# Boostとその実装技術

## ～Boost の薄い話から濃い話まで～

# 自己紹介

- C++ スキー
- Boost スキー
- D&E 未読
- 猫スキー
- 吉田秀彦モドキ

# アジェンダ

- Boost について
- Boost.SharedPtr
  - Boost.SharedPtr の実装技術
- Boost.Spirit
  - Boost.Spirit の実装技術

# Boost について

- Boost とは

  →Wikipedia「C++の先駆的な開発者のコミュニティ、及びそのコミュニティによって公開されているオープンソースライブラリのこと」

  →便利な C++ のライブラリ

  →Boost の一部が C++0x に入る

# Boost.SharedPtr

# Boost.SharedPtr

- Boost.SharedPtr とは
  インスタンスの所有権を共有したポインタ
  誰も所有しなくなれば自動的に解放される

```
{
    boost::shared_ptr<Hoge> p(new Hoge);
    {
        boost::shared_ptr<Hoge> p2(new Hoge);
        p = p2;
    }
}
```

# Boost.SharedPtr

- インスタンスが所有されている限り解放されない。
  - 誰がインスタンスを所有しているか気にする必要が無い。
  - 責任を明確にしなくて良い。

- 上手に使えばリソースリークが無くなる。

# 内部実装

# Boost.SharedPtr

- 内部でインスタンスの参照数をカウントしている。
  - 参照数が 0 になったら解放
  - コピーするたびに参照数を増やす
  - デストラクタが呼ばれるたびに参照数を減らす

# Boost.SharedPtr

```
template<class T>
class shared_ptr {
    T* p_;
    int* count_;
    void release() {
        if (--*count_ == 0) {
            delete p_;
            delete count_;
        }
    }
public:
    explicit shared_ptr(T* p) : p_(p), count_(new int(1)) { }
    shared_ptr<T>& operator=(const shared_ptr<T>& s) {
        release();
        p_ = s.p_;
        count_ = s.count_;
        ++*count_;
        return *this;
    }
    ~shared_ptr() { release(); }
};
```

# Boost.SharedPtr

```
delete p_;
```

・デフォルトでは delete で解放される。
　自作のアロケータでも対応可能。

```
Hoge* AllocHoge();
void DeallocHoge(Hoge* p);
struct HogeDeleter {
    void operator()(Hoge* p) { DeallocateHoge(p); }
};
{
    boost::shared_ptr<Hoge> hoge(AllocHoge(), DeallocHoge);
    boost::shared_ptr<Hoge> hoge2(AllocHoge(), HogeDeleter());
}
```

## Boost.SharedPtr

```
template<class T>
class shared_ptr {
    T* p_;
    Deleter d_;
public:
    template<class Deleter>
    shared_ptr(T* p, Deleter d) : p_(p), d_(d) { }
    ~shared_ptr() { d_(p_); }
};
```

- Deleter をメンバに持てないので不可

↓

# Type Erasure

# Boost.SharedPtr

```
struct placeholder {
    virtual ~placeholder() { }
    virtual void destroy() = 0;
};
template<class T, class Deleter>
struct holder : public placeholder {
    T* p_;
    Deleter d_;
    holder(T* p, Deleter d) : p_(p), d_(d) { }
    virtual void destroy() {
        d_(p_);
    }
};
```

```
Hoge* p;
HogeDeleter d;
placeholder* holder(
    new holder<Hoge, HogeDeleter>(p, d));
...
holder->destroy();
```

# Boost.SharedPtr

```
template<class T>
class shared_ptr {
    T* p_;
    int* count_;
    placeholder* holder_;
public:
    template<class Deleter>
    shared_ptr(T* p, Deleter d)
        : p_(p), count_(new int(1))
        , holder_(new holder<T, Deleter>(p, d)) { }
    ...
    ~shared_ptr() {
        if (--*count_ == 0) {
            holder_->destroy();
            delete holder_;
            delete count_;
        }
    }
};
```

