情報システム工学II(システム開発工学)

オブジェクト指向プログラミング

Object-Oriented and Generic Programming in C++

山田泰司

taiji@aihara.co.jp

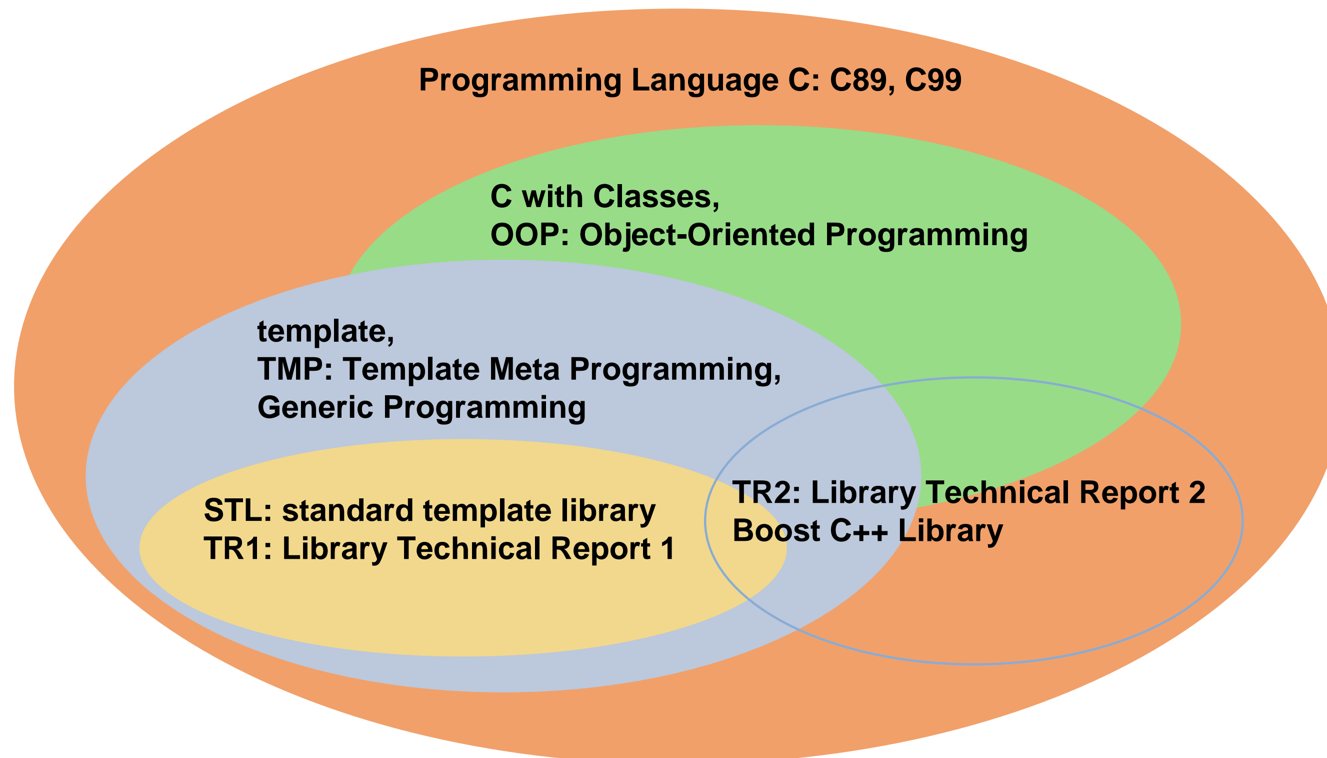
株式会社あいはら 研究開発チーム

プログラミング言語C++とは

1. **プログラミング言語** C: C89, C99
2. **オブジェクト指向** C++: **クラス付き** C
3. **テンプレート** C++: **ジェネリックプログラミング、**TMP: **テンプレートメタプログラミング**
4. **標準テンプレートライブラリ** (STL): **コンテナ、イテレータ、アルゴリズム、関数オブジェクトなど**

の「マルチパラダイムプログラミング言語」である。

C++0x **の正式な標準化** (C++11) **とコンパイラの対応により、将来的には、さらに効率的、さらに高度なプログラミングが可能となろう。**



Programming Language C++: C++98, C++03, C++0x

プログラミング言語C++の学び方

1. M.A. Ellis & B. Stroustrup,“The Annotated C++ Reference Manual,” 1990, 1992. **足立、小山訳**, **「注解 C++リファレンスマニュアル」**, Addison-Wesley, 1992, 1993.
2. S. Meyers, “Effective C++: 55 Specific Ways to Improve Your Programs and Designs, 3rd Edition,” 2005. **小林 訳**, **「Effective C++ 第 3 版：プログラムとデザインを改良するための 55 項目」**, Pearson Education, 2006, 2007.
3. D.R. Musser, G.J. Derge & A. Saini, “STL Tutorial and Reference Guide: C++ Programming with the Standard Template Library, 2nd Edition,” Addition-Wesley, 2001. **滝沢、牧野 訳**, **「STL—標準テンプレートライブラリによる C++プログラミング 第 2 版」**, Pearson Education, 1997, 2001.
4. S. Meyers, “Effective STL: 50 Specific Wayes to Improve Your Use of the Standard Template Library,” 2001. **細谷 訳**, **「Effective STL: STL を使いこなす 50 の鉄則」**, Pearson Education, 2002, 2005.

その他の入門書で「突っ掛かり」を掴んでも良いが、必ず文献 1. で体系的に理解しておくこと。そして必ず、文献 2. を「読破」すること。さもないと、オブジェクト指向を「乱用」しがちなプログラマになりやすい。そして、わからなければいつでも文献 1. に立ち戻ること。

以上とは独立して、例えば文献 3. で STL に慣れること。すべての汎用アルゴリズムについて理解しておく必要はないが、背後の「実装」について、その計算量を常に意識できる基礎知識が必須となる。文献 4. がそれを補ってくれるだろう。

クラス付きC

oopgp/ex00.cc:

```
class B { // 基底クラス
public: // パブリックメンバ
  int a;                                          // = 10;              /* ここで初期化子は NG */
  double b[3];                                    // = { 20, 21, 22 }; /* ここで初期化子は NG */
  static int c;                                   // = 0;               /* ここで初期化子は NG */
  // コンストラクタ : メンバ初期化子リスト
  B() : a(10), e("50"), f(M_PI) {}                // , c(0)             /* ここで静的メンバ変数の初期化子は NG */
  B(int a) : a(a), e("51"), f(M_PI) {}            // コンストラクタの引数による多重定義
  //B(const B &that) { *this = that; }            // デフォルトのコピーコンストラクタ
  B(const B &that) { *this = that; a = 12; }      // コピーコンストラクタの定義
  // デストラクタ
  virtual ~B() {}                                 // デストラクタの定義、派生クラスのための仮想
  // メンバ・オペレータ関数
  const double &operator[](std::size_t pos) const // コンストメンバ関数
  {
    return d[pos];
  }
  double &operator[](std::size_t pos)             // 非コンストメンバ関数
  {
    //return d[pos];
    return const_cast<double &>(static_cast<const B &>(*this)[pos]);
  }
  // メンバ関数
  const char *get_e() const { return e; }         // コンストメンバ関数
  double &ref_f() { return f; }                   // 非コンストメンバ関数
private: // プライベートメンバ
  static double d[3];
  const char *e;
  double f;
};
int B::c = 30;                                    // 静的メンバ変数の初期化子
double B::d[] = { 40, 41, 42 };                   // 静的メンバ変数の初期化子
```

