Programming C++
Lecture Note 7
連想コンテナを使う

Jie Huang

記憶領域の動的割付と解放

- C++では、以下のように、実行中にnew演算子によって、動的に領域を割り当てることができる。

```
int *p = new int;
railroad_car *pt = new railroad_car;
```

- 配列については、以下のように割り当てる。

```
char *name = new char[20];
```

- new演算子によって生成されるオブジェクトは必要なくなった時、プログラマの責任によって、解放しなくてはならない。解放は以下のdelete演算子を使う。

```
delete p;
delete pt;
```

- 解放されるオブジェクトが配列の場合は、以下のように指定する。

```
delete [] name;
```

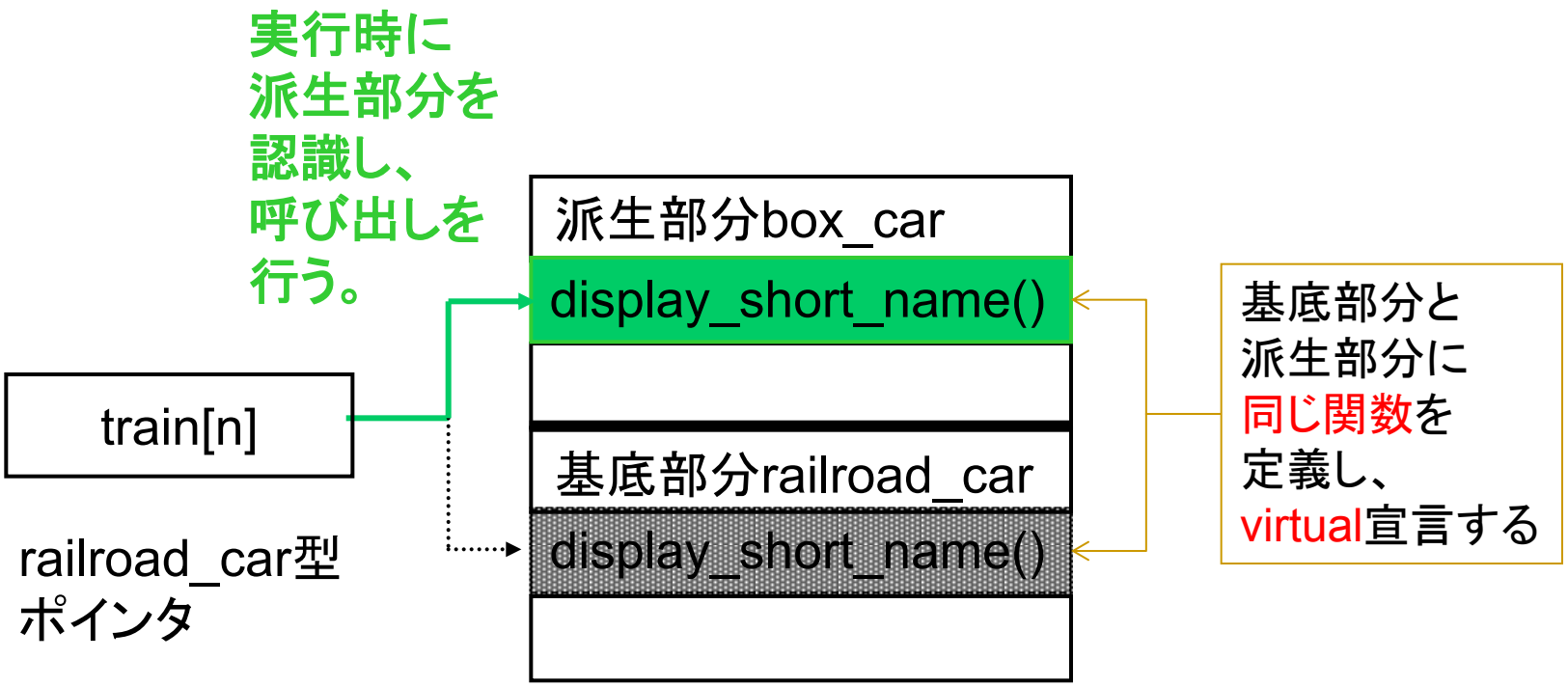
派生クラスから基底クラスへのポインタの型変換

- クラスの階層構造を使ってオブジェクトを定義する場合、派生クラスオブジェクトを基底クラスのポインタで統一的に扱いたいことがよくある。
  例えば:

```
railroad_car *train[100];
train[0] = new box_car;
train[1] = new tank_car;
```

