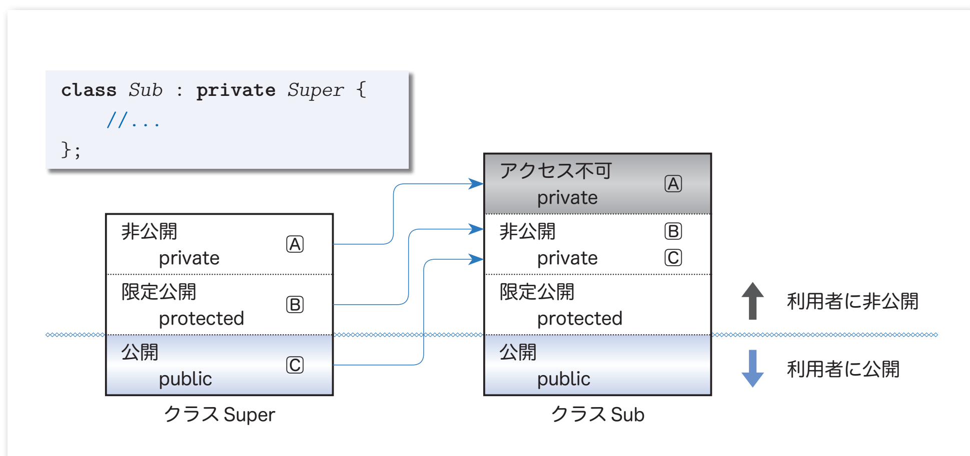

Fig.4-5 private 派生とメンバのアクセス性

protected 派生

クラス *Super* から **protected** 派生を行う例を示したのが、**List 4-10** です。

この **protected** 派生では、派生クラス *Sub* にとっての基底クラス *Super* は、**限定公開基底クラス**（*protected base class*）となります。

List 4-10 chap04/Protected.cpp

```
// protected派生とメンバのアクセス性（注：エラーとなる行はコメントアウト）
#include "Super.h"

class Sub : protected Super {
    void f() {
//      pri = 1;    // クラス内部でもアクセスできない        ―1
        pro = 1;
        pub = 1;
    }
};

int main()
{
    Sub x;

//  x.pri = 1;      // クラス外部からアクセスできない
//  x.pro = 1;      // クラス外部からアクセスできない        ―2
//  x.pub = 1;      // クラス外部からアクセスできない
}
```

protected 派生におけるメンバのアクセス性を示したのが **Fig.4-6** です。

派生クラスのメンバ関数とフレンド関数から基底クラスの非公開メンバⒶをアクセスできない点は、**private** 派生と同様です。

ただし、基底クラスの限定公開メンバⒷと公開メンバⒸとが、派生クラス中で**限定公開メンバ**として扱われる点が、**public** 派生と異なります（当然、これらのメンバは、派生クラスの利用者である外部に対しては公開されません）。

▶ 限定公開メンバは、*Sub* から派生したクラス（*Super* の孫に相当するクラス）の内部ではアクセスできますが、その外部からはアクセスできなくなります。

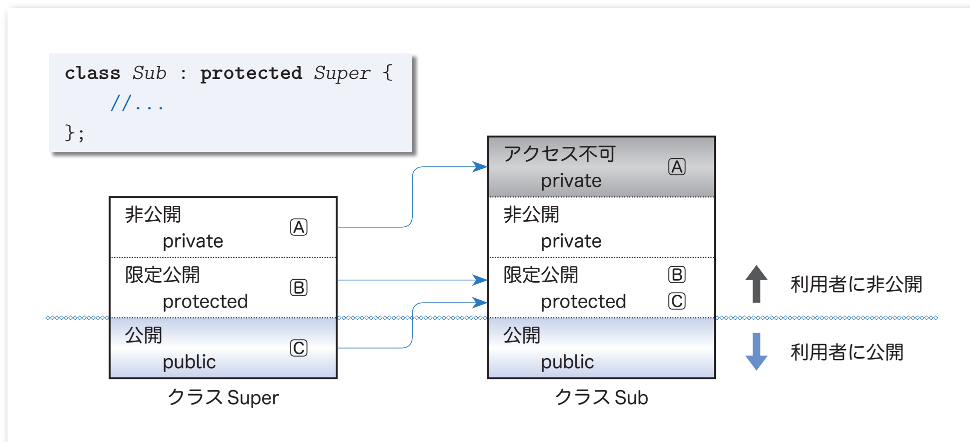

Fig.4-6 protected 派生とメンバのアクセス性

public 派生

クラス *Super* から `public` 派生を行う例を示したのが、**List 4-11** です。

この `public` 派生では、派生クラス *Sub* にとっての基底クラス *Super* は、**公開基底クラス**（*public base class*）となります。

List 4-11 chap04/Public.cpp

```
// public派生とメンバのアクセス性（注：エラーとなる行はコメントアウト）
#include "Super.h"

class Sub : public Super {
    void f() {
//      pri = 1;     // クラス内部からのアクセス不可        ❶
        pro = 1;
        pub = 1;
    }
};

int main()
{
    Sub x;

//  x.pri = 1;      // クラス外部からのアクセス不可        ❷
//  x.pro = 1;      // クラス外部からのアクセス不可
    x.pub = 1;      // 公開属性が維持される               ❸
}
```