クラス付きC（つづき）

```
class D : public B { // 派生クラス、パブリック継承
public: // パブリックメンバ
  int a;

  // コンストラクタ : メンバ初期化子リスト
  D() : B(), a(19), f(M_E) { init(); }
  D(const D &that) : B(that), a(19), f(M_E) { init(); } // コピーコンストラクタの多重定義

  // デストラクタ
  virtual ~D()                                         // デストラクタの多重定義
  {
    delete[] g;
  }
  double &ref_f() { return f; }                        // メンバ関数の多重定義
  friend const double *get_g(const class D &that);     // フレンド関数の宣言、非メンバ関数

private: // プライベートメンバ
  double f;
  double *g;
  void init()                                          // プライベート・メンバ関数
  {
    double g[3] = { 60, 61, 62 };
    this->g = new double[3];
    memcpy(this->g, g, sizeof(g));
  }
};

// class D とのフレンド関数
const double *get_g(const class D &that) { that.c = 39; return that.g; }
```

クラス付きC（つづき）

```
#include <iostream>      // 標準入出力 for C++
#include <cmath>         // math. for C++
#include <string.h>      // memcpy, etc of C
        :
int main()
{
  std::cout << B::c << '\n';            // 静的メンバ変数 B::c

  B X, Y(11), Z(Y);                     // 基底クラス B のインスタンス X(), Y(11), Z(Y)
  X.c = 31;                             // 静的メンバ変数 B::c
  X[2] = Y[2] = Z[2] = 49;              // 基底クラス B の非コンストメンバ関数、オペレータ []
  std::cout << "B X. " << X.a << X.b[0] << X.c << X[2] << X.get_e() << X.ref_f() << '\n';
  std::cout << "B Y. " << Y.a << Y.b[0] << Y.c << Y[2] << Y.get_e() << Y.ref_f() << '\n';
  std::cout << "B Z. " << Z.a << Z.b[0] << Z.c << Z[2] << Z.get_e() << Z.ref_f() << '\n';

  D W(static_cast<const D &>(Z));       // 派生クラス D のインスタンス W(Z)
  B *V = new D(W);                      // 基底クラス B 型ポインタ *V = new D(W)
  D *U = dynamic_cast<D *>(V);          // 派生クラス D 型ポインタ *U = (D *)V

  std::cout << "D W. " << W.a  << W.b[0]  << W.c  << W[2]    << W.get_e()  << W.ref_f()  << W.B::ref_f() << get_g(W)[0] ;
  std::cout << "B V->" << V->a << V->b[0] << V->c << (*V)[2] << V->get_e() << V->ref_f() << U->ref_f()   << get_g(*U)[0];

  using std::swap;
  swap(W.ref_f(), W.B::ref_f());
  swap(V->ref_f(), U->ref_f());
  std::cout << "D W. " << W.a  << W.b[0]  << W.c  << W[2]    << W.get_e()  << W.ref_f()  << W.B::ref_f() << get_g(W)[0] ;
  std::cout << "B V->" << V->a << V->b[0] << V->c << (*V)[2] << V->get_e() << V->ref_f() << U->ref_f()   << get_g(*U)[0];
  delete V;
  return 0;
}
```

まず、静的メンバ変数は、すべてのインスタンス経由でアクセスできるものであるが、実体はたった一つしかないことがわかる。
また、プライベートメンバへはアクセス不能であるからして、パブリックメンバ関数、フレンド非メンバ関数を介してアクセスせざるを得ないことに注意しよう。
また、派生クラスは基底クラスの振る舞いを継承しつつ、さまざまな拡張が成されてることに注目しよう。
また、const メンバ関数では、ここでは、メンバ変数に変更を加えない、加えられないことにも注目しよう。

（例題１）クラスの定義

1. 先のプログラム `ex00.cc` の出力が、なぜそのような値になるか、説明せよ。
   （ヒント）どのコンストラクタもしくは関数が呼ばれているのか、常に意識できるようになること。
2. オペレータ `[]` の返り値の型 `double &`で、`const` と非 `const` の片方どちらかがない場合、どのようになるか説明せよ。
   （ヒント）＊＊＊が無ければ＊＊＊で代用される。＊＊＊が無ければ、代入をするところで＊＊＊となる。
3. また、デストラクタが仮想関数でない場合、どのような不都合が起こるか、述べよ。
   （ヒント）派生クラスを指す基底クラス型ポインタを `delete` した場合、どのデストラクタが呼ばれるのか？
4. また、メンバ変数 `f` は派生クラスのインスタンスでいくつあるのか答えよ。
   （ヒント）容易にアクセス出来なくなっているだけ。上例の `W.B::ref_f()` と `U->ref_f()` を見よ。
5. そして、基底クラスのメンバ関数 `ref_f()` を仮想関数に変更すると、出力はどのように変化するか答えよ。
   （ヒント）仮想関数であればいつでも派生クラスのメンバ関数が呼ばれるようになることに留意せよ。
6. さらに、その出力の変化が生じないようにするには、`std::cout`、`swap` の引数をどのように変更すれば良いか答えよ。
   （ヒント）`W.B::ref_f` のようなメンバ関数の指定の仕方を参考にせよ。

STL コンテナ：可変長配列 vector

oopgp/container00.cc:

```
#include <vector>
       :
  using std::string;                            // ちなみに、string の実体は std::basic_string<char>
  const string names[] = { "Taro", "Hanako", "Jiro", "Keiko", };        // string 型の配列
  const string family_names[] = { "Kato", "Yamada", "Ikeguchi", };      // string 型の配列

  std::cout << "names: ";                       // まずは、ただ出力するだけ。std::cout << 任意の型、を使う
  for (size_t i=0; i<sizeof(names)/sizeof(string); i++)
    std::cout << names[i] << ", ";
  std::cout << "\n";
  std::cout << "family names: ";
  for (size_t i=0; i<sizeof(family_names)/sizeof(string); i++)
    std::cout << family_names[i] << ", ";
  std::cout << "\n";

  std::vector<string> full_names;               // string 型の vector
  for (int i=0; i<int(sizeof(family_names)/sizeof(string)); i++)
    for (int j=0; j<int(sizeof(names)/sizeof(string)); j++)
      full_names.push_back(family_names[i] + " " + names[j]);
                                                // string を結合「+」したものを full_names の後ろに格納

  std::cout << "\nfull names: ";                // full_names を出力
  for (std::vector<string>::iterator it = full_names.begin();
       it != full_names.end(); ++it)            // string 型の vector のイレテータ（反復子）
    std::cout << *it << ", ";
  std::cout << "\n";
       :
```

