Vol. 36 No. 4　　情　報　処　理　　Apr. 1995

解　説

プログラミング言語最新情報-II

6. C++

—大規模システム記述を支援したオブジェクト指向プログラミング言語†—

保木本　晃弘††

1. はじめに

C++は，1980年頃，米国AT&Tベル研究所のBjarne Stroustrup博士によって開発された“C with Classes”と呼ばれるプログラミング言語がはじまりと言われている[1]．“C with Classes”には，C言語にSimula67の特徴であるクラスや派生クラスなどの機構が採り入れられていた．この“C with Classes”は，1983年に再設計が行なわれ，1985年にはC++ version 1.0[2]として広く知られることとなった．version 1.0では，クラス，派生クラス，仮想関数，演算子の多重定義などの機構が提供されている．その後，1989年には，version 1.0の機構に加えて多重継承，仮想基底クラス，純仮想関数などの追加といくつかの改良がなされたversion 2.0が公開された．そして，1990年に公開されたC++ version 3.0[3]では，テンプレート，例外処理機構などを導入することによって再利用性と可読性の高いプログラミングを可能としている．このテンプレートと例外処理機構は，version 3.0以前のC++によるプログラミング技法を根本的に変えてしまうほど大きな影響を与えている．C++は，現在，参考文献3)を基本文書としてANSIのX3J16委員会とISOのWG21委員会によって言語仕様とライブラリの標準化が進められている．このISO/ANSI C++の規格案では，新たにスコープ制御のためのnamespaceやtypeid，dynamic_cast〈T〉などの型操作機構が導入されている．

本稿では，C++が提供している様々な機構の中で主にオブジェクト指向プログラミングを支援するための基本的な機構とversion 3.0以降に導入された特徴的な機構について簡単に紹介する．2.では，オブジェクト指向プログラミングを支援するための基本的な機構について述べる．3.では，主にC++ version 3.0以降で導入された新しい機構について述べる．4.では，標準化の動向について簡単に述べる．

2. C++の基本的な機構

2.1 クラス定義とオブジェクトの生成

多くのオブジェクト指向プログラミング言語では，オブジェクトの内部構造とその振舞いを記述するためにクラスの機構を提供している．C++におけるクラスは，ユーザが定義できる型であり，オブジェクト内部のデータ群とそれに適応できる関数群・手続き群を宣言する．

たとえば，int型の値を格納するスタックのクラスは，次のように定義することができる．

```
class Stack
{
public:
  Stack (int sz)
    {top=stack=new int[size-sz];}
  Stack (const Stack& st);
    ~Stack()
    {delete[] stack; }
  bool empty() const;
  bool full() const;
  void push (const int& n)
    {* top++=n; }
  void pop(int& n)
    {n=*--top; }
private:
  int * stack;
  int * top;
  int   size;
```

† C++—An Object Oriented Programming Language by Akihiro HOKIMOTO (School of Information Science, Japan Advanced Institute of Science and Technology).

†† 北陸先端科学技術大学院大学情報科学研究科

```
};
```

C＋＋では，クラスの内部で宣言された変数をデータメンバ，関数をメンバ関数と呼んでいる．クラス名と同じ名前を持ったメンバ関数は，コンストラクタと呼ばれており，オブジェクトが生成されるときに呼び出される．このコンストラクタには，主にデータメンバを初期化するための処理を記述する．コンストラクタの中で他のオブジェクトの値によってデータメンバを初期化するコンストラクタは，コピーコンストラクタと呼ばれている．コピーコンストラクタは，

```
Stack(const Stack& st);
```

のように同じクラスのオブジェクトへの参照を引数としたインタフェースをしている．このコピーコンストラクタが明示的に宣言されていないときには，コンパイラによって自動生成される．コンパイラによって自動生成されるデフォルトコピーコンストラクタは，コピー元のオブジェクトのデータメンバの値をそのオブジェクトのデータメンバへコピーする．また，引数にとった型のオブジェクトをそのクラスのオブジェクトへ変換するためのコンストラクタは，型変換コンクトラクタと呼ばれている．この型変換コンストラクタは，

```
Complex(const Integer& number);
```

のように他のクラスのオブジェクトの参照を引数した単一引数のコンストラクタである．この型変換コンストラクタは，クラス Integer のオブジェクトをクラス Complex のオブジェクトへ変換する．また，~Stack()のようにクラス名の前に~の付いた名前を持ったメンバ関数は，デストラクタと呼ばれており，オブジェクトが消滅されるときに呼び出される．デストラクタには，そのオブジェクトが消滅することによって必要となる後処理を記述する．