- Boost.SharedPtr まとめ
  - 参照カウント方式
    - コピーコンストラクタ・代入演算子やデストラクタを使ってカウントしている
  - 任意の削除方法を指定可能
    - Type Erasure という技法を使っている
- Boost.SharedPtr おまけ
  - boost::shared_ptr 同士のキャストが可能
  - ポインタと同程度にスレッドセーフ
  - 例外安全の保証

# Boost.Spirit

# Boost.Spirit

- Boost.Spirit とは

  →C++で書くことのできる構文解析器

- 構文解析とは

  →Wikipedia「ある文章の文法的な関係を説明すること」

  →k.inaba「プログラミング言語や設定ファイルなどの構造を持ったテキストデータを、ただの文字列として与えられた状態から、言語の文法などにしたがって構造を取り出す作業」(Boost C++ Libraries 第2版より引用)

- 文法

例)XMLタグの文法

```
〈tag attr1=″value″ attr2=″value″　　〉
```

(1) 最初に '<' が来る必要がある
(2) その次にタグ名が来る必要がある
(3) 次に[空白 属性]というパターンが0回以上連続する
(4) 次には空白があっても無くても良くて
(5) 最後は '>' で終わる

という感じで定義されている。

↑こういう規則のことを文法という

- 文法の表現方法

日本語

「a の次に B が続き、その次は d になる。
ただし B は b または b の次に c が続くという意味である。」

BNF

```
<rule> ::= a<B>d
<B> ::= b|bc
```

正規右辺文法（正規表現）

```
a(b|(bc))d
```

Boost.Spirit

```
rule = ch_p('a') >> B >> 'd';
B = ch_p('b') | (ch_p('b') >> 'c');
```

# デモ

# 内部実装

- 式の構造を記憶している

```
ch_p('a') >> B
```

'a' と B が >> で繋がれたことを覚えている

→'a' の次に B が来るかどうか、という解析が出来る

```
ch_p('a') | B
```

'a' と B が | で繋がれたことを覚えている

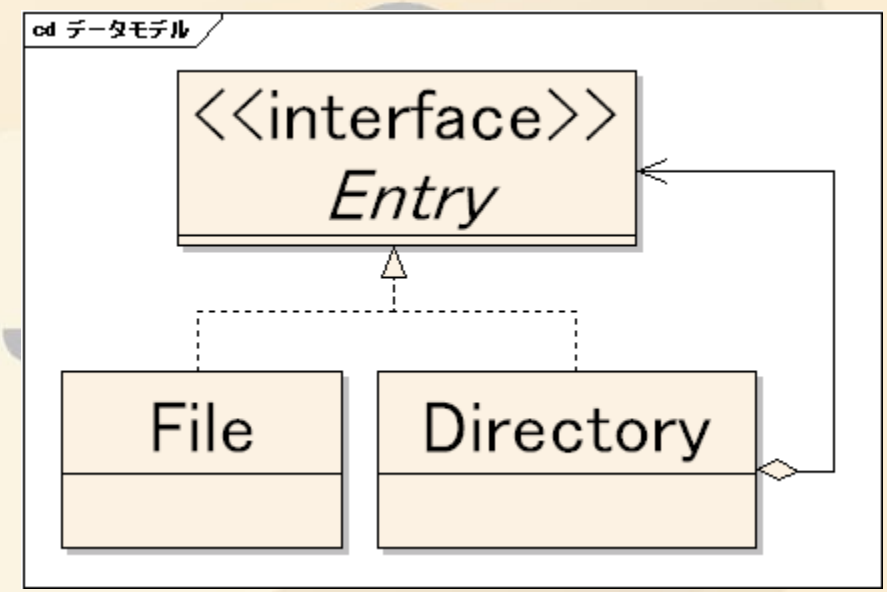
→'a' になるか B になるか、という解析ができる

```
B = ch_p('b') | (ch_p('b') >> 'c');
```