イテレータ（反復子）のループスタイルに慣れよう。

STL コンテナ：双方向リスト list

oopgp/container02.cc:

```
#include <list>
#include <netinet/in.h> // struct in_addr, etc
#include <arpa/inet.h>  // inet_ntop, etc
        :
  const char *const addrs[] = {                   // ポインタも内容も const
    "60.32.126.145",
        :
    "172.16.127.1",
  };

  std::list<struct in_addr> in_addr_list;         // struct in_addr 型の list

  for (size_t i=0; i<sizeof(addrs)/sizeof(char *); i++) {
    struct in_addr addr;
    if (inet_pton(AF_INET, addrs[i], &addr) != 1) continue;      // inet_pton で変換できたら
    in_addr_list.push_back(addr);                 // addr を in_addr_list の後ろに格納
  }

  for (int a=1; a<argc; a++) {                    // コマンドライン引数についても同様に行なう
    struct in_addr addr;
    if (inet_pton(AF_INET, argv[a], &addr) != 1)
      continue;
    in_addr_list.push_front(addr);                 // addr を in_addr_list の『前に』格納
  }

  std::cout << "in_addr list: ";                  // in_addr_list を出力
  for (std::list<struct in_addr>::iterator it = in_addr_list.begin();
       it != in_addr_list.end(); ++it) {          // struct in_addr 型の list のイテレータ
    char b[64];
    std::cout << inet_ntop(AF_INET, &*it, b, sizeof(b)) << ", ";        // ここで「&*」は必要
  }
  std::cout << "\n";
        :
```

このようにイテレータとは、さまざまな繰り返しループに必要なポインタ等を「抽象化」したもの。

STL コンテナ：ソートされた連想コンテナ map

oopgp/container05.cc:

```
        :
#include <map>
        :
using std::string;
struct hostinfo { std::vector<string> addrs, addr6s; };
        :
  const char *hosts[] = {
    "www.freebsd.org",
        :
    "localhost",
  };    // 以上のホスト hosts[i] の IPv4 アドレスリスト、IPv6 アドレスリストを調べて、
        // hostinfos[hosts[i]].addrs, hostinfos[hosts[i]].addr6s に記憶していく

  std::map<string, struct hostinfo> hostinfos;  // <string, struct hostinfo>型の map
  struct addrinfo hints, *res, *r;
  memset(&hints, 0, sizeof(struct addrinfo));
  hints.ai_family = AF_UNSPEC;
  hints.ai_socktype = SOCK_STREAM;
  int gai_errno;
  for (size_t i=0; i<sizeof(hosts)/sizeof(*hosts); i++) {
    if ((gai_errno=getaddrinfo(hosts[i], NULL, &hints, &res)) != 0) continue;
    std::cout << hosts[i] << ": ";
    for (r=res; r!=NULL; r=r->ai_next) {
      char host[NI_MAXHOST];
      if ((r->ai_family != AF_INET && r->ai_family != AF_INET6) ||
          (gai_errno=getnameinfo(r->ai_addr, r->ai_addrlen, host, sizeof(host), NULL, 0, NI_NUMERICHOST)) != 0) continue;
      std::cout << host << ", ";                        // getaddrinfo で得たアドレスリストを getnameinfo で文字列表現にて得て、
      std::vector<string> &a = (r->ai_family == AF_INET6) ?
        hostinfos[hosts[i]].addr6s : hostinfos[hosts[i]].addrs;
      a.push_back(host);        // char host[] を hostinfos[hosts[i]].addrs もしくは hostinfos[hosts[i]].addr6s の後ろに格納
    }
    std::cout << "\n";                                  // hostinfos の [] の中が文字列であることに注目！
  }
        :
```

：ソートされた連想コンテナ map（つづき）

oopgp/container05.cc:

```
          :
    std::cout << "\nhostinfos for IPv4:\n";
    for (size_t i=0; i<sizeof(hosts)/sizeof(*hosts); i++) {
      if (hostinfos[hosts[i]].addrs.empty())              // hostinfos[hosts[i]].addrs が空じゃなければ
        continue;
      std::cout << hosts[i] << ": ";
      for (std::vector<string>::iterator it = hostinfos[hosts[i]].addrs.begin();
           it != hostinfos[hosts[i]].addrs.end(); ++it)   // hostinfos の [] の中が文字列であることに注目！
        std::cout << *it << ", ";
      std::cout << "\n";
    }
    std::cout << "\nhostinfos for IPv6:\n";
    for (size_t i=0; i<sizeof(hosts)/sizeof(*hosts); i++) {
      if (hostinfos[hosts[i]].addr6s.empty())             // hostinfos[hosts[i]].addr6s が空じゃなければ
        continue;
      std::cout << hosts[i] << ": ";
      for (std::vector<string>::iterator it = hostinfos[hosts[i]].addr6s.begin();
           it != hostinfos[hosts[i]].addr6s.end(); ++it)  // hostinfos の [] の中が文字列であることに注目！
        std::cout << *it << ", ";
      std::cout << "\n";
    }
          :
```

このように C++だと、なんと distionay["キー"] = "バリュー" のようなコンテナまで出来てしまう。
但し、std:map は重複するキーには対応していない（後述）。

：ソートされた連想コンテナ map（つづき）

oopgp/container05.cc:

```
$ make container05
$ ./container05
        :
hostinfos for IPv4:
www.freebsd.org:        69.147.83.33,
www.netbsd.org:         204.152.190.12,
www.openbsd.org:        129.128.5.191,
www.debian.org:         194.109.137.218,
www.ubuntu.com:         91.189.94.8,
www.fedoraproject.org:  209.132.176.122, 66.35.62.162, 80.239.156.214, 152.46.7.221,
fedoraproject.org:      209.132.176.122, 66.35.62.162, 80.239.156.214, 152.46.7.221,
www.linux.org:          198.182.196.56,
www.opensolaris.org:    72.5.123.5,
www.macosforge.org:     17.254.17.248,
www.microsoft.com:      207.46.193.254, 207.46.192.254, 65.55.12.249,
www.tron.org:           202.32.0.94,
code.google.com:        209.85.143.99, 209.85.143.104, 209.85.143.147,
localhost:              127.0.0.1,

hostinfos for IPv6:
localhost:              ::1,
```

コンテナ map **は、**Perl **のようなスクリプト言語の「連想配列」と考えればよい。模式的に書けば、以下のようなデータ構造となっている。**

```
hostinfos['www.freebsd.org'] = { .addrs = { 69.147.83.33, }, .addr6s = {} }
        :
hostinfos['code.google.com'] = { .addrs = { 209.85.143.99, 209.85.143.104, 209.85.143.147, }, .addr6s = {} }
hostinfos['localhost']       = { .addrs = { 127.0.0.1, }, .addr6s = { ::1, } }
        :
```