クラスのメンバは，private:や public:などのキーワードを用いて明示的にアクセス制御をすることができる．public:によって宣言されたメンバは，他のオブジェクトから直接アクセスすることができる．これに対して，private:によって宣言されたメンバは，そのクラスに属するオブジェクト以外から直接アクセスすることはできない．たとえば，クラス Stack において public:で宣言されている Stack()，push()，pop()などのメンバ関数は他クラスのオブジェクトから直接アクセスすることができるが，private:で宣言された top，stack などのデータメンバは他クラスのオブジェクトから直接アクセスすることはできない．

C＋＋のオブジェクトは，

```
Stack * stack=new Stack(10);
```

のように演算子 new を用いることで明示的に生成することができる．演算子 new は，オブジェクトを格納するためのメモリ領域を確保できなかった場合には，値として 0 を返す．また，_new handler という変数にエラー処理のための関数のポインタを代入することで，メモリ領域を確保できなかったときの処理を指定することができる．演算子 new によって生成されたオブジェクトは，

```
delete stack;
```

のように delete によって明示的に消滅されるまで存在する．このようにポインタによって指されたオブジェクトのメンバ関数は，

```
stack → push(5);
```

とすることで呼び出すことができる．

また，オブジェクトは，

```
Stack stack(10);
```

のように変数として宣言することによっても生成される．関数の中で宣言されたオブジェクトは，その関数から抜けると消滅する．このように生成されたオブジェクトのメンバ関数は，

```
stack. push(5);
```

とすることで呼び出すことができる．また，

```
Stack stack(10);
Stack& s=stack;
```

のように参照を用いた場合にも

```
s. push(5);
```

とすることで呼び出すことができる．

メンバ関数の処理は，クラス Stack の push()，pop()のようにクラス定義中に記述したり，次のようにクラス定義の外に記述することができる．

```
void Stack :: push(const int& n)
{
  * top＋＋=n;
}
```

メンバ関数の処理記述の中では，現在操作しているオブジェクトのポインタが格納されている this

と呼ばれるポインタを利用することができる．このthisポインタは，主に戻り値としてオブジェクトの参照を返したりする場合に利用される．また，クラスStackのempty()やfull()のようにメンバ関数の宣言の後ろにconstが宣言されたメンバ関数は，その関数処理の中でデータメンバへの書き込みが行なわれないことを宣言している．このように宣言された関数の中でデータメンバへの書き込みを行なう処理を記述するとコンパイル時にエラーを発生する．また，Stack(const Stack& st)のようにオブジェクトへの参照がconstである場合には，その参照を用いて呼び出すことができるメンバ関数がconstで宣言されているメンバ関数に限られる．このようにメンバ関数のインタフェースをconstを用いて宣言することにより，データメンバを誤って書き換えることを避けることができる．

2.2 演算子の多重定義

C++では，シグネチャ(signature)の異なった同じ名前の関数あるいは演算子の多重定義が許されている．この関数および演算子の多重定義をうまく利用することによって，よく利用されている記法にあった可読性の高いプログラムを記述することを可能としている．

たとえば，複素数のためのクラスComplexは，演算子の多重定義を用いて定義することにより，次のように定義することができる．

```
class Complex
{
public:
  Complex(double r=0, double i=0);
  Complex(const Complex& number);
  Complex(const Integer& number);
  //…
  Complex& operator=(const Complex&
c) const;
  Complex  operator+(const Complex&
c) const;
  Complex  operator*(const Complex&
c) const;
  bool     operator==(const Com-
plex& c) const;
  //…
private:
  double real;
  double imag;
};
```

このように定義されたクラスComplexでは，次のような記述が可能となる．

```
void do_it()
{
  Complex c1(4,2);
  Complex c2=Complex(3,5);
  Complex c3=c1;
  Integer i1=3;
  c3=c1+c2;
  c3=c1+Complex(8);
  c3=c1*c2+Complex(10,8);
  c3=c1+c2+Complex(i1);
}
```

これは，通常の算術演算において利用している記法と近いため，プログラムの可読性が向上しているといえる．演算子の多重定義における演算子の優先順序は，通常の演算子の優先順序と同じである．

2.3 入れ子クラス

クラスの定義において従属関係にあるクラスを独立したクラスとして定義することは，クラス間の関係の理解を妨げる原因の一つとなる．C++では，このような状況を回避するために，クラス宣言の中で新しいクラスを宣言することを可能としている．このような入れ子クラスの名前は，そのクラスが宣言されたクラスの内部で局所的に有効となる．