- この場合、派生クラス型から基底クラス型へのポインタの代入はデフォルトで可能である。
  注）反対に基底クラスへのポインタを派生クラスのポインタに代入する場合は情報が欠如し、デフォルトの処理ができない。
- しかし、基底クラス型のポインタからは派生クラスの派生部分が見えなくなる。この例では、box_carとtank_carの拡張された部分はtrain[0]とtrain[1]ポインタからは見えないので、それに対するアクセスはできない。この見えない部分へのアクセスを可能にするのが、仮想関数である。

仮想関数(virtual function)



仮想関数(virtual function)

- 仮想関数の定義

```
class railroad_car {
  public: railroad_car () {}
    virtual void display_short_name(){cout<<"rrc";}
};
class box_car: public railroad_car {
  public: box_car () {}
    virtual void display_short_name(){cout<<"box";}
};
class tank_car: public railroad_car {
  public: tank_car () {}
    virtual void display_short_name(){cout<<"tnk";}
};
class engine: public railroad_car {
  public: engine () {}
    virtual void display_short_name(){cout<<"eng";}
};
class caboose: public railroad_car {
  public: caboose () {}
    virtual void display_short_name(){cout<<"cab";}
};
```

仮想関数(virtual function)

- 仮想関数の役割

```
railroad_car *train[100];
main(){
  int n, car_count, type_code;
  for(car_count = 0; cin >> type_code; ++car_count)
    if( type_code == 0) train[car_count] = new engine;
    else if( type_code == 1) train[car_count] = new box_car;
    else if( type_code == 2) train[car_count] = new tank_car;
    else if( type_code == 3) train[car_count] = new caboose;
  for( n = 0; n < car_count; ++n){
    // ポインタの指す実体に従って、関数呼び出しをしたい
    train[n]->display_short_name();
    cout << endl;
  }
```

仮想関数(virtual function)

- 基底クラスでvirtual指定される関数は、派生クラスにおいて再定義を行うことができる。再定義された関数は実行時に、基底クラスの仮想関数の代わりに実行される。
- したがって、ポインタの型の変換によって見えなくなった部分の関数も呼び出せるようになる。
- この様に、ポインタで指されるオブジェクトの関数が実行時にその実体によって決定されるので、動的結合という。
- このように同じ関数でも、オブジェクトによって異なる処理をするメカニズムをポリモーフィズム(polymorphism, 多態性)という。

- なお、基底クラスの関数でvirtual宣言されていれば、派生クラス関数のvirtual宣言は省略できる。
- virtual指定は関数の名前、パラメータおよび戻り値(戻り値が異なる場合はエラーとなる)が完全に一致しなければならない。

- また、基底クラスを純粋のインターフェースとして扱う場合は、基底クラスにおいて仮想関数の実体を定義する必要がなく、純粋仮想関数とすることができる。

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

純粋仮想関数と抽象クラス

- C++では実体のない純粋仮想関数(pure virtual function)を含んだクラスを抽象クラス(abstract class)という。抽象クラスは他のクラスの基底クラスになることはできるが、自分自身に属するクラスオブジェクトを作ることができない。

```
class classA {
public:
  virtual void a1() = 0; // pure virtual function
  virtual void a2() = 0; // pure virtual function
};
class classB: public classA {
public:
  void a1(){…}
};
```

- 純粋仮想関数は全て再定義されなければクラスのオブジェクトを作ることができない。
  クラスclassBでは、純粋仮想関数a1の再定義を行っているが、a2の再定義は行っていないので、同じくオブジェクトを作ることができない。