STL アルゴリズム：ソート sort

oopgp/algorithm00.cc:

```
$ diff -u container00.cc algorithm00.cc
        :
+#include <algorithm>
+#include <functional>  // greater, etc
        :
   std::vector<string> full_names;
   for (int i=0; i<int(sizeof(family_names)/sizeof(string)); i++)
     for (int j=0; j<int(sizeof(names)/sizeof(string)); j++)
       full_names.push_back(family_names[i] + " " + names[j]);
        :
+  //sort(full_names.begin(), full_names.end());                    // 昇順にソートする場合
+  using std::greater;
+  sort(full_names.begin(), full_names.end(), greater<string>());   // 降順にソートする場合
+
        :

$ ./algorithm00
names: Taro, Hanako, Jiro, Keiko,
family names: Kato, Yamada, Ikeguchi,

full names: Kato Taro, Kato Hanako, Kato Jiro, Kato Keiko, Yamada Taro, Yamada Hanako, Yamada Jiro, ...

full names: Yamada Taro, Yamada Keiko, Yamada Jiro, Yamada Hanako, Kato Taro, Kato Keiko, Kato Jiro, ...
```

このような汎用アルゴリズムが他にもたくさん用意されている。並び替えが生じない for_each,find,search **系、並び替えが生じる** remove,unique,rotate **系、ソート関連** sort,binary_search,lower_bound,upper_bound **系などに大別される。**

STL アルゴリズム：述語検索

oopgp/algorithm01.cc:

```
$ diff -u container00.cc algorithm01.cc
+using std::string;
+bool full_names_many_a(const string &full_name)
+{
+  return (count(full_name.begin(), full_name.end(), 'a') > 2);
+} // この count も汎用アルゴリズムのひとつ：string full_name で 'a' である要素の数
         :
   std::vector<string> full_names;
   for (int i=0; i<int(sizeof(family_names)/sizeof(string)); i++)
     for (int j=0; j<int(sizeof(names)/sizeof(string)); j++)
       full_names.push_back(family_names[i] + " " + names[j]);
         :
+  std::cout << "\nfull names(many a): ";
+  for (std::vector<string>::iterator it =
+         find_if(full_names.begin(), full_names.end(), full_names_many_a);
+        it != full_names.end();
+        it = find_if(++it, full_names.end(), full_names_many_a))
+    std::cout << *it << ", ";
   std::cout << "\n";   // find_if で述語 full_names_many_a が真を返すような要素を、次々と探している
         :

$ ./algorithm01
names: Taro, Hanako, Jiro, Keiko,
family names: Kato, Yamada, Ikeguchi,

full names: Kato Taro, Kato Hanako, Kato Jiro, Kato Keiko, Yamada Taro, Yamada Hanako, Yamada Jiro, ...

full names(many a): Kato Hanako, Yamada Taro, Yamada Hanako, Yamada Jiro, Yamada Keiko,
```

このようなイテレータを返す汎用アルゴリズム find_if **でループを構成できる。また、** find_if **に述語として関数ポインタを指定していることにも注目。**

STL アルゴリズム：述語検索（別解）

oopgp/algorithm02.cc:

```
$ diff -u algorithm01.cc algorithm02.cc
        :
-bool full_names_many_a(const string &full_name)
-{
-  return (count(full_name.begin(), full_name.end(), 'a') > 2);
-}
+struct full_names_many_a {
+  bool operator()(const string &full_name) const
+  {
+    return (count(full_name.begin(), full_name.end(), 'a') > 2);
+  }
+};
        :
   std::cout << "\nfull names(many a): ";
   for (std::vector<string>::iterator it =
-        find_if(full_names.begin(), full_names.end(), full_names_many_a);
+        find_if(full_names.begin(), full_names.end(), full_names_many_a());
        it != full_names.end();
-       it = find_if(++it, full_names.end(), full_names_many_a))
+       it = find_if(++it, full_names.end(), full_names_many_a()))
     std::cout << *it << ", ";
        :
$ ./algorithm01
names: Taro, Hanako, Jiro, Keiko,
family names: Kato, Yamada, Ikeguchi,

full names: Kato Taro, Kato Hanako, Kato Jiro, Kato Keiko, Yamada Taro, Yamada Hanako, Yamada Jiro, ...

full names(many a): Kato Hanako, Yamada Taro, Yamada Hanako, Yamada Jiro, Yamada Keiko,
```

このように、述語に先の関数ポインタではなく、クラスのオペレータ () のメンバ関数でも同様に実現できる。こちらの方が柔軟性がある（後述）。

STL アルゴリズム：すべての要素への関数の適用

oopgp/algorithm04.cc:

```
$ diff -u container00.cc algorithm04.cc
        :
+#include <algorithm>

+using std::string;
+void std_cout_string_comma(string s)
+{
+  std::cout << s << ", ";
+}

 int main()
 {
        :
   std::cout << "\nfull names: ";
-  for (std::vector<string>::iterator it = full_names.begin();
-       it != full_names.end(); ++it)
-    std::cout << *it << ", ";

+  for_each(full_names.begin(), full_names.end(), std_cout_string_comma);

   std::cout << "\n";
        :
   return 0;
 }
```

このように、すべての要素に適用すべき手続きを `for_each` **で関数を指定することで一行で書けてしまう。**

STL アルゴリズム：分割、巡回

oopgp/algorithm06.cc:

```
$ diff -u algorithm01.cc algorithm06.cc
        :
+void std_cout_string_comma(const string &s)
+{
+  std::cout << s << ", ";
+}
        :
+  for_each(full_names.begin(), full_names.end(), std_cout_string_comma);
   std::cout << "\n";

+  std::vector<string>::iterator sep = partition(full_names.begin(), full_names.end(), full_names_many_a);

   std::cout << "\nfull names(many a): ";
+  for_each(full_names.begin(), sep, std_cout_string_comma);
+  std::cout << "\n";
+  std::cout << "\nfull names(others): ";
+  for_each(sep, full_names.end(), std_cout_string_comma);
+  std::cout << "\n";
        :
+  rotate(full_names.begin(), sep, full_names.end());
+
+  std::cout << "\nfull names(rotated): ";
+  for_each(full_names.begin(), full_names.end(), std_cout_string_comma);
+  std::cout << "\n";
        :
```

`partition` で述語 `full_names_many_a` が真を返すような要素を先頭へ詰め直し、偽を返すような要素の先頭を `sep` に返している。よって、上例の `rotate` では `sep` が先頭となるような巡回が行なわれることになる。特に意味がある例題ではない。

STL アルゴリズム：関数オブジェクト

oopgp/algorithm06.cc:

```
$ diff -u algorithm01.cc algorithm06.cc
        :
+struct full_names_change {
+  string f, t;
+  void operator()(string &full_name) const
+  {
+    string::iterator it;
+    if ((it = search(full_name.begin(), full_name.end(), f.begin(), f.end())) !=
+       full_name.end()) {
+      full_name.erase(it, it+f.length());
+      full_name.insert(it, t.begin(), t.end());
+    }
+  }
+};
        :
+  struct full_names_change change;
+  change.f = "Taro";
+  change.t = "George";
+  for_each(full_names.begin(), full_names.end(), change);
+
+  std::cout << "\nfull names(changed): ";
+  for_each(full_names.begin(), full_names.end(), std_cout_string_comma);
   std::cout << "\n";
        :
```

クラスのオペレータ `()` **のメンバ関数で、しかもそのインスタンス** `change` **による関数オブジェクトを** `for_each` **に指定している。ここで** `change()` **ではないことに注意。そのメンバ変数が意図したように作用していることから、この方法が関数ポインタよりも柔軟性があることがわかる。**