たとえば，文字列を格納するクラスStringは，入れ子クラスを用いて次のように定義することができる．

```
class String
{
public:
  typedef char *       string_type;
  typedef unsigned int size_type;
private:
  class rep;
public:
  String(const string_type string="");
  String(const String& string);
  ~String();
```

```
    size_type length() const;
    //…
  private:
    rep * string_rep;
  };
  class String:: rep
  {
public:
    rep(const string_type string);
    rep(const rep& string);
    ~rep();
    size_type length() const;
    //…
  private:
    int          count;
    string_type str;
  };
```

このクラス String のクラス定義中に宣言されたクラス rep は，入れ子クラスである．入れ子クラスの定義は，クラス定義の中あるいは上記の String::rep のようにクラスの外で定義することができる．入れ子クラス String::rep のアクセス制御は，通常のアクセス制御規則に従う．

2.4 派生クラス

クラス継承は，クラスの拡張性を支援するための機構の一つである．C＋＋では，クラス継承を支援するための機構として派生クラスを提供している．派生クラスのもとになるクラスは，基底クラスと呼ばれている．派生クラスは，基底クラスからデータメンバとメンバ関数を継承する．ただし，コンクトラクタ/デストラクタ，代入演算子などは継承されない．派生クラスは，基底クラスから継承したメンバを再定義したり，新しいメンバを追加することができる．

たとえば，ファイルシステムのパス名のクラス PathName は，派生クラスを用いて次のように定義することができる．

```
  class PathName: public String
  {
  public:
    PathName(const String& pathname);
    PathName(const PathName& pathname);
      ~PathName();
    PathName& operator=(const String& path);
    PathName& operator=(const PathName& path);
    void basename(String& base_name);
    void prefix(String& pref);
    void suffix(String& suff);
    //…
  private:
    String       basename_rep;
  };
```

クラス PathName は，クラス String から派生したクラスであり，クラス String で定義されているメンバを継承している．このため，クラス PathName では，クラス String のメンバを自分のメンバとして利用することができる．

C＋＋では，基底クラスに対してもアクセス制限が適用される．**表-1** に派生クラスと基底クラスのアクセス制御の関係を示す．たとえば，基底クラスのメンバが public :で，基底クラスの指定方法が public の場合には派生クラスのオブジェクトおよび派生クラス以外のオブジェクトともに基底クラスのメンバにアクセスできる．これに対して，基底クラスのメンバが protected :で，基底クラスの指定方法が public の場合には，派生クラスのオブジェクトは基底クラスのメンバにアクセスできるが，派生クラス以外のオブジェクトはアクセスすることができない．

2.5 抽象クラス

C＋＋では，インタフェースのみを規定し，実装を持っていないメンバ関数を定義することができる．このようなメンバ関数は，純仮想関数と呼ばれており，実行時に派生クラスで定義されている同じインタフェースの実装を持ったメンバ関数

表-1 公開，非公開，保護の関係

基底クラスの指定方法	基底クラスのメンバ					
	private：		public：		protected：	
	派生	他	派生	他	派生	他
private	×	×	×	×	×	×
public	×	×	○	○	○	×
protected	×	×	○	×	○	×

注意：○はアクセス可能，×はアクセス不可を意味する．また，派生は派生クラスのオブジェクト，他は派生クラス以外のオブジェクトである．

にバインドされる．

たとえば，多角形，四角形，正方形，楕円，円などを含む図形クラス群は，次のように定義することができる．

```
class Shape
{
public:
  virtual void draw()=0;
  virtual void erase();
  virtual void move(const Point& point);
  //…
};
class Polygon: public Shape
{
public:
  virtual void draw();
  virtual void erase();
  virtual void move(const Point& point);
  //…
};
class Rectangle: public Polygon {…};
class Square: public Rectangle {…};
class Ellipse: public Shape {…};
class Circle: public Ellipse {…};
```

このクラス Shape の定義にある，

```
virtual void draw()=0;
```

は，draw()が純仮想関数であることを宣言している．C++では，このような純仮想関数を持ったクラスを抽象クラスと呼んでおり，そのクラスのオブジェクトを生成することができない．また，

```
virtual void erase();
```

は，erase()が仮想関数であることを宣言している．仮想関数は，純仮想関数と同様にその実装は実行時にバインドされるが，そのメンバ関数自身が実装を持っている．このため，仮想関数を持っているが，純仮想関数を持っていないクラスのオブジェクトは生成することができる．ここで定義されている図形クラス群のクラス階層は，**図-1** のようになる．