`public` 派生におけるメンバのアクセス性を示したのが **Fig.4-7** です。

他の派生と同様に、派生クラスのメンバ関数とフレンド関数から、基底クラスの非公開メンバⒶをアクセスすることは不可能です。

基底クラスの限定公開メンバⒷは、派生クラス中においても**限定公開メンバとしての扱い**を受け、基底クラスの公開メンバⒸは、派生クラスでも**公開メンバ**として扱われますから、派生クラスの利用者に公開されます（❸）。すなわち、**基底クラスの非公開以外のメンバ（限定公開メンバと公開メンバ）のアクセス性が、派生クラスで維持されます。**

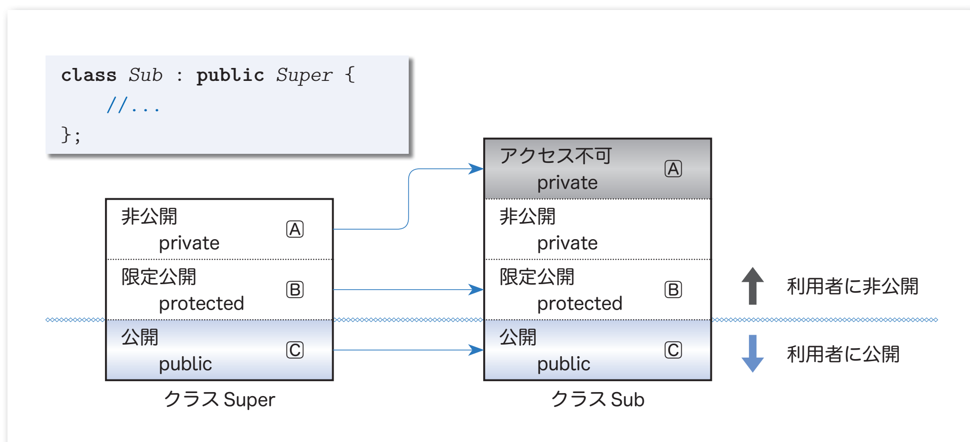

Fig.4-7 public 派生とメンバのアクセス性

三つの派生のまとめ

3 種類の派生（**private** 派生、**protected** 派生、**public** 派生）に共通する原理や規則性をまとめると、以下のようになります。

- **どの派生を行っても、基底クラスの非公開メンバを派生クラスからアクセスすることはできない。**
- **“○○派生”を行うと、基底クラスの公開メンバが、派生クラスの“○○部”に所属することになる。**
- **限定公開メンバは、外部には公開されないが、自分の子供（直接派生するクラス）には公開される。**

基底クラス部分オブジェクトとコンストラクタ初期化子

Fig.4-1（p.145）で考えた派生を、実際のプログラムとして実現してみましょう。そのプログラムを **List 4-12** に示します。なお、ここでの派生は **public** 派生とし、さらに、両クラスに対して、コンストラクタを追加しています。

コンストラクタ初期化子

クラスの派生では、基底クラスの資産が継承されるのが原則ですが、コンストラクタとデストラクタが例外であることは、必ず覚えておく必要があります。

重要 クラスの派生において、コンストラクタとデストラクタは継承されない。

もっとも、派生クラスにおいて、コンストラクタをゼロから作り直す必要があるかというと、そうではありません。

Column 4-2 基底クラス非公開部のアクセス

派生クラスを基底クラスのフレンドであると明示的に宣言すれば、基底クラスの非公開メンバにもアクセスできるようになります。

```
class Super {
    friend class Sub;    // Sub君は、僕のお友達ですよ!!
    // …
};
class Sub : Super {
    // クラスSuperの非公開部を自由にアクセスできる
};
```

クラス *Sub* は、クラス *Super* から派生したクラスでもあり、フレンドクラスでもあるということになります。

※ ただし、よほど特別な事情でもない限り、このような宣言を行うべきではありません。

List 4-12　chap04/BaseDerived.h

```
// 基底クラスと派生クラス

#ifndef ___Member
#define ___Member

#include <iostream>

//--- 基底クラス ---//
class Base {
    int a;
    int b;

public:
    Base(int aa, int bb) : a(aa), b(bb) { }

    void func() const {
        std::cout << "a = " << a << '\n';
        std::cout << "b = " << b << '\n';
    }
};

//--- 派生クラス ---//
class Derived : public Base {
    int x;

public:
    Derived(int aa, int bb, int xx) : Base(aa, bb), x(xx) { }

    void method() const {
        func();
        std::cout << "x = " << x << '\n';
    }
};

#endif
```

直接基底クラスのコンストラクタを呼び出す

クラス *Derived* のコンストラクタに注目しましょう。点線部のコンストラクタ初期化子中のメンバ初期化子 x(xx) の働きによって、メンバ x が xx の値で初期化されます。