ここで、`"Taro"`**を** `search` **で探して見つかれば、それを** `erase` **して、そこに**`"George"`**を挿入しているわけだが、**`search` **は** `string.h` **の** `strstr(3)` **の** C++**汎用アルゴリズム版とでもいうべきものである。**

（例題２）関数オブジェクト

先の oopgp/algorithm06.cc では、まだ full_names_many_c が関数オブジェクトになっておらず、柔軟性がまだ不十分である。クラス full_names_change を参考に、以下のように partition を呼べるように oopgp/algorithm06.cc を改良せよ。

```
          :
    struct full_names_many_c many_o;
    many_o.c = 'o';
    many_o.n = 1;
    std::vector<string>::iterator sep = partition(full_names.begin(), full_names.end(), many_o);
          :

  $ ./algorithm06-2
  names: Taro, Hanako, Jiro, Keiko,
  family names: Kato, Yamada, Ikeguchi,

  full names: Kato Taro, Kato Hanako, Kato Jiro, Kato Keiko, Yamada Taro, Yamada Hanako, Yamada Jiro, ...

  full names(many o): Kato Taro, Kato Hanako, Kato Jiro, Kato Keiko,

  full names(others): Yamada Taro, Yamada Hanako, Yamada Jiro, Yamada Keiko, Ikeguchi Taro, ...
          :
```

（ヒント）どうしても判らなければ oopgp/algorithm06-2.cc を見てもよい。

STL アルゴリズム：特殊な型のソートと検索

oopgp/algorithm07.cc:

```
#include <iostream>     // cout, etc
#include <vector>
#include <algorithm>
#include <string.h>     // memcmp, etc
#include <netinet/in.h> // struct in_addr, etc
#include <arpa/inet.h>  // inet_ntop, etc

// struct in_addr 型の vector のソートに使う
bool in_addr_less(const struct in_addr a, const struct in_addr b)
{
  return ntohl(a.s_addr) < ntohl(b.s_addr);
}

// struct in_addr 型の vector の検索に使う
bool operator<(const struct in_addr a, const struct in_addr b)
{
  return ntohl(a.s_addr) < ntohl(b.s_addr);
}
        :
int main(int argc, char *argv[])
{
  const char *addrs[] = {
    "60.32.126.145",
        :
    "172.16.127.1",
  };
  std::vector<struct in_addr> in_addr_list;                        // struct in_addr 型の vector

  for (size_t i=0; i<sizeof(addrs)/sizeof(char *); i++) {
    struct in_addr addr;
    if (inet_pton(AF_INET, addrs[i], &addr) != 1)                  // inet_pton で変換できたら
      continue;
    in_addr_list.push_back(addr);                                  // addr を in_addr_list の後ろに格納
  }
        :
```

：特殊な型のソートと検索（つづき）

oopgp/algorithm07.cc:

```
        :
    for (int a=1; a<argc; a++) {                              // コマンドライン引数についても同様に行なう
      struct in_addr addr;
      if (inet_pton(AF_INET, argv[a], &addr) != 1)
        continue;
      in_addr_list.push_back(addr);                           // addr を in_addr_list の後ろに格納
    }

    sort(in_addr_list.begin(), in_addr_list.end(), in_addr_less); // in_addr_less でソート

    std::cout << "in_addr list:\n";                           // in_addr_list を出力
    for (std::vector<struct in_addr>::iterator it = in_addr_list.begin();
         it != in_addr_list.end(); ++it) {                    // struct in_addr 型の vector のイテレータ
      char b[64];
      std::cout << inet_ntop(AF_INET, &*it, b, sizeof(b)) << "\n";     // ここで「&*」は必要
    }

    std::cout << "\nbinary_search in_addr list:\n";
    std::string taddrs = "192.168.3.1";                       // 検索対象
    struct in_addr addr;
    inet_pton(AF_INET, taddrs.c_str(), &addr);                // c_str() for C
    if (binary_search(in_addr_list.begin(), in_addr_list.end(), addr))    // addr を検索
      std::cout << taddrs + ": found\n";
    else
      std::cout << taddrs + ": not found\n";

    return 0;
  }
```

このように、オペレータ「<」を多重定義することにより、特別な型の検索も可能となる。この場合、二分探索を用いているので、予めソートしておく必要がある。その際、特別な型であるので、関数ポインタ、もしくは、関数オブジェクトを指定することになる。

（例題３）重複するキーの検索

先の、ホスト名からアドレスリストへの map を作成する oopgp/container05.cc をベースに、加えて、重複し得るアドレスからホスト名への multimap を作成せよ。さらに、重複し得るキーについての検索を行なえるようにせよ。出力例は以下のようになる。

```
$ ./algorithm08 152.46.7.221
        :       # ← ここは container05 の出力と同じ
in_addr map for IPv4:
17.254.17.248   = www.macosforge.org
65.55.12.249    = www.microsoft.com
66.35.62.162    = www.fedoraproject.org
66.35.62.162    = fedoraproject.org
69.147.83.33    = www.freebsd.org
72.5.123.5      = www.opensolaris.org
80.239.156.214  = www.fedoraproject.org
80.239.156.214  = fedoraproject.org
        :       # ↑ ここがソートされていることに注目

find in_addr map for IPv4:
152.46.7.221: found     = www.fedoraproject.org
66.35.62.162, 80.239.156.214, 152.46.7.221, 209.132.176.122,
152.46.7.221: found     = fedoraproject.org
66.35.62.162, 80.239.156.214, 152.46.7.221, 209.132.176.122,

hostinfos for IPv6:
        :       # ← ここは container05 の出力と同じ

in6_addr map for IPv6:
::1     = localhost

find in6_addr map for IPv6:
::1: found      = localhost
::1,
```

（例題３）の解答例

oopgp/algorithm08.cc:

```
$ diff -u container05.cc algorithm08.cc
        :
+// in_addr 型の multimap の内部的なソートに使う
+bool operator<(const struct in_addr a, const struct in_addr b)
+{
+  return ntohl(a.s_addr) < ntohl(b.s_addr);
+}

+// in6_addr 型の multimap の内部的なソートに使う
+bool operator<(const struct in6_addr a, const struct in6_addr b)
+{
+  return (memcmp(&a.s6_addr, &b.s6_addr, sizeof(a.s6_addr)) < 0);
+}
        :
   std::map<string, struct hostinfo> hostinfos;
+  std::multimap<struct in_addr, string> in_addr_map;   // in_addr からホスト名への multimap in_addr_map
+  std::multimap<struct in6_addr, string> in6_addr_map; // in6_addr からホスト名への multimap in6_addr_map
   struct addrinfo hints, *res, *r;
   memset(&hints, 0, sizeof(struct addrinfo));
   hints.ai_family = AF_UNSPEC;
        :
       // getaddrinfo で得たアドレスリストを getnameinfo で文字列表現にて得て、
       std::vector<string> &a = (r->ai_family == AF_INET6) ?
        hostinfos[hosts[i]].addr6s : hostinfos[hosts[i]].addrs;
       a.push_back(host);
       // char host[] を hostinfos[hosts[i]].addrs もしくは hostinfos[hosts[i]].addr6s の後ろに格納
       // したところで…
        :
```