一つ以上の純仮想関数あるいは仮想関数を持ったクラスの派生クラスのオブジェクトは，これらのクラスのポインタあるいは参照によって参照することができる．たとえば，上で定義したクラス群は，抽象クラス Shape のポインタを用いることによって次のような記述ができる．

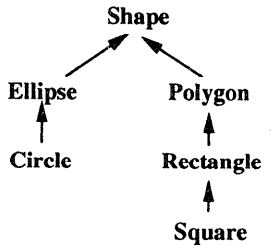

図-1 図形クラスのクラス階層例

```
void do_it()
{
  Shape * polygon=new Polygon(…);
  Shape * circle=new Circle(…);
  polygon → draw();
  polygon → erase();
  circle → draw();
}
```

このとき，Shape のポインタ polygon，circle が指しているオブジェクトは，それぞれクラス Polygon と Circle のオブジェクトである．この記述において draw()と erase はどちらも仮想関数である．このため，polygon → draw()，polygon → erase()，circle → draw()の呼出しは，それぞれ Polygon:: draw()，Polygon:: erase()，Circle:: draw()にバインドされて呼び出される．

このように抽象クラスのような多相型を用いることによって，異なる型のオブジェクトを規定されたインタフェースによって統一的に扱うことができる．このため，オブジェクトの違いを意識することなく処理を記述することが可能となる．

また，抽象クラスは，同じ役割を持ったクラスの多重実装を実現するためにも利用することができる．たとえば，計算機のハードウェア仕様に依存するクラスを抽象クラスとして定義し，それぞれのハードウェアに合わせた実装を持ったクラスを定義することにより，ハードウェアに依存しないクラスをあまり変更することなく，他の計算機へ移植することが比較的容易なシステム構築することが可能となる．

3. C++ version 3.0 以降に導入された機構

本章では，C++ version 3.0 以降で導入されたテンプレート，例外処理機構および ISO/ANSI C++の規格案で導入された型操作，名前空間について簡単に紹介する．

3.1 テンプレート

ソフトウェアの開発においてモジュールの再利用性は，非常に重要な問題の一つである．特に，大規模システムソフトウェアでは，クラスや関数の再利用は，小規模なシステムと比較して深刻な問題となる．C++ version 3.0 以降で導入されたテンプレート機構は，クラスや関数の再利用性を高める上で重要な機構の一つである．

3.1.1 クラステンプレート

C++が提供するクラステンプレートは，クラスの雛型を与えることによってクラスがどのように構成されるかを定義したものである．テンプレート引数として型や整数などを与えることによって，引数の値を確定したテンプレートクラスを生成する．たとえば，クラステンプレート List とクラステンプレート Stack は次のように定義することができる．

```
template <class T>
class List
{
public:
  typedef T       value_type;
  typedef size_t size_type;
  //…
private:
  struct node;
public:
  class iterator;
  //…
public:
  bool       empty() const;
  size_type size() const;
  void insert(const value_type& v, iterator& i);
  void remove(value_type& v, iterator& i);
  void push_first(const value_type& v);
  void pop_first(value_type& v);
  //…
};
template <class Container>
class Stack
{
public:
  typedef  Container:: value type  value_type;
  typedef  Container:: size_type  size_type;
public:
  bool          empty() const
                {return c. empty(); }
  size_type     size() const
                {return c. size(); }
  void          push(const  value_type& v)
                {c. push_first(v); }
  void          pop(value_type& v)
                {c. pop_first(v); }
protected:
  Container c;
};
```

クラステンプレート List の定義にある template <class T> は，それが T をテンプレート引数としたクラステンプレートであることを示している．クラステンプレート定義で宣言されている T は，テンプレート引数で指定されている型に置き換えられる．たとえば，int 型のリストや char 型のスタックは，クラステンプレート List やクラステンプレート Stack を用いて次のように宣言される．

```
List <int>                list;
Stack <List <char>> stack;
```

このとき，Stack <List <char>> は，クラス List によって実装された char のスタックである．また，クラステンプレート Stack のインタフェース規則に合わせてクラステンプレート Vector を宣言することによって，

```
Vector <int>              vector;
Stack <Vector <char>> stack;
```

のようにクラステンプレート Vector によって実装されたテンプレートクラス Stack <Vector

〈〈char〉〉を宣言することができる．

テンプレート引数には，整数を指定することもできる．たとえば，クラステンプレート Buffer は次のように定義することができる．

```
template <class T, int sz>
class Buffer
{
  T buf[sz];
};
```