もう一つのメンバ初期化子である青網部は、以下に示す形式です。

基底クラス名 (仮引数宣言節)

これは、**直接基底クラスのコンストラクタの呼出し**の指示であり、**コンストラクタ初期化子**（*constructor initializer*）と呼ばれます。本コンストラクタ初期化子によって、親クラス *Base* のコンストラクタに、データメンバ a と b の初期化を委ねているのです。

重要　基底クラスから継承したデータメンバの初期化は、コンストラクタ初期化子によって、直接基底クラスのコンストラクタに委ねるのが原則である。

▶ クラス *Derived* のコンストラクタを以下のように定義することはできません。

```
Derived(int aa, int bb, int xx) : a(aa), b(bb), x(xx) { }    // エラー
```

その理由は単純です。基底クラス *Base* で非公開であるデータメンバ a と b に対するアクセスが、メンバクラス *Derived* 内では許可されないからです。

また、メンバ初期化子で初期化できるのは、直接基底クラスのみです。間接基底クラス（親クラスでなく、お爺さん以上のクラス）の初期化を指定することはできません。

基底クラス部分オブジェクト

前ページで定義した派生クラス*Derived*を実際に利用するプログラムを作りましょう。**List 4-13**に示すのが、その一例です。

List 4-13 chap04/BaseDerivedTest1.cpp

```
// 派生の一例

#include <iostream>
#include "BaseDerived.h"

using namespace std;

int main()
{
    Derived dv(1, 2, 3);

    cout << "dv.func()\n";    dv.func();    // Baseから継承したメンバ関数
    cout << "dv.method()\n";  dv.method();  // Derivedに所属するメンバ関数
}
```

実行結果

```
dv.func()
a = 1
b = 2
dv.method()
a = 1
b = 2
x = 3
```

main関数では、クラス*Derived*型のオブジェクト*dv*が定義されています。このオブジェクトには、クラス*Base*から継承したデータメンバa，bと、クラス*Derived*で追加されたデータメンバxが存在しており、それぞれ1，2，3で初期化されます。

その初期化の様子を示したのが、**Fig.4-8**です。クラス*Derived*のコンストラクタが、データメンバaとbの初期化をクラス*Base*のコンストラクタに委ねることは、前ページで学習したとおりです。

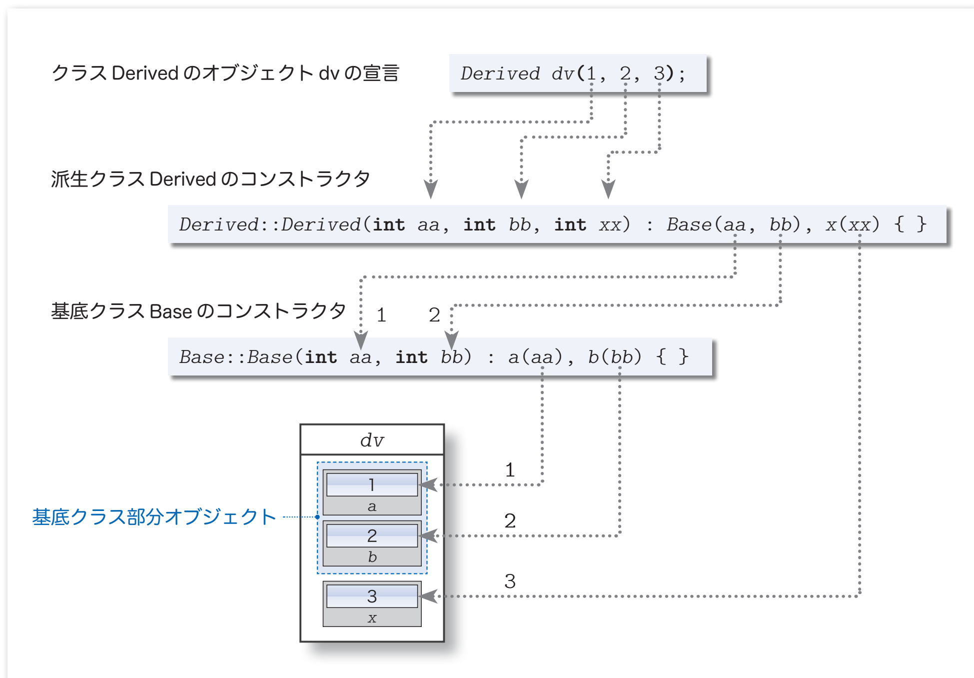

Fig.4-8 コンストラクタの働きと基底クラス部分オブジェクト

なお、図に示すように、派生クラス型のオブジェクト中に含まれている基底クラス型のオブジェクト（データメンバaとbとから構成されている部分）は、**基底クラス部分オブジェクト**（*base class sub-object*）と呼ばれます。

重要 派生クラス型のオブジェクトの中に含まれている、基底クラス型のオブジェクトは、基底クラス部分オブジェクトと呼ばれる。

さて、main関数では、オブジェクトdvに対して、二つのメンバ関数funcとmethodを呼び出しています。

メンバ関数funcは、基底クラスBase内で右のように定義されています。

```
void Base::func() const {
    std::cout << "a = " << a << '\n';
    std::cout << "b = " << b << '\n';
}
```

