# Programming C++
# Lecture Note 10
# メモリ管理と低レベルのデータ構造

Jie Huang

# ポインタと配列

- ポインタ
  - オブジェクトのアドレスをあらわす。
  - &はアドレス演算子(&xはxのアドレスを表す)
  - *はデリファレンス演算子(*pはポインタpの指すオブジェクトを表す)
- 注意
  - `int *p;` と `int* p;` は等価
  - `int* p, q;` でpはint型オブジェクトへのポインタ、qはint型オブジェクト(ポインタではない)
  - ポインタの値が0(¥0)の場合はどこも指さないnullポインタで、ポインタの有効、無効の判断に使われる

# 配列とポインタの関係

- 配列もコンテナの一種
- 配列の例
  ```
  double coords[3];
  ```
- あるいは
  ```
  const  size_t NDim= 3; // <cstddef>で定義
  double coords[Ndim];
  ```
- ポインタの差は通常以下の型で定義される
  ```
  ptrdiff_t pd; //<cstddef>で定義
  ```
- 配列の名前は配列の最初の要素へのポインタ
  coords[0]と*coordsは等価

# ポインタの算術

- 配列はランダムアクセスコンテナ
- ポインタはランダムアクセス反復子
- p: 配列のm番目の要素を指すポインタとする
  p+nは(m+n)番目の要素を表し
  p−nは(m−n)番目の要素を表す
- 配列はランダムコンテナと同じように、インデックス演算子を使うことができる
  - 配列aのn番目の要素はa[n]
  - p[n] は *(p+n) と等価である
- 配列コンテナから他のコンテナへのコピーの例

```
vector <double> v;
copy(coords, coords+NDim, back_inserter(v));
```

# 配列の初期化

- 以下のように、配列の長さを指定することなく、初期化できる。

```
const int month_lengths[] ={
    31, 28, 31, 30, 31, 30,
    31, 31, 30, 31, 30, 31
};
const char hello[] = {'H', 'e', 'l', 'l', 'o', '¥0'};
```

と

```
const char * const hello = "Hello";
```

は等価である

- 文字列リテラルとstringコンテナ
  文字列リテラルとは従来C言語で使われる¥0で終わる文字型配列
  文字列リテラルを使ったstringの初期化

```
string s("Hello");
string s(hello, hello+strlen(hello));
```

# 配列の初期化

- 成績をABC評価に変換するプログラム例

```
string letter_grade(double grade) {
  static const double numbers[] = {97, 94, 90,
    87, 84, 80, 77, 74, 70, 60, 0
  };
  static const char* letters[] = {“A+”, “A”, “A-”,
    “B+”, “B”, “B-”, “C+”, “C”, “C-”, “D”, “F”
  };
  static const size_t ngrades =
    sizeof(numbers)/sizeof(*numbers);
  for (size_t i = 0; i < ngrades; i++) {
    if (grade >= numbers[i]) return letters[i];
  }
  return “?¥?¥?”; // ?は2つ以上続けて書けないので¥で区切る
}
```

# 関数へのポインタ

- 関数はコピーしたり、代入したり、また引数として関数へ渡すことができないので、代わりに、関数へのポインタが使われる。

```
/*引数と返り値がintである関数へのポインタ*/
int (*fp)(int);
int next(int n){return n+1;}
//このとき下の2つは同じ意味:
fp = &next;
fp = next;
```

- また、以下2つの場合はともに同じ関数の呼び出しとなる

```
i = (*fp)(i);
i = fp(i);
```

# 関数へのポインタの使用例

- 関数へのポインタは他の関数への引数として使われる。
  例えば、ライブラリのfind_if関数

```
template<class In, class Pred>
In find_if(In begin, In end, Pred f){
  while (begin != end && !f(*begin))
     ++begin;
  return begin;
}
```

  この例では、f(*begin)が有効となる関数が必要となる。

# 関数へのポインタを戻す関数

- 関数へのポインタを返す関数の宣言
  ```
  double (*get_analysis_ptr())(const vector<Student_info>&);
  ```
- typdefを使った宣言の仕方
  ```
  typedef double (*analysis_fp)(const vector<Student_info>&);
  analysis_fp get_analysis_ptr();
  ```
  関数get_analysis_ptr()は型analysis_fpの関数へのポインタを返す。