この定義では，バッファのサイズ sz がテンプレート引数として与えられている．このため，クラステンプレート Buffer のバッファサイズはコンパイル時に静的に決定される．クラステンプレート Buffer のテンプレートクラスは次のように宣言される．

```
Buffer <int, 10>    int_buf;
Buffer <char, 10>   char_buf10;
Buffer <char, 20>   char_buf20;
```

このとき，Buffer〈char, 10〉と Buffer〈char, 20〉は，異なる型として扱われる．

3.1.2 関数テンプレート

関数テンプレートは，メンバ関数あるいは大域関数にアルゴリズムの雛型を与えることによって，関数がどのような処理を行なうかを定義したものである．関数テンプレートは，クラステンプレート引数を確定することによってテンプレート関数を生成する．

たとえば，2つの引数を持った関数 max()は，関数テンプレートを用いて次のように定義することができる．

```
template <class T>
inline
 const T& max(const T& a, const T& b)
{
  return a>b? a: b;
}
```

このとき，

```
int    i=max(3, 4);
float  f=max(5.7, 3.1);
```

などは，i=4，f=5.7 となる．

関数テンプレートは，関数のシグネチャが引数の型あるいは数で区別できるならば多重定義することができる．たとえば，上記で定義した関数 max()は，

```
char * s=max("abc", "def");
```

の場合，文字列のポインタの大きさを比較してしまう．このとき，ポインタの比較ではなく文字列の比較を行なうためには，次のように max()の多重定義を行なえばよい．

```
template <class T>
inline const T& max(const T& a, const T& b)
{
  return a>b? a: b;
}
inline char * max(char * s1, char * s2)
{
  return (strcmp(s1, s2)>0)? s1: s2;
}
```

3.2 例外処理機構

大規模なシステムソフトウエアでは，実行時にシステムエラーが発生した場合であっても実行を止めることなくエラーに対応することが要求されることがある．このようなエラーによる例外処理を支援するために，version 3.0 以降の C++では，catch/throw と呼ばれる機構を導入した．この catch/throw を用いたプログラムでは，エラーをキーワード throw を用いて例外ハンドラに知らせる．throw は，try の中で実行されたコードあるいはそこから呼び出された関数の中でのみ有効である．try の中で throw された例外オブジェクトは，try の後に定義された catch によって受け取られ，そこに記述された処理を実行する．

たとえば，catch/throw を用いたクラステンプレート Vector は次のように定義することができる．

```
class memory_exhausted {…};
template <class T>
class Vector
{
public:
  typedef T          value_type;
  typedef size_t     size_type;
private:
  const size_type max=10000;
public:
  class bad_size
```

```
  {
  public:
    bad_size(size_type sz){size=sz;}
    //…
    size_type size;
};
  class bad_range;
public:
  Vector(size_type sz=100)
  {
    if (sz<0 ‖ max<=sz)
      throw bad_size(sz);
    if ((v=new value_type[s=sz])==0)
      throw memory_exhausted();
  }
  value_type& operator[] (size_type i)
const
  {
    if (i<0‖s<=i)
      throw bad_range(i);
    return v[i];
  }
  //…
private:
  value_type * v;
  size_type    s;
};
```

このクラステンプレート Vector では，自由記憶領域を使い果たした(memory_exhausted)，添字のレンジオーバ(bad_range)，サイズ指定エラー(bad_size)などの例外を定義している．このクラステンプレート Vector は，try を用いて次のように記述することができる．

```
void do_it()
{
  try {
    Vector <int> v(10);
    do_something(v);
  }
  catch (memory_exhausted& m) {
    cerr≪m.error_message()≪endl;
    exit -1;
  }
  catch (Vector <int> :: bad_range& r) {
    cerr≪r.error_message() ≪endl;
    exit -1;
  }
  catch (Vector <int> :: bad_size& s) {
    cerr≪s.error_message() ≪endl;
    exit  1;
  }
}
```

このとき，do_something()が

```
void do_something(Vector <int> & v)
{
  int i=v[3];
}
```

ならば，エラーが発生することなく実行される．ところが，次のようなプログラムでは，Vector <int> :: bad_range の例外が発生する．a

```
void do_something(Vector <int> & v)
{
  int i=v[100];
}
```

i=v[100]; で発生した例外は，do_it()内の catch(Vector <int> :: bad_range& s)によって受け取られ，ここに記述された例外処理を実行する．

発生した例外が catch によって受け取ることができなかったときには，デフォルト例外処理として関数 terminate()が呼び出される．terminate()は，スタックが壊れているのを発見した場合などにも呼び出される．terminate()は，

```
typedef void (* vfp)();
vfp set_terminate(vfp func);
```