（例題３）の解答例（つづき）

oopgp/algorithm08.cc:

```
        :
+      switch (r->ai_family) {
+      case AF_INET:
+       in_addr_map.insert(                      // in_addr_map へ pair<struct in_addr, string> を挿入
+          std::pair<struct in_addr, string>(
+           ((struct sockaddr_in *)r->ai_addr)->sin_addr, string(hosts[i])
+                                          )
+                            );
+       break;
+      case AF_INET6:
+       in6_addr_map.insert(                     // in6_addr_map へ pair<struct in6_addr, string> を挿入
+          std::pair<struct in6_addr, string>(
+           ((struct sockaddr_in6 *)r->ai_addr)->sin6_addr, string(hosts[i])
+                                           )
+                             );
+       break;
+      }
    }
    std::cout << "\n";
  }
        :
```

（例題３）の解答例（つづき）

oopgp/algorithm08.cc:

```
          :
  +  std::cout << "\nin_addr map for IPv4:\n";           // in_addr map をリストアップ
  +  for (std::multimap<struct in_addr, string>::iterator it =
  +          in_addr_map.begin(); it != in_addr_map.end(); ++it) {
  +    char b[NI_MAXHOST];
  +    std::cout << inet_ntop(AF_INET, &it->first, b, sizeof(b)) <<
  +      "\t= " << it->second << "\n";           // stp::multimap の要素は std::pair であり、その要素は first, second で取得
  +  }
  +
  +  std::cout << "\nfind in_addr map for IPv4:\n";
  +  std::string taddrs = (1 < argc) ? argv[1] : "127.0.0.1"; // taddrs を検索
  +  struct in_addr addr;
  +  inet_pton(AF_INET, taddrs.c_str(), &addr);   // c_str() for C
  +  for (std::multimap<struct in_addr, string>::iterator
  +          tlb = in_addr_map.lower_bound(addr),
  +          tub = in_addr_map.upper_bound(addr),
  +          tit = tlb; tit != tub; ++tit) { // 重複しうる検索の場合 tit=[lower_bound(addr),upper_bound(addr)) が検索結果の範囲となる
  +    std::cout << taddrs + ": found\t= " << tit->second << "\n";
  +    for (std::vector<string>::iterator it = hostinfos[tit->second].addrs.begin();
  +          it != hostinfos[tit->second].addrs.end(); ++it) {
  +      std::cout << *it << ", ";                 // 検索されたアドレスのホスト名から hostinfos に格納されているアドレスリストを表示
  +    }
  +    std::cout << "\n";
  +  }
          : // 以下、IPv6 についても、同様のリストアップと検索
```

このように、multimap(当然のことながら map も) はいつでもソートされていることに注目せよ。よって、いつでも検索可能である。

namespace, typedef

先の例のように、様々な型が絡み合って型指定が冗長になっていく。グローバルスコープの using namespace **や** typedef **をスコープ内で宣言することにより、型の指定子が簡潔になる。**

oopgp/algorithm09.cc:

```
$ diff -u algorithm08.cc algorithm09.cc
        :
+using namespace std;           // std:: が省略できるようになる

-using std::string;
-   std::vector<string> addrs, addr6s;
+   vector<string> addrs, addr6s;
        :

-   std::map<string, struct hostinfo> hostinfos;
-   std::multimap<struct in_addr, string> in_addr_map;
-   std::multimap<struct in6_addr, string> in6_addr_map;
+   map<string, struct hostinfo> hostinfos;
+   multimap<struct in_addr, string> in_addr_map;
+   multimap<struct in6_addr, string> in6_addr_map;

+   typedef multimap<struct in_addr, string>::iterator in_addr_map_it;   // ← 注目！
+   typedef multimap<struct in6_addr, string>::iterator in6_addr_map_it; // ← 注目！
        :
-   std::cout << "\nin_addr map for IPv4:\n";
-   for (std::multimap<struct in_addr, string>::iterator it =
+   cout << "\nin_addr map for IPv4:\n";
+   for (in_addr_map_it it =
            in_addr_map.begin(); it != in_addr_map.end(); ++it) {
       char b[NI_MAXHOST];
-      std::cout << inet_ntop(AF_INET, &it->first, b, sizeof(b)) <<
-        "\t= " << it->second << "\n";     /* first and second in std::pair */
+      cout << inet_ntop(AF_INET, &it->first, b, sizeof(b)) <<
+        "\t= " << it->second << "\n";     /* first and second in pair */
    }
        :
```

boost::format 書式付き型安全出力

std::cout <<による標準出力は printf と比べて「型安全」だが「冗長」である。Boost の format クラスを用いることで、型安全かつ簡潔に出力できるようになる。
oopgp/ex10.cc:

```
$ diff -u ex00.cc ex10.cc
        :
+#include <boost/format.hpp>
        :
+using namespace boost;
        :
+   std::cout << format("%d\n") % B::c;
        :
+   std::cout << format("B X. %d %g %d %g %s %g\n") % X.a % X.b[0] % X.c % X[2] % X.get_e() % X.ref_f();
+   std::cout << format("B Y. %d %g %d %g %s %g\n") % Y.a % Y.b[0] % Y.c % Y[2] % Y.get_e() % Y.ref_f();
+   std::cout << format("B Z. %d %g %d %g %s %g\n") % Z.a % Z.b[0] % Z.c % Z[2] % Z.get_e() % Z.ref_f();
        :
+   std::cout << format("D W. %d %g %d %g %s %g %g %g\n") % W.a  % W.b[0]  % W.c  % W[2]    % W.get_e()  % W.ref_f()  % ...
+   std::cout << format("B V->%d %g %d %g %s %g %g %g\n") % V->a % V->b[0] % V->c % (*V)[2] % V->get_e() % V->ref_f() % ...
        :
+   std::cout << format("D W. %d %g %d %g %s %g %g %g\n") % W.a  % W.b[0]  % W.c  % W[2]    % W.get_e()  % W.ref_f()  % ...
+   std::cout << format("B V->%d %g %d %g %s %g %g %g\n") % V->a % V->b[0] % V->c % (*V)[2] % V->get_e() % V->ref_f() % ...
        :
```

演算子「%」が引数の区切りとして使われていることに注目。

boost::format（つづき）

そもそも「型安全」であるので、すべて char * **型でも全く問題なく、文字列型に自動的に変換される。**

oopgp/ex20.cc:

```
$ diff -u ex00.cc ex20.cc
        :
+#include <boost/format.hpp>
        :
+using namespace boost;
        :
+  std::cout << format("%s\n") % B::c;
        :
+  std::cout << format("B X. %s %s %s %s %s %s\n") % X.a % X.b[0] % X.c % X[2] % X.get_e() % X.ref_f();
+  std::cout << format("B Y. %s %s %s %s %s %s\n") % Y.a % Y.b[0] % Y.c % Y[2] % Y.get_e() % Y.ref_f();
+  std::cout << format("B Z. %s %s %s %s %s %s\n") % Z.a % Z.b[0] % Z.c % Z[2] % Z.get_e() % Z.ref_f();
        :
+  std::cout << format("D W. %s %s %s %s %s %s %s %s\n") % W.a  % W.b[0]  % W.c  % W[2]    % W.get_e()  % W.ref_f()  % ...
+  std::cout << format("B V->%s %s %s %s %s %s %s %s\n") % V->a % V->b[0] % V->c % (*V)[2] % V->get_e() % V->ref_f() % ...
        :
+  std::cout << format("D W. %s %s %s %s %s %s %s %s\n") % W.a  % W.b[0]  % W.c  % W[2]    % W.get_e()  % W.ref_f()  % ...
+  std::cout << format("B V->%s %s %s %s %s %s %s %s\n") % V->a % V->b[0] % V->c % (*V)[2] % V->get_e() % V->ref_f() % ...
        :
```