# ファイルの読み書き

- ファイルアクセス用クラスは<fstream>に定義されている
  //ファイル”in”を変数名infileで読み込み用にオープンする
  ifstream infile(“in”);
  //ファイル”out”を変数名outfileで書き込み用にオープンする
  ofstream outfile(“out”);
  ここで、ファイル名の指定は文字列リテラルで、string型ではないので、string sの文字列をファイル名として使う時は以下のように関数c_strを使う必要がある。
  ifstream infile(s.c_str());
- Inという名前のファイルからOutにコピーする例

```
int main() {
  ifstream infile(“in”);
  ofstream outfile(“out”);
  string s;
  while (getline(infile, s)) outfile << s << endl;
  return 0;
}
```

# 標準エラーストリーム

- 通常標準エラー出力は標準出力coutと分かれる
  - cerr：バッファリングしないエラー出力
  - clog：バッファリングを行うエラー出力、
    即出力しない場合もあるので、ログの出力に向いている。

# main関数の引数

- コマンドラインからmain関数への引数

```
int main(int argc, char** argv);
```

ここで、argcは引数の数、argvは引数の配列の最初の要素へのポインタ

- コマンドラインの引数を全部出力するプログラム例

```
int main(int argc, char** argv){
  if (argc > 1) {
  for (i = 1; i < argc-1; ++i)
    cout << argv[i] << " ";
  cout << argv[i] << endl;
  return 0;
}
```

このプログラムをコンパイルして作った実行ファイルをsayとする

```
say Hello, world
```

と入力すると、

```
Hello, world
```

と出力される。

# main関数の引数

- コマンドラインで与えた複数のファイルを全部出力する例

```
int main (int argc, char **argv) {
  int fail_count = 0;
  for (int I = 1; I < argc; i++) {
    ifstream in(argv[i]);
    if (in) {
      string s;
      while (getline(in, s)) cout << s << endl;
    } else {
      cerr << “ファイルを開けません” << argv[i] << endl;
      ++fail_count;
  }
  return fail_count;
}
```

# 3種類のメモリ管理

- 自動メモリ管理
  自動変数のような場合で、自動的にメモリ領域が確保され、また、解放される。
  例えば以下の場合はメモリが解放されるので、戻り値として返す変数へのポインタは無効となる:
  ```
  int* invalid_pointer(){int x; return &x;}
  ```
- 静的なメモリ確保
  ```
  int* pointer_to_static(){
    static int x; return &x;
  }
  ```
- 動的なメモリ確保
  - new, deleteを使う

# newとdeleteを使ったメモリ管理

- newを使ったオブジェクト領域の確保
  ```
  int* p = new int;
  ```
- 初期化を必要とする場合は
  ```
  int* p = new int(42);
  int* pointer_to_dynamic(){
    return new int(0);
  }
  ```
  ここで、組み込み型は1つだけの引数を与えることができる。
  クラス型はコンストラクタのパラメータに合わせて引数を与えることができる
- newにより確保されたメモリ領域の解放
  ```
  delete p;
  ```
  ここで、pが0(nullポインタ)の場合もエラーはなく、単に何もせずに終わる。

# 初期化におけるクラスと非クラスの違い

- Tがクラス型の場合は
  T* p = new T;
  T* p = new T();
  どちらもデフォルトのコンストラクタによる初期化が行われる。
- Tが組み込み型の場合は
  T* p = new T;
  初期化は行われない。
  T* p = new T();
  ゼロで初期化が行われる。

# 配列の動的生成と破棄

- newとdeleteによるオブジェクトの配列の生成と破棄

```
T* p = new T[n];    //生成
delete[] p;         //破棄
```

- newによるオブジェクトの生成は要素数が0の場合もエラーはなく、要素のない配列を作ることができる。この場合のポインタはどこも指さない。
- また、要素のないポインタはdeleteによってエラーなく解放することができる。
- 例えば以下のプログラムは正常に動くことができる。

```
n = 0;
T* p = new T[n];
vector<T> v(p, p+n);
delete [] p;
```

# 配列の動的生成と破棄

- 文字列リテラルをコピーするプログラム例

```
char * duplicate_chars (const char* p) {
  size_t length = strlen(p)+1;
  char* result = new char[length];
  copy(p, p+length, result);
  return result;
}
```