のパラメータとして与えられた関数の最も新しいものを実行する．set_terminate()の戻り値は，更新前の関数のポインタである．terminate()のデフォルトとしては abort()が使用されている．

3.3 型 操 作

ISO/ANSI C++の規格案では，型情報が処理系によって支援され，static_cast，reinterpret_cast，const_cast，dynamic_cast，typeid などの型を扱う操作が追加された．ここでは，dynamic_cast と typeid について簡単に述べる．

dynamic_cast <T> (v)は v の型を型 T で指定された型に変化するキャストである．型 T は，すでに定義されているクラスのポインタか参照，

または void * でなければならない．また，v は，多相型のポインタまたは参照でなければならない．T は，v のクラスを基底クラスとしたポインタあるいは参照であり，結果として派生クラスのオブジェクトへのポインタあるいは参照を返す．このとき，v のクラスは一つ以上の仮想関数を持っている必要がある．v が T に変換できない場合，ポインタ型のキャストであれば 0 を返し，参照の場合には例外として bad_cast を throw する．たとえば，

```
class Ellipse
{/* at least one virtual function */};
class Circle: public Ellipse
{…};
Ellipse * pe1=new Ellipse;
Ellipse * pe2=new Circle;
Circle * pc1=dynamic_cast <Circle *>
(pe1);
Circle * pc2=dynamic_cast <Circle *>
(pe2);
```

において，dynamic_cast<Circle *>(pe1)は 0 を返し，dynamic_cast <Circle *> (pe2)はクラス Circle のオブジェクトを指す．

typeid は，変数の型情報として const type_info&を返す．もし，引数が多相型の参照ならば，実際のオブジェクトの型が返される．もし，エラーが発生した場合には，例外として bad_typeid を throw する．たとえば，

```
class X
{
  // class X is polymorphic
  //…
  virtual void f();
};
```

のように一つ以上の仮想関数を持ったクラス X において，

```
void do_it(X * p)
{
  type_info& typeinfo1=typeid(p);
  type_info& typeinfo2=typeid(* p);
}
```

のとき，typeid(p)は，X * の type_info を返し，type(* p)はクラス X の type_info あるいはクラス X から派生したクラスの type_info を返す．

クラス type_info は，type_info. h において次のように定義されている．

```
class type_info
{
private:
  type_info(const type_info&);
  type_info& operator = ( const type
_info&);
public:
  virtual ~type_info();
  int operator==(const type_info&)
const;
  int operator!=(const type_info&)
const;
  int before(const type_info&);
  const char * name() const;
  //…
};
```

たとえば，typeid を用いて次のような記述が可能となる．

```
void do_it(const Shape& s1, const
Shape& s2)
{
  if (typeid(s1)==typeid(s2)){
    //…
  }
}
```

この例において，if 文の処理は s 1 と s2 の型が同じ場合にのみ処理される．

3.4 名前空間

複数のグループによって開発されるソフトウェアや複数のクラスライブラリを利用する開発では，クラスや関数の名前の衝突が問題となることがある．ISO/ANSI C++の規格案では，従来の大域的な名前のスコープを制御するために，名前空間と呼ばれる機構を導入した．ある名前空間の中で宣言した型，オブジェクト，関数などは名前空間のメンバであり，この名前空間の外で定義された名前と衝突しない．たとえば，名前空間の宣言は次のように記述することができる．

```
namespace stdlib
{
  class String {…};
  void f(String& s);
```

```
};
namespace mylib
{
  class String {…};
  void f(String& s);
};
```

ここでは，namespace stdlib と namespace mylib のクラス String は，名前空間が異なるために名前の衝突が起きない．名前空間のメンバは，名前空間の中で定義することができる．また，名前の付けられた名前空間のメンバは，次のように名前空間の外部に定義することもできる．

```
void stdlib:: f(String& s)
{
  String ss ="asdf";
}
```

関数の中で利用する名前空間は，using を用いて指定することができる．using によって指定された名前空間のメンバは，それが宣言された関数の中で大域スコープで宣言されたのと同じように利用することができる．

たとえば，名前空間 stdlib を利用する関数は using を用いて次のように記述することができる．

```
void do_it()
{
  using namespace stdlib;
  String s; // stdlib:: String;
  f(s);      // stdlib:: f;
  mylib:: String s2;
  mylib:: f(b2);
}
```

また，using は，

```
void do_it()
{
  using stdlib:: String;
  using stdlib:: f;
  //…
}
```