Derived型のdvに対するfuncの呼出しであるdv.func()によって、データメンバaとbの値を正しく表示できることは、以下のことを示しています。

- **クラスの派生がpublic派生であるため、基底クラスBaseの公開メンバ関数funcが、派生クラスDerivedにおいても公開メンバとして継承されている。**
- **基底クラスBaseの非公開メンバa，bは、クラスDerivedからはアクセスできないとはいえ、オブジェクトdvの内部に基底クラス部分オブジェクトとして（ちゃんと）含まれている。**

もう一つのメンバ関数methodは、派生クラスDerived内で右のように定義されています。

```
void Derived::method() const {
    func();
    std::cout << "x = " << x << '\n';
}
```

この関数の黒網部は、クラスBaseのメンバ関数funcを呼び出す関数呼出しです。この呼出しによって表示されるのが、左ページ実行結果内の黒網部です（すなわち、基底クラス部分オブジェクト内のデータメンバaとbの値を表示します）。

この実行結果は、以下のことを示しています。

- **クラスBaseから継承したメンバ関数funcは、クラスDerivedの中では、あたかもクラスDerivedのメンバ関数であるかのように呼び出せる。**

ここで、funcを呼び出す式の形式が“関数名()”であることに注意しましょう。もし他のクラスのメンバ関数であれば、呼出しの形式は“オブジェクト名.関数名()”となるはずです。

重要 基底クラスから継承したメンバ関数は、派生クラス型のメンバ関数およびフレンドの中では、あたかも派生クラス型のメンバ関数であるかのように利用できる。

スライシング

引き続き、クラス *Base* と *Derived* を利用するプログラムを考えていきます。**List 4-14** のプログラムを見てください。

List 4-14 chap04/BaseDerivedTest2.cpp

```
// スライシング

#include <iostream>
#include "BaseDerived.h"

using namespace std;

int main()
{
    Derived dv(1, 2, 3);
    Base bs(99, 99);

    cout << "bsの初期状態\n";      bs.func();

    bs = dv;                          // OK：スライシング   ←1
    cout << "dvを代入した後\n";    bs.func();

//  dv = bs;                          // コンパイルエラー   ←2
}
```

実行結果

```
bsの初期状態
a = 99
b = 99
dvを代入した後
a = 1
b = 2
```

main 関数では、前ページのプログラムと同様、クラス *Derived* 型のオブジェクト *dv* が定義されており、データメンバ a，b，x が 1，2，3 で初期化されます。

一方、クラス *Base* 型のオブジェクト *bs* は、データメンバ a と b の両方が 99 となるように初期化されています。その *bs* に対してメンバ関数 *func* 関数を呼び出すと、二つのデータメンバ a と b の値が 99 と表示されます。

*

さて、1の代入をよく見てください。左側の *bs* は基底クラス型で、右側の *dv* は派生クラス型であり、左右のオペランドの型が異なります。このときの代入の様子を示したのが、**Fig.4-9** 1です。派生クラス型オブジェクトである *dv* 内の基底クラス部分オブジェクトのデータメンバ a と b の値が、1 と 2 に更新されます。

この代入を一般的に表現すると、次のようになります。

> **重要** 基底クラスオブジェクトに対して、派生クラスオブジェクトを代入すると、派生クラス内の“基底クラス部分オブジェクト”の部分がコピーされる。

この代入では、オブジェクト *dv* 内の基底クラス部分オブジェクトが代入され、そうでない部分、すなわち、クラス *Derived* に特有の情報（この場合は、データメンバ x の値）は切り捨てられて失われます。このように、代入において一部の情報が欠けてしまうことを、**スライシング**（*slicing*）といいます。