スマートポインタ：自動（削除）ポインタ

oopgp/smartprt00.cc:

```
        :
#include <memory>       // auto_ptr

        :
class B {
public:
  B(std::string label) : label(label)   { std::cout << "created:\t"   << label << "\n"; }
  virtual ~B()                          { std::cout << "destroyed:\t" << label << "\n"; }
  void knock() const                    { std::cout << "knocked:\t"   << label << "\n"; }
private:
  std::string label;
};
        :
  {
    std::cout << "\nscope #2\n";
    const B *const Y = new B("Y");
    Y->knock();
    /* (forgot to) delete Y; */
  }                                     // Y は動的記憶域確保された変数なので、delete Y されないとゾンビとなる
  {
    std::cout << "\nscope #3\n";
    const std::auto_ptr<const B> Z(new B("Z"));
    Z->knock();
  }                                     // Z も動的記憶域確保された変数となるが、auto_ptr<> Z の寿命とともに自動的に delete Z してくれる
  {
    std::cout << "\nscope #3\n";
    std::auto_ptr<const B> X(new B("X")), Y(new B("Y")), Z(new B("Z"));
    Y->knock();
    X = Y = Z;
    Y->knock();
  }                                     // しかし、auto_ptr に代入してしまうと、その管理機構が失われ、重複する delete Z で致命的エラー
        :
```

自動的に delete **してくれるので、**auto_ptr **は極めて便利である。但し、上記のように万能ではない。**

スマートポインタ：共有ポインタ Boost

oopgp/smartprt10.cc:

```
        :
#include <boost/shared_ptr.hpp> // shared_ptr
using namespace boost;
        :
class B {
public:
  B(std::string label) : label(label)   { std::cout << "created:\t"   << label << "\n"; }
  virtual ~B()                          { std::cout << "destroyed:\t" << label << "\n"; }
  void knock() const                    { std::cout << "knocked:\t"   << label << "\n"; }
private:
  std::string label;
};
        :
  {
    std::cout << "\nscope #2\n";
    const B *const Y = new B("Y");
    Y->knock();
    /* (forgot to) delete Y; */
  }                                     // Y は動的記憶域確保された変数なので、delete Y されないとゾンビとなる
  {
    std::cout << "\nscope #3\n";
    const shared_ptr<const B> Z(new B("Z"));
    Z->knock();
  }                                     // Z も動的記憶域確保された変数となるが、shared_ptr<> Z の寿命とともに自動的に delete Z される
  {
    std::cout << "\nscope #3\n";
    shared_ptr<const B> X(new B("X")), Y(new B("Y")), Z(new B("Z"));
    Y->knock();
    X = Y = Z;
    Y->knock();
  }                                     // 一方、shared_ptr であれば、それに代入された時点で delete X; delete Y; するので無問題
        :
```

ポインタへの代入にも配慮して自動的に delete してくれるので、shared_ptr もまた極めて便利である。但し、これもやはり万能ではなく「循環参照」までは面倒見切れない、などがある。

スマートポインタ：配列の自動（削除）ポインタ

oopgp/smartprt01.cc:

```
        :
#include <memory>        // auto_ptr

        :
class B {
public:
  B() : label("")                        { std::cout << "created:\n"; }
  B(std::string label) : label(label)    { std::cout << "created:\t"   << label << "\n"; }
  virtual ~B()                           { std::cout << "destroyed:\t" << label << "\n"; }
  void knock() const                     { std::cout << "knocked:\t"   << label << "\n"; }
  void name(std::string label) const     { this->label = label; std::cout << "named:\t" << label << "\n"; }
private:
  mutable std::string label;             // logical const with const member functions
};
        :
  {
    std::cout << "\nscope #3\n";
    std::vector<B> Z(3);                 // std::auto_ptr<B> Z(new B[3]); としてもコンパイルは通ってしまうが…
    Z[0].name("Z0"), Z[1].name("Z1"), Z[2].name("Z2");
    Z[2].knock();
  }                                      // auto_ptr は delete[] には未対応、故に配列には使えない。よってこのように vector<> を使う
  {
    std::cout << "\nscope #3\n";
    std::vector<B> X(3), Y(3), Z(3);
    X[0].name("X0"), X[1].name("X1"), X[2].name("X2");
    Y[0].name("Y0"), Y[1].name("Y1"), Y[2].name("Y2");
    Z[0].name("Z0"), Z[1].name("Z1"), Z[2].name("Z2");
    Y[2].knock();
    X = Y = Z;
    Y[2].knock();
  }                                      // これも問題ないが、そもそも、この代入は意図した動作ではないだろう…
        :
```

配列へのポインタには auto_ptr **は使ってはならない。そもそも配列ではなく** vector **で十分な場合がほとんどのはず。そうでなければ、次の** shared_ptr **を使う。**

スマートポインタ：配列の共有ポインタ Boost

oopgp/smartprt11.cc:

```
        :
#include <boost/shared_array.hpp>       // shared_ptr
using namespace boost;
        :
class B {
public:
  B() : label("")                      { std::cout << "created:\n"; }
  B(std::string label) : label(label)  { std::cout << "created:\t"   << label << "\n"; }
  virtual ~B()                         { std::cout << "destroyed:\t" << label << "\n"; }
  void knock() const                   { std::cout << "knocked:\t"   << label << "\n"; }
  void name(std::string label) const   { this->label = label; std::cout << "named:\t" << label << "\n"; }
private:
  mutable std::string label;           // logical const with const member functions
};
        :
  {
    std::cout << "\nscope #3\n";
    const shared_array<const B> Z(new B[3]);
    Z[0].name("Z0"), Z[1].name("Z1"), Z[2].name("Z2");
    Z[2].knock();
  }                                    // shared_ptr なら配列にも使用可能である。
  {
    std::cout << "\nscope #3\n";
    shared_array<const B> X(new B[3]), Y(new B[3]), Z(new B[3]);
    X[0].name("X0"), X[1].name("X1"), X[2].name("X2");
    Y[0].name("Y0"), Y[1].name("Y1"), Y[2].name("Y2");
    Z[0].name("Z0"), Z[1].name("Z1"), Z[2].name("Z2");
    Y[2].knock();
    X = Y = Z;
    Y[2].knock();
  }                                    // これも問題ない。それに、この代入が意図した動作であろう。
        :
```