のように利用する名前空間のメンバを直接指定することができる．

また，派生クラスは，using を用いて使用する基底クラスのメンバを厳密に指定することができる．たとえば，

```
class Base
{
public:
  virtual void f(int);
  virtual void f(double);
  //…
};
class Derived: public Base
{
public:
  using Base:: f;
  virtual void f(int);
  virtual void f(char); // add new f
  //…
};
```

のように明示的に記述することができる．

4. 標準化の動向

C++の標準化作業は，1989年12月の米国ワシントンD.C.における ANSI X3J16 委員会の発足により始まった．ANSI X3J16 委員会によって標準化作業が本格的に始まる以前は，C++の事実上の標準として参考文献1)，3)と AT&T の cfront の3つのバージョンが存在していたが，ANSI X3J16 委員会はこれらのバージョンの中で3)を基本文書として選択した．そして1994年，ANSI X3J16 委員会は，5年前にわたる標準化作業の結果として ISO の手続きの第一段階の投票のための規格案を可決した．この規格案の可決によって，標準C++の主要な要素のガイドラインが整ったといえる．また，この標準化作業の中で，標準C++ライブラリの詳細についても検討されてきた．この標準C++ライブラリの規格案の一部は，参考文献4)として出版されている．また，1994年7月にカナダのウォータルーで開催された ISO WG21 と ANSI X3J16 の合同委員会では，ヒューレットパッカード社の標準テンプレートライブラリ(Standard Template Library)[5]を正式に規格案に取り込むことなどが承諾された．さらに，1995年2月に公開された ANSI/ISO Resolutions[6]の中では，新しく explicit，typename などが追加されている．

たとえば，typename の追加によって従来までクラステンプレートで使用されていた，

```
template <class T>
class Stack
{
public:
  typedef T:: value_type value_t;
  T:: value_type v;
};
```

のような記述はエラーとなり，

```
template <class T>
class Stack
{
public:
  typedef typename T:: value_type value
_t;
  typename T:: value_type v;
};
```

のようにしなければならない．C++の最初の標準規格案は，1995年の春頃，公開レビューのために公開される予定である．なお，C++の標準化の詳細に関しては，参考文献7)を参照するとよい．

5. おわりに

本稿では，C++が提供するいくつかの特徴的な機構について簡単に紹介してきた．ここで紹介した特徴は，C++の仕様のほんの一部でしかないが，C++によるプログラミングの雰囲気をつかんでいただければ幸いである．

また，紙面の都合で紹介することができなかったが，現在までにC++を拡張しようとする研究が盛んに行なわれてきた．たとえば，C++に新しい多相型を導入する研究としてSignatures[8]がある．また，C++に分散や並列プログラミングを支援するための研究としては，参考文献9)～16)などがある．C++や拡張C++を用いて構築された比較的有名なシステムとしては参考文献17)～22)などがある．

最後に非常に簡単ではあるがC++クラスライブラリについて述べる．以前，C++のクラスライブラリとして有名であったものには，InterViews[23]，ET++[24]，NIHCL[25]などがある．InterViewsは，X Window System用のクラスライブラリであり，Windowアプリケーションを容易に構築するためのツールキットを提供している．ET++は，Window Systemに依存しないグラフィックインタフェースを実現するためのライブラリである．ライブラリがOSやWindow Systemに依存しないようにシステム依存の部分を抽象クラスによって定義している．OSやWindow Systemに依存した部分は，これらの派生クラスとして実現することによって様々なシステムに移植することを可能としている．NIHCLは，Smalltalk80に似たクラスライブラリを提供している．しかし，これらのクラスライブラリは，テンプレートなどが導入される以前の古いC++に対応したものである．このため，テンプレート，例外処理機構，標準ライブラリなどの導入によってクラスライブラリの構築技法も今後大幅に変更されることが予想される．

また，ANSIの規格案に採用することが承諾されたHP社のSTL(Standard Template Library)[5]は，algorithm, container, iterator, function object, adapterの5つの要素を含んでいる．今後は，ISO/ANSI C++の言語仕様や標準クラスライブラリとともに標準テンプレートライブラリなどの動向についても注目する必要があるだろう．

謝辞 本稿執筆の機会を与えていただいた東京工業大学情報処理センターの松田裕幸氏，北陸先端科学技術大学院大学情報科学研究科の渡部卓雄氏ならびに閲読者の方々に感謝いたします．