▶ スライシングは、チーズなどの食材を薄切りにすることなどを表す動詞 slice に由来します。

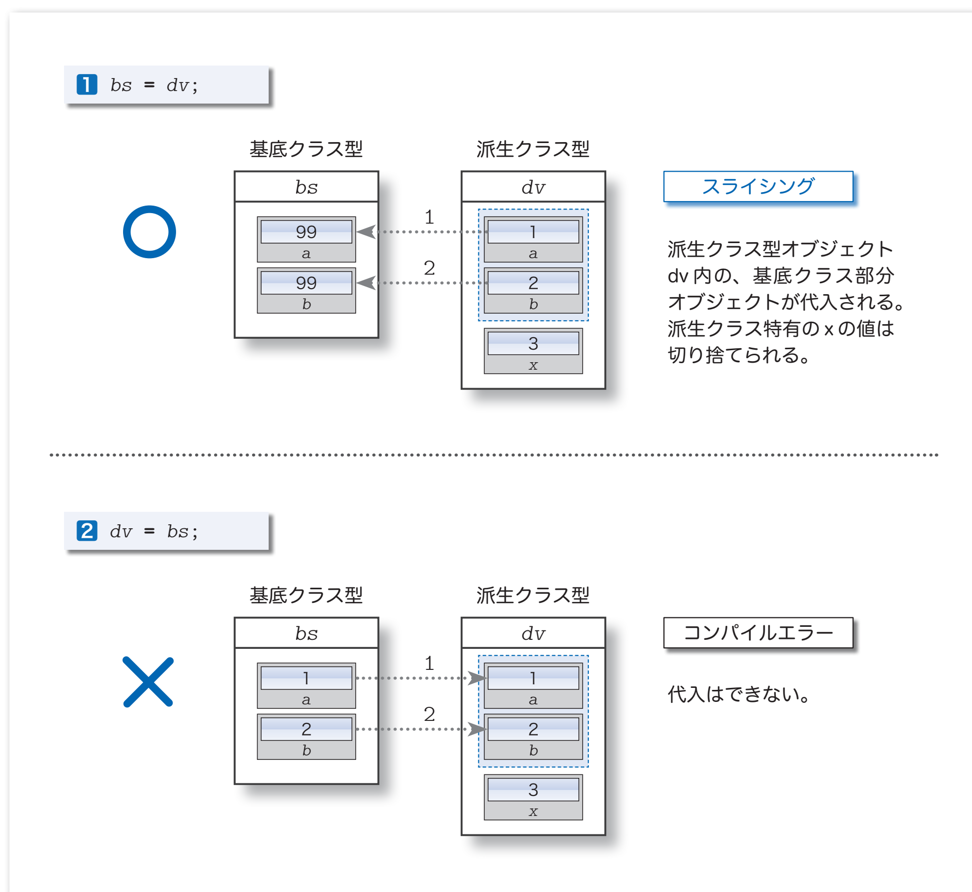

Fig.4-9 基底クラスと派生クラスのオブジェクト間の代入

なお、逆向きの代入、すなわち、派生クラス型オブジェクトへの基底クラス型オブジェクトの代入は不可能です。そのため、**2**はコンパイルエラーとなります。

▶ ここに示すプログラムは、コンパイルエラーの発生を抑止するために、// を付けてコメントアウトしています。

重要 派生クラスオブジェクトに対して、基底クラスオブジェクトを代入することはできない。

このような規則となっている理由は単純です。もし、このような代入が指示されても、データメンバ *x* に代入すべき値が決定不能だからです。

▶ まさか、データメンバに不定値を代入するわけにはいかないですね。

継承とデフォルトコンストラクタ

基底クラスのコンストラクタを派生クラスが継承しないことを p.152 で学習しました。とはいえ、**コンストラクタは間接的な形で継承されている**のです。

このことを、**List 4-15** に示すプログラムで検討していきましょう。

List 4-15 chap04/constructor.cpp

```
// 基底クラスと派生クラスのコンストラクタ

#include <iostream>

using namespace std;

//===== 基底クラス =====//
class Base {
    int x;

public:
    //--- コンストラクタ ---//
    Base() : x(99) { cout << "Base::xを99で初期化。\n"; }

    //--- xのゲッタ ---//
    int get_x() const { return x; }
};

//===== 派生クラス =====//
class Derived : public Base {
    // コンストラクタを含め何も定義しない
};

int main()
{
    Derived d;

    cout << "d.get_x() = " << d.get_x() << '\n';
}
```

実行結果

```
Base::xを99で初期化。
d.get_x() = 99
```

クラス *Base* がもつ唯一のデータメンバが、**int** 型の *x* です。コンストラクタは、その *x* を 99 で初期化します。メンバ関数 *get_x* は、データメンバ *x* のゲッタです。

クラス *Derived* は、クラス *Base* の派生クラスです。コンストラクタが 1 個も定義されていないため、下記のデフォルトコンストラクタ**A**が自動的に定義されるはずです。

```
A Derived::Derived() { }        // コンパイラによって定義されるコンストラクタ？
```

▶ コンストラクタを定義しないクラスに対しては、本体が空であるコンストラクタが、コンパイラによって自動的に定義されることは、『入門編』で学習しました。

main 関数では、クラス *Derived* 型のオブジェクト *d* を宣言しています。上記のコンストラクタが呼び出されるのであれば、データメンバ *x* の初期値は不定値になるはずです（コンストラクタ本体が空であるため）。ところが、オブジェクト *d* 内に基底クラス部分オブジェクトとして含まれるデータメンバ *x* は 99 で初期化されます。

▶ 実行結果（メンバ関数 *d.get_x* の呼出しによって 99 が返却されること）で確認できます。

実は、コンパイラによって自動的に定義されるデフォルトコンストラクタは、（概念上は）以下のようになっているのです。

```
B Derived::Derived() : Base() { }  // コンパイラによって定義されるコンストラクタ
```

網かけ部は、直接基底クラスであるクラス *Base* のデフォルトコンストラクタの呼出しです。*Base* のコンストラクタが自動的に呼び出されるため、メンバ *x* は 99 で初期化されるのです。