配列のポインタにも shared_ptr **は有効である。とは言え、**vector **や** tr1::array **では事足りない場合に** shared_ptr **の使用を検討しよう。**

Boost.Asio: UDP broadcast

Boost にはマルチプラットフォームでネットワーク通信を行なう Asio クラスが提案されている。これを使うと、Unix 系でも Windows でもほぼ同一のコードで、極めて簡単にソケットプログラミングが実現できる。以下のサンプルは WOL: Wake On Lan、つまり MAC アドレスからなるマジックパケットを UDP でブロードキャストするプログラムである。

oopgp/wol.cc:

```
#include <iostream>             // cout, etc
#include <boost/asio.hpp>
#include <libgen.h>             // basename, etc
#include <net/if.h>             // netinet/if_ether.h, etc
#include <netinet/if_ether.h>   // ether_aton, etc
using namespace std;
using namespace boost::asio;

void usage(int argc, char *argv[])
{
  fprintf(stderr, "usage:\n");
  fprintf(stderr, "\t%s [-h] [-s peer] [-p port] ethernet-address\n", basename(argv[0]));
}
size_t make_magic_packet(const char *maddr, char *const buf)
{
  struct ether_addr eaddr, *ea;
  size_t s = 0;

  if ((ea = ether_aton(maddr)))
    eaddr = *ea;
  else if (ether_hostton(maddr, &eaddr) == 0)
    ;
  else {
    cerr << "wol: illegal ethernet address for " << maddr << "\n"; exit(1);
  }
  memset(&buf[0]+s, 0xff, 6); s+=6;
  for (int i=0; i<16; i++) {
    memcpy(&buf[0]+s, eaddr.ether_addr_octet, 6); s+=6;
  }
  return s;
}
        :
```

Boost.Asio: UDP broadcast（つづき）

```
        :
int main(int argc, char *argv[])
{
  char *host = "255.255.255.255", *port = "9", *maddr = NULL;

  for (int a=1; a<argc; a++)
    if (string(argv[a]) == "-h") {
      usage(argc, argv); exit(1);
    }
    else if (string(argv[a]) == "-s" && a+1 < argc)
      host = argv[++a];
    else if (string(argv[a]) == "-p" && a+1 < argc)
      port = argv[++a];
    else
      maddr = argv[a];
  if (!maddr) {
    usage(argc, argv); exit(1);
  }
  try {                          // Boost.Asio を使うと UDP クライアントの実装が、このブロックに書いたコードだけで済んでしまう！
    io_service io_service;
    ip::udp::resolver resolver(io_service);
    ip::udp::resolver::query query(ip::udp::v4(), host, port);
    ip::udp::endpoint peer = *resolver.resolve(query);
    ip::udp::socket socket(io_service);
    socket_base::broadcast option(true);
    socket.open(ip::udp::v4());
    socket.set_option(option);
    char buf[1024];
    size_t s = make_magic_packet(maddr, buf);
    socket.send_to(buffer(buf, s), peer);
  }
  catch (exception &e) {
    cerr << "wol: exception: " << e.what() << "\n";
  }
  return 0;
}
```

Boost uBLAS

oopgp/math00.cc:

```
#include <iostream>
#include <cmath>                // M_PI, etc
#include <boost/numeric/ublas/matrix.hpp>
#include <boost/numeric/ublas/vector.hpp>
#include <boost/numeric/ublas/io.hpp>
        :
namespace la = boost::numeric::ublas;
        :
  const int n_div = 1000000;
  double a = (360./n_div)*M_PI/180, c = cos(a), s = sin(a);
  double m[3][3] = {
    {  1,  0,  0 },
    {  0,  c, -s },
    {  0,  s,  c },
  };
  double v[3] = { 1, 2, 3 }, v0[3], v1[3];
        :
  std::cout << "******** uBLAS matrix mult. matrix in C++ Boost\n";
  la::matrix<double> M(3, 3);
  la::vector<double> V(3);
  for (int i=0; i<3; i++)
    for (int j=0; j<3; j++)
      M(i, j) = m[i][j];
  for (int i=0; i<3; i++)
    V[i] = v[i];
  std::cout << V << "\n";

  for (int i=0; i<n_div; i++) { // ベクトル V=行列 M ×ベクトル V を n_div 回計算
    V = prod(M, V);
  }
  std::cout << V << "\n";
        :
```

Boost には、Fortran で書かれた著名な BLAS: Basic Linear Algebra Subprograms を C++テンプレートへ移植した uBLAS が搭載されている。これを使えば、主な線形代数の数値計算は揃ってしまうだろう。

Boost math::quaternion

oopgp/math00.cc:

```
#include <iostream>
#include <cmath>
        :
#include <boost/math/quaternion.hpp>
        :
  std::cout << "******** quaternion rot. vector in C++ Boost\n";
  boost::math::quaternion<double> P(0, v[0], v[1], v[2]), Q(q[3], q[0], q[1], q[2]);
  std::cout << P.unreal() << "\n";
  for (int i=0; i<n_div; i++) { // 四元数 P=四元数 Q•P•Q^{-1}を n_div 回計算
    P = Q * P / Q;
  }
  std::cout << P.unreal() << "\n";
        :
```

Boost には、CG の分野などで重宝されている複素数の拡張である四元数クラス `math::quaternion` 及び八元数クラス `math::octonion` が搭載されている。これを使えば、基本的な四元代数、八元代数の数値計算が簡単に出来てしまう。

オブジェクト指向プログラミング：まとめ

クラス付き C**、標準テンプレートライブラリ** (STL)**、**Boost **の概観を通して：**

- ▷ **クラスと継承、名前空間、及び、**C++**での様々な注意点**
- ▷ STL: **コンテナとイテレータ**
- ▷ STL: **汎用アルゴリズム** (**ソート、検索など**)
- ▷ STL: **関数オブジェクト**
- ▷ C++, Boost: **スマートポインタ** (`auto_ptr`, `shared_ptr`)
- ▷ Boost: format **による書式付き型安全出力**
- ▷ Boost: Boost.Asio **による** UDP **パケット送信**
- ▷ Boost: uBLAS **による線形代数クラスの利用**
- ▷ Boost: quaternion **クラスによる四元代数クラスの利用**

その他の重要な話題：

- ▷ **例外処理**
- ▷ STL: **関数アダプタ、テンプレートによる関数オブジェクト**
- ▷ **コンテナと汎用アルゴリズムの計算量の詳細**
- ▷ TR1: **正規表現、タプル、**`array`**、**`traits`**、**`result_of`**、乱数、数学関数**
- ▷ Boost: **プログラムオプション、文字列アルゴリズム、ハッシュ、マルチ配列、無名関数、メタプログラミングライブラリ、**MPI(Message Passing Interface)**、有理数、区間演算、特殊関数、統計分布、等**
- ▷ TMP: **テンプレートメタプログラミングの実践**

冒頭にあげた以外の参考文献：

- ▷ ‘Boost C++ Library,’’ http://www.boost.org/ **公式のドキュメントが正確である。なるべく一次情報に近いところで学ぶべきである。**
- ▷ ‘C++0x **最終ドラフト**,’’ http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2008/n2798.pdf