純粋仮想関数と抽象クラス

- 派生クラスにおいて純粋仮想関数のすべてを再定義すれば、その派生クラスのオブジェクトを作ることができるようになる。
  上の例ではさらに以下のようにclass Cを派生定義すれば、オブジェクトを生成することができるようになる

```
class classC: public classB {
  void a2(){...}
};
classA obja;          // 存在できない
classB objb;          // 存在できない
classC objc;          // 存在できる
```

  抽象クラスは単独では存在しないので、共通の取り扱いをしたい複数のクラスのインタフェースをまとめて定義するためのクラスとして用いることができる。

多重継承と仮想基底クラス

- 以下の例では、classDにはclassBとclassCから受け継いだ2つのclassAが存在する。
- しかし、仮想基底を宣言すれば、2つのclassAは統合し、共通の1つの存在となる。

```
class classA {
public: int a;
};
class classB: public virtual classA {
    ...
};
class classC: public virtual classA {
    ...
};
class classD: public classB, public classC {
    ...
};
```

連想コンテナとは

- 連想コンテナmapの要素はpairクラス(メンバーがfirstとsecondからなる)のオブジェクトで、firstがキーの値、secondが対応する値を保持する。キーに値を対応させる(associate)という意味で、連想コンテナと呼ばれる。例えば、
  map<string, int> counters; // キーがstring, 値がint型のmap
- mapのインデックスにキーを与えるとmapを検索し、同じキーを持つ要素(pairオブジェクト)を探し出し、メンバー変数secondを返す。例えば
  counters[k] = 2;
  では、firstの値がkであるpair要素を探し出し、そのメンバー変数のsecondを返し、それに値2を代入する。
- mapは対応するキーが存在しなければ、自動的にそのエントリーを作る。
- 連想コンテナはキーに基づいて要素を効率的に挿入したり、取り出したりすることができる。
- 連想コンテナのキーはvectorでの要素のインデックスと違って、要素が破棄されない限り、キーの値は変わらない。
- 連想コンテナは自分で順序を作るので、要素の順番をプログラムで変更したりすることができない。

連想コンテナを使って単語を数える

```
int main(){
  string s;
  map<string, int> counters; // キーがstring, 値がintのmap
  while (cin >> s) // 変数sにキーを入力し
    ++counters[s]; // sの値対応するカウンタをインクリメントする
  // イテレータを定義し、mapの最初の要素を指すように初期化する
  map<string,int>::const_iterator it=counters.begin();
  // mapの全ての要素をイテレータを使って出力する
  while( it!=counters.end() ){
    cout << it->first // 要素のキーを出力
         <<"¥t"<<it->second<<endl; // 続けて値を出力
    ++it; // イテレータをインクリメントする
  }
  return 0;
}
```

ワードの出現行番号表を作る

- 関数xrefは入力した文字列から文字列のvectorを作り出し、文字列と出現行数を対応するmapを作る。
- find_worksは文字列探し出す、あるいは分割する関数で、デフォルトではsplit関数を使う。他に同じ型の関数例えばfind_urlsなどが使える。

```
map<string, vector<int> >
xref(istream& in, vector<string>
find_words(const string&) = split){
  string line; int line_number = 0;
  map<string, vector<int> > ret; //stringとint型vectorのmap
  while (getline(in, line)) { //inから文字列を1行入力
    ++line_number; // 行番号をインクリメント
    vector<string> words = find_words(line); //分割又は検索
    // 文字列と対応する行番号vectorのmapを生成する
    for (vector<string>::const_iterator
      it = words.begin(); it != words.end(); ++it)
      ret[*it].push_back(line_number);
  }
  return ret;
}
```

ワードの出現行番号表を作る

- main関数は以下となる

```
int main(){
  map<string, vector<int> > ret = xref(cin); //mapの生成
  //mapの出力
  for (map<string, vector<int> >::
    const_iterator it = ret.begin();
    it != ret.end(); ++it) {
    cout << it->first << “ occurs on line(s): ”;//ワード出力
    vector<int>::const_iterator line_it
      = it->second.begin();
    cout << *line_it; ++line_it; //最初の出現行番号出力
    while (line_it != it->second.end()) {//残りの行番号出力
      cout << ", " << *line_it; ++line_it;
    }
   cout << endl;
  }
  return 0;
}
```