直接基底クラスのコンストラクタがそのまま継承されることはないものの、基底クラスのデフォルトコンストラクタが暗黙裏に呼び出されることが分かりました。

重要 コンパイラによって定義されるデフォルトコンストラクタは、直接基底クラスのデフォルトコンストラクタを呼び出す。

クラス *Base* のコンストラクタを以下の定義に置きかえてみましょう。そうすると、クラス *Base* ではなく、クラス *Derived* がコンパイルできなくなります。

```
1 Base::Base(int xx) : x(xx) { cout << "Base::xを" << x << "で初期化。\n"; }
```

クラス *Derived* のデフォルトコンストラクタBの網かけ部の、『クラス *Base* のデフォルトコンストラクタを呼び出す処理』が不可能になるからです。

＊

さて、デフォルトコンストラクタとは、“引数を受け取らないコンストラクタ”ではなく、“引数を与えることなく呼び出せるコンストラクタ”でしたね。そのため、クラス *Base* のコンストラクタを以下のように定義すれば、コンパイルエラーを回避できます。

```
2 Base::Base(int xx = 99) : x(xx) { cout << "Base::xを" << x << "で初期化。\n"; }
```

ここで検討した内容から、以下に示す注意点が得られます。

重要 コンストラクタを定義しないクラスは、そのクラスの直接基底クラスが“引数を与えることなく呼び出せるデフォルトコンストラクタ”をもっていなければ、コンパイルエラーとなる。

▶ デフォルト（default）には、『既定（値）』、『省略時（指定がない場合）に採用される選択（値）』、などの意味があります。

演習 4–1

List 4-15 のクラス *Base* のコンストラクタを上記の2に置きかえたプログラムを作成して、本ページで解説されている内容を確認せよ。

派生クラスオブジェクトの初期化

コンストラクタの実行に際して、コンストラクタの本体とコンストラクタ初期化子は、どのような順序で起動されてオブジェクトの初期化が行われるのでしょうか。

List 4-16 のプログラムで検証しましょう。

List 4-16 chap04/initialize.cpp

```
// 基底クラスとメンバの初期化を確認するクラス群
#include <iostream>

using namespace std;

//===== クラスDerivedの基底クラス =====//
class Base {
    int x;
public:
    Base(int a = 0) : x(a) { cout << "Base::xを" << x << "で初期化。\n"; }
};

//===== クラスDerivedにメンバとして含まれるクラス =====//
class Memb {
    int x;
public:
    Memb(int a = 0) : x(a) { cout << "Memb::xを" << x << "で初期化。\n"; }
};

//===== クラスDerivedはクラスBaseからpublic派生 =====//
class Derived : public Base {
    int y;
    Memb m1;
    Memb m2;
    void say() { y = 0; cout << "Derived::yを" << y << "で初期化。\n"; }
public:
    Derived()                                          { say(); }
    Derived(int a, int b, int c) : m2(a), m1(b), Base(c) { say(); }
};

int main()
{
    Derived d1;

    cout << '\n';

    Derived d2(1, 2, 3);
}
```

実行結果

```
Base::xを0で初期化。
Memb::xを0で初期化。
Memb::xを0で初期化。
Derived::yを0で初期化。

Base::xを3で初期化。
Memb::xを2で初期化。
Memb::xを1で初期化。
Derived::yを0で初期化。
```

本プログラムには単純な構造のクラスが三つあります。

クラス *Base* とクラス *Memb* は、実質的に同じ構造のクラスであり、**int** 型のデータメンバ *x* とコンストラクタのみをもちます。両クラスのコンストラクタが行うのは、仮引数 *a* に受け取った値でデータメンバ *x* を初期化して、その旨を表示することです。

クラス *Base* から **public** 派生するクラス *Derived* には、**int** 型のデータメンバ *y* に加えて、クラス *Memb* 型のデータメンバ *m1* と *m2* とがあります。

多重定義されている二つのコンストラクタは、いずれも、メンバ関数 *say* を呼び出して、データメンバ *y* を 0 で初期化する旨のメッセージを表示します。

main 関数では、クラス *Derived* のオブジェクト *d1* と *d2* が定義されています。各コンストラクタによる初期化の過程を詳しく見ていきましょう。

■ *Derived::Derived()* による *d1* の初期化

基底クラス *Base* のコンストラクタと二つの *Memb* 型のメンバのコンストラクタが自動的に呼び出されていることが、実行結果から分かります。

もちろん、呼び出されているのはデフォルトコンストラクタです（各クラスのコンストラクタのデフォルト実引数の値 0 がコンストラクタに渡されています）。

▶ 既に学習したように、もしクラス *Base* と *Memb* にデフォルトコンストラクタがなければ、このコンストラクタはコンパイルエラーとなります。

■ *Derived::Derived*(**int** *a*, **int** *b*, **int** *c*) による *d2* の初期化

コンストラクタ初期化子 “*m2*(*a*), *m1*(*b*), *Base*(*c*)” によって、メンバ *m2*, *m1* のコンストラクタと基底クラス *Base* のコンストラクタが呼び出されて初期化が行われます。

ただし、初期化の順序は、コンストラクタ初期化子の並び *m2*, *m1*, *Base* とは一致しません。実行結果は、以下のことを示しています。

- 基底クラスの初期化よりも先にメンバの初期化を指定しているにもかかわらず、基底クラスのほうが先に初期化されている。
- 値 1 を指定された *m2* より、値 2 を指定された *m1* のほうが先に初期化されている。