参考文献

1) Stroustrup, B.: The C++ Programming Language, second edition, Addison-Wesley (1991), 邦訳「プログラミング言語C++第2版」斉藤，三好，追川，宇佐美 訳，トッパン．
2) Stroustrup, B.: The C++ Programming Language, Addison-Wesley (1986), 邦訳「プログラミング言語C++」斉藤 訳，トッパン．
3) Ellis, M. and Stroustrup, B.: The Annotated C++ Reference Manual, Addison-Wesley (1990), 邦訳「注解C++リファレンスマニュアル」足立，小山 訳，トッパン．
4) Plauger, P.: The Draft Standard C++ Library, Prentice Hall (1995).
5) Stepanov, A. and Lee, M.: The Standard Template Library, Technical Report HPL-94-34, Hewlett-Packard Laboratories (1994).
6) Stroustrup, B.: ANSI/ISO Resolutions, in ftp: //ftp. std. com/AW/stroustrup2e/new_iso. ps (1995).
7) Stroustrup, B.: The Design and Evolution of

C++, Addison-Wesley (1994).
8) Baumgartner, G. and Russo, V.: Signatures: A C++ Extension for Type Abstraction and Subtype Polymorphism, Technical Report CSD-TR-93-059, Purdue University (1993).
9) Chandy, K. and Kesselman, C.: C++: A Declarative Concurrent Object Oriented Programming Notation, in Research Directions in Concurrent Object Oriented Programming (1993).
10) Chiba, S.: Open C++ Release 1.2 Programmer's Guide, Technical Report No. 93-3, The University of Tokyo (1993).
11) Gourhant, Y. and Shapiro, M.: FOG/C++: a Fragmented-Object Generator, in Proceedings of C++ Conference '90 USENIX (1990).
12) Ishikawa, Y., Tokuda, H. and Mercer, C.: Object-Oriented Real-Time Language Design: Constructs for Timing Constraints, in Proceedings of OOPSLA '90 (1990).
13) Ishikawa, Y.: The MPC++ Programming Language V 1.0 Specification with Commentary Document Version 0.1, Technical Report TR-94014, Real World Computing Partnership (1994).
14) Kale, L. and Krishnan, S.: CHAME++: A Portable Concurrent Object Oriented System Based On C++, in Proceedings of OOPSLA '93 (1993).
15) Larus, J., Richards, B. and Viswanathan, G.: C**: A Large-Grain, Object-Oriented, Data Pallalel Programming Language, Technical Report 1126, University of Wisconsin-Madison (1992).
16) Seliger, R.: Extending C++ to Support Remote Procedure Call, Concurrency, Exception Handling, and Garbage Collection, in Proceedings of C++ Conference '90 USENIX (1990).
17) Bershad, B., Lazowska, E. and Levy, H.: Presto: A System for Object Oriented Parallel Programming, Software: Practice and Experience, Vol. 18, No. 8 (1988).
18) Campbell, R., Islam, N., Raila, D. and Madany, P.: Experiences Designing and Implementating an Object-Oriented System in C++, Communications of the ACM, Vol. 36, No. 9 (1993).
19) Christopher, W., Procter, S. and Anderson, T.: The Nachos Instructional Operating System, Technical Report ftp: //ftp.cs.berkeley.edu/ucb/nachos/, University of California Berkeley (1992).
20) Dixon, G., Parrington, G., Shrivastava, S. and Wheater, S.: The Treatement of Persistent Objects in Arjuna, in Proceedings of ECOOP '89 (1989).
21) Hamilton, G. and Kougiouris, P.: The Spring Nucleus: A Microkernel for Objects, Technical Report SMLI TR-93-14, Sum Microsystems Laboratories Inc. (1993).
22) Yokote, Y.: The Apertos Reflective Operating System: The Concept and Its Implementation, in Proceedings of OOPSLA '92 (1992).
23) Linton, M. and Calder, P.: The Design and Implementation of Interviews, in Proceedings of C++ Conference '87 USENIX (1987).
24) Weinand, A., Gamma, E. and Marty, R.: The Design and Implementation of ET++, a Seamless Object-Oriented Application Framework, Structured Programming, Vol. 10, No. 2 (1989).
25) Gorlen, K., Orlow, S. and Plexico, P.: Data Abstraction and Object-Oriented Programming in C++, John Wiley and Sons (1990).

(平成6年1月19日受付)

保木本 晃弘

1968年生. 1990年3月(労働省所管)職業訓練大学校情報工学科卒業. 1992年3月アロカ(株)退社. 1994年3月北陸先端科学技術大学院大学情報科学研究科博士前期課程修了, 北陸先端科学技術大学院大学情報科学研究科博士後期課程在学中. 移動計算機環境, オペレーティングシステム, 実時間処理, オブジェクト指向計算に興味を持つ.