決まられた文法に従った文生成プログラム

- 文生成ルール(文法)

カテゴリ	展開ルール
<noun>	cat
<noun>	dog
<noun>	table
<noun-phrase>	<noun>
<noun-phrase>	<adjective> <noun-phrase>
<adjective>	large
<adjective>	brown
<adjective>	absurd
<verb>	jumps
<verb>	sits
<location>	on the stairs
<location>	under the sky
<location>	wherever it wants
<sentence>	the <noun-phrase> <verb> <location>

決まられた文法に従った文生成プログラム

- 文生成ルールに従って展開例

<sentence>
-> the <noun-phrase> <verb> <location>
-> the <adjective> <noun-phrase> jumps on the stairs
-> the absurd <noun> jumps on the stairs
-> the absurd cat jumps on the stairs

決まられた文法に従った文生成プログラム

- ルール集合を表す3つの型

```
typedef vector<string> Rule; //展開ルールの文字列
typedef vector<Rule> Rule_collection; //ルールの集まり
typedef map<string, Rule_collection> Grammar;//文法
```

- カテゴリとルール集合の対応表Grammarを作る関数

```
Grammar read_grammar(istream& in){
  Grammar ret;  string line;
  while (getline(in, line)) { //1行の文字列を入力
    //split関数によって、カテゴリと続く展開ルールに分ける
    vector<string> entry = split(line);
    if (!entry.empty())
      //対応するカテゴリのルール集に新しいルールを追加する
      ret[entry[0]].push_back(
        Rule(entry.begin() + 1, entry.end()));
  }
  return ret;
}
```

決まられた文法に従った文生成プログラム

- Grammarから文を生成する

```
vector<string> gen_sentence(const Grammar& g){
  vector<string> ret;
  gen_aux(g, “<sentence>”, ret); return ret;
}
void gen_aux(const Grammar& g, const string& word,
  vector<string>& ret){
  if(!bracketed(word)){ ret.push_back(word);}
  else {//括弧付き文字だけを展開する
    Grammar::const_iterator it = g.find(word);//エントリを探す
    if (it == g.end()) throw logic_error(“empty rule”);
    const Rule_collection& c = it->second;//対応するルール集
    const Rule& r=c[nrand(c.size())];//ランダムにルールを決める
    for (Rule::const_iterator i = r.begin();
        i != r.end(); ++i) gen_aux(g, *i, ret);//再帰呼び出し
  }
}
```

決まられた文法に従った文生成プログラム

- 分類を表す文字列の判断

```
bool bracketed(const string& s){
  return s.size() > 1 && s[0] == '<'&& s[s.size()-1] == '>';
}
```

- ランダムに要素を作る

```
int nrand(int n){
  if (n <= 0 || n > RAND_MAX)
    throw domain_error("Argument to nrand is out of range");
  //領域をbucket_sizeに分ける。余った部分は切り捨てる
  const int bucket_size = RAND_MAX / n;
  int r;
  //発生した乱数が余りの部分の場合は再発生を行う
  do r = rand() / bucket_size; while (r >= n);
  return r;
}
```

この関数は乱数をn個の等しい大きさのbucketと半端なサイズの残り値に分割する。半端な値(r >= n)の場合は乱数を作り直す。

決まられた文法に従った文生成プログラム

- 文生成のmain関数

```
int main(){
  //文法を読み込み、文を生成する
  vector<string> sentence=gen_sentence(read_grammar(cin));
  //生成した文の最初のワードの表示
  vector<string>::const_iterator it = sentence.begin();
  if (!sentence.empty()) { cout << *it; ++it;}
  //残りのワードの表示(ワード間はスペース区切り)
  while (it != sentence.end()) {cout << " " << *it; ++it;}
  cout << endl;
  return 0;
}
```