コンストラクタの初期化作業は、次に示す順序で行われます。必ず覚えましょう。

1 基底クラスのコンストラクタによって基底クラス部分オブジェクトが初期化される。
2 宣言された順にデータメンバが初期化される。
3 コンストラクタの本体が実行される。

▶ 2の “宣言された順” は、コンストラクタ初期化子の並びの順ではなく、クラス定義におけるデータメンバ自体の宣言の順序です。このことは、『入門編』の p.411 で学習済みです。

重要 コンストラクタに指定するメンバ初期化子の並びの順序は、先頭を基底クラスコンストラクタ初期化子とし、その後に、データメンバの宣言と同じ順序でデータメンバ初期化子を並べたものとすると、紛らわしさがなくなる。

なお、**デストラクタにおける後始末では、この逆の順序でオブジェクトの解体が行われる**ことになっています。

演習 4–2

デストラクタの起動の順序を確認できるように、**List 4-16** のプログラムを書きかえたプログラムを作成せよ。

コピーコンストラクタとデストラクタと代入演算子

クラスの派生において、デフォルトコンストラクタが間接的な形で継承されることは既に学習しました。それでは、それ以外の特殊なメンバ関数である、コピーコンストラクタ、デストラクタ、代入演算子はどうでしょうか。

これらのことを、**List 4-17** のプログラムで確認していくことにします。

List 4-17 chap04/Array.cpp

```
// 基底クラスと派生クラスの代入演算子とデストラクタ

#include <iostream>

using namespace std;

//===== 超簡易配列クラス =====//
class Array {
    static const int num = 5;        // 要素数（固定）
    int *p;

public:
    //--- デフォルトコンストラクタ ---//
    Array() : p(new int[num]) { cout << "領域確保\n"; }

    //--- コピーコンストラクタ ---//
    Array(const Array& x) : p(new int[x.num]) {
        for (int i = 0; i < num; i++) p[i] = x.p[i];     // xの全要素をコピー
        cout << "コピー初期化\n";
    }

    //--- デストラクタ ---//
    ~Array() { delete[] p;  cout << "領域解放\n"; }

    //--- 代入演算子 ---//
    Array& operator=(const Array& x) {
        for (int i = 0; i < num; i++) p[i] = x.p[i];
        return *this;
    }

    //--- 全要素に値vを代入 ---//
    void set(int v) { for (int i = 0; i < num; i++) p[i] = v; }

    //--- 全要素の値を表示 ---//
    void print() const { for (int i = 0; i < num; i++) cout << p[i] << ' '; }
};

//===== 超簡易配列クラス（派生クラス）=====//
class ArrayX : public Array {
    // コンストラクタを含め何も定義しない
};

int main()
{
    ArrayX a1;
    a1.set(15);      // a1の全要素に15を代入

    ArrayX a2(a1);   // a1で初期化

    ArrayX a3;
    a3 = a1;         // a1の全要素をa3にコピー

    cout << "配列a1 : ";   a1.print();   cout << '\n';
    cout << "配列a2 : ";   a2.print();   cout << '\n';
    cout << "配列a3 : ";   a3.print();   cout << '\n';
}
```

実行結果

```
領域確保
コピー初期化
領域確保
配列a1 : 15 15 15 15 15
配列a2 : 15 15 15 15 15
配列a3 : 15 15 15 15 15
領域解放
領域解放
領域解放
```

クラス *Array* は、1-3 節で学習した配列クラス *IntArray* を簡易化したものです。配列の要素数は 5 に固定されます。

▶ 定義されているメンバ関数の働きは、以下のとおりです。
- デフォルトコンストラクタは、配列用の領域を確保します。
- コピーコンストラクタは、配列用の領域を確保して、仮引数 *x* に受け取ったコピー元配列の全要素をコピーします。
- デストラクタは、配列用の領域を解放します。
- 代入演算子は、*x* の全要素を自分自身の配列にコピーします。
- メンバ関数 *set* は、全要素に同一値 *v* を代入します。
- メンバ関数 *print* は、全要素の値を表示します。

派生の働きを確認するための実験用クラスである *ArrayX* は、クラス *Array* から **public** 派生したクラスです。このクラスの中では、データメンバやメンバ関数は一切定義されていません。

main 関数では、*ArrayX* 型のオブジェクト *a1*, *a2*, *a3* を定義しています。デフォルトコンストラクタ、コピーコンストラクタ、デストラクタ、代入演算子のいずれもが、期待どおりの働き、すなわち基底クラス *Array* と同じ働きをしていることが、実験結果から確認できます。

重要 派生クラス内で、特殊メンバ関数（コピーコンストラクタ、デストラクタ、代入演算子）が定義されなければ、基底クラスのものと実質的に同一の働きをする関数が自動的に定義される。

なお、コンパイラによって自動的に定義される特殊メンバ関数は、いずれも **inline** かつ **public** です。代入演算子についていえば、具体的には、以下のようになります。

重要 派生クラス *X* 内で代入演算子が定義されなければ、以下の形式の代入演算子が自動的に定義される。

```
X& X::operator=(const X&);
```

▶ 自動的に定義される代入演算子の形式が上記のようになるのは、以下の二つの条件のいずれもが満たされる場合です（基底クラスの名前が *B* であるとします）。

- 直接基底クラスが、**const** *B*&, **const volatile** *B*& あるいは *B* を仮引数の型とするコピー代入演算子をもつ。
- そのクラスのすべてのクラス型 *M*（あるいはその配列型）の非静的データメンバについて、それらのクラス型が、**const** *M*&, **const volatile** *M*& あるいは *M* を仮引数の型とするコピー代入演算子をもつ。

そうでない場合、暗黙に定義される代入演算子は、次の形式となります。

```
X& X::operator=(X&);
```

継承と差分プログラミング

派生に関する基本的な学習が一通り終了しました。優待会員クラス *VipMember* を、一般会員クラス *Member* からの派生を行うように変更しましょう。そのように実現した優待会員クラスを示します。ヘッダ部が **List 4-18** で、ソース部が **List 4-19** です。

▶ **List 4-4**（p.142）と **List 4-5**（p.143）は、試作版であったことから、クラス名を便宜的に *VipMember0* としていました。今回は、クラス名を *VipMember* にしています。

List 4-18 Member1/VipMember.h

```
// 優待会員クラス（第１版：ヘッダ部）

#ifndef ___VipMember
#define ___VipMember

#include <string>
#include "Member.h"   ← 基底クラスの定義を取り込む

//===== 優待会員クラス（第１版：ヘッダ部）=====//
class VipMember : public Member {
    std::string privilege;   // 特典

public:
    //--- コンストラクタ ---//
    VipMember(const std::string& name, int no, double w, const std::string& prv);

    //--- 特典取得（privilegeのゲッタ）---//
    std::string get_privilege() const { return privilege; }

    //--- 特典設定（privilegeのセッタ）---//
    void set_privilege(const std::string& prv) {
        privilege = (prv != "") ? prv : "未登録";
    }
};

#endif
```

List 4-19 Member1/VipMember.cpp

```
// 優待会員クラス（第１版：ソース部）

#include <iostream>
#include "VipMember.h"

using namespace std;

//--- コンストラクタ ---//
VipMember::VipMember(const string& name, int no, double w, const string& prv)
                    : Member(name, no, w)
{
    set_privilege(prv);          // 特典を設定
}
```

以下に示す三つのデータメンバは、基底クラス *Member* から継承されます。

- データメンバ　*full_name*, *number*, *weight*

ただし、基底クラスの非公開メンバですから、派生クラス *VipMember* のメンバ関数からのアクセスは不可能です。

また、**クラス *VipMember* はクラス *Member* から“public 派生”している**ため、基底クラス *Member* の公開メンバは、派生クラス *VipMember* でも公開されます。すなわち、以下のメンバは、一般会員クラス *Member* から継承するとともに、クラス *VipMember* の外部から利用できる状態となっています。

- メンバ関数　　*name*, *no*, *get_weight*, *set_weight*

さて、優待会員クラスでは、以下のメンバが新しく定義されています。

- データメンバ　　*privilege*
- メンバ関数　　*get_privilege*, *set_privilege*
- コンストラクタ　　*VipMember*

コンストラクタでは、網かけ部のコンストラクタ初期化子によって、名前、会員番号、体重の初期化を直接基底クラス *Member* のコンストラクタに委ねています。

＊

試作版・優待会員クラス *VipMember0* を利用するプログラム例（**List 4-6**：p.143）を、第 1 版の優待会員クラス用に書きかえたのが **List 4-20** です。

List 4-20　　Member1/VipMemberTest.cpp

```
// 優待会員クラス（第１版）の利用例

#include <iostream>
#include "VipMember.h"

using namespace std;

int main()
{
    VipMember mineya("峰屋龍次", 17, 89.2, "会費全額免除");

    double weight = mineya.get_weight();     // 峰屋君の体重
    mineya.set_weight(weight - 15.3);        // 峰屋君の体重を更新（15.3kg減量）

    cout << "No." << mineya.no() << "：" << mineya.name()
         << " (" << mineya.get_weight() << "kg) "
         << " 特典＝" << mineya.get_privilege() << '\n';
}
```

実行結果

```
No.17：峰屋龍次　(73.9kg) 特典＝会費全額免除
```

基底クラスから継承した公開メンバ関数 *no*, *name*, *get_weight*, *set_weight* を、派生クラスである *VipMember* 型のオブジェクト *mineya* に適用できることが確認できます。

＊

継承のメリットの一つは、異なる部分や追加する部分のみを作成して開発コストを抑制する**差分プログラミング**（*incremental programming*）が行えることです。うまく継承を行えば、プログラム開発の効率アップや保守性の向上が図れます。

▶ 継承の本当のメリットは差分プログラミングではありません。次節で学習する is-A の関係の実現や、それを応用した（次章以降で学習する）多相性（ポリモーフィズム）の実現です。