B も式になっている

→再帰的な構造になっている

- 再帰的な構造をクラスで定義するパターン
  - Composite パターン
- 代表的な例
  - ディレクトリ構造



# Boost.Spirit

```
class parser {
    virtual bool parse(Input in) = 0;
};
```

```
parser* rule; parser* B;
rule = ch_p('a') >> B >> ch_p('d');
B = ch_p('b') | (ch_p('b') >> ch_p('c'));
```

```
bool result = rule->parse(Input("abcd"));
```

# Boost.Spirit

```
class chlit : public parser {
    char c_;
public:
    chlit(char c) : c_(c) { }
    virtual bool parse(Input in) {
        return in.read() == c_;
    }
};
```

```
parser* ch_p(char c) { return new chlit(c); }
```

# Boost.Spirit

```
class sequence : public parser {
    parser* left_; parser* right_;
public:
    sequence(parser* l, parser* r) : left_(l), right_(r) { }
    virtual bool parse(Input in) {
        if (!left_->parse(in)) return false;
        if (!right_->parse(in)) return false;
        return true;
    }
};
```

```
parser* operator>>(parser* left, parser* right) {
    return new sequence(left, right);
}
```

# Boost.Spirit

```
class alternative : public parser {
    ...
    virtual bool parse(Input in) {
        Input save(in);
        if (l_->parse(in)) return true;
        in = save;
        if (r_->parse(in)) return true;
        return false;
    }
};
```

```
parser* operator|(parser* left, parser* right) {
    return new alternative(left, right);
}
```

- 式の構造をインスタンスで表現する方法
  - 仮想関数経由なので遅い
  - 型情報が消える

↓

# Expression Template

# Boost.Spirit

```
class chlit {
    char c_;
public:
    chlit(char c) : c_(c) { }
    bool parse(Input in) {
        return in.read() == c_;
    }
};
```

```
chlit ch_p(char c) { return chlit(c); }
```

# Boost.Spirit

```
template<class Left, class Right>
class sequence {
    Left left_; Right right_;
public:
    sequence(Left l, Right r) : left_(l), right_(r) { }
    bool parse(Input in) {
        if (!l_.parse(in)) return false;
        if (!r_.parse(in)) return false;
        return true;
    }
};
```

```
template<class Left, class Right>
sequence<Left, Right> operator>>(Left left, Right right) {
    return sequence<Left, Right>(left, right);
}
```

# Boost.Spirit

```
template<class Left, class Right>
class alternative {
    ...
    bool parse(Input in) {
        Input save(in);
        if (l_.parse(in)) return true;
        in = save;
        if (r_.parse(in)) return true;
        return false;
    }
};
```

```
template<class Left, Right>
alternative<Left, Right> operator|(Left left, Right right) {
    return alternative<Left, Right>(left, right);
}
```

# Boost.Spirit

```
bool result =
  (ch_p('b') | (ch_p('b') >> ch_p('c'))).parse(Input("bc"));
```

↓

```
alternative<chlit, sequence<chlit, chlit> >
```

・式の構造を型で表現している

→Expression Template

# Boost.Spirit

```
?? B = (ch_p('b') | (ch_p('b') >> ch_p('c')));
```

- 戻り値の型が難しくて代入できない。
  - Type Erasure を使う
    - 仮想関数経由になるから遅くなる
    - ケースバイケース

- Boost.Spirit まとめ
  - 式の構造を型で表現している
    - Expression Template
  - 複雑になる型
    - Type Erasure
- Boost.Spirit おまけ
  - 実際は CRTP(Curiously Recurring Template Pattern) **が使われていたりしてもっと複雑**
  - Boost.Spirit は LL(k) の構文解析器

# 時間が余った時用

## 構文解析の手法

### 下降型

ルートのルールを展開していって、入力した文字列と一番下のルールの文字が一致するかどうかを調べる。

### 上昇型

入力の文字列がルールに変換できるかを調べ、最終的にルートのルールになるかどうかを調べる。

## Boost.Spirit は下降型

R→aBd
B→b|bc

# 参考

- **Boost 本家**

  http://www.boost.org/

  - Boost.SharedPtr (日本語)

    http://boost.cppll.jp/HEAD/libs/smart_ptr/shared_ptr.htm

  - Boost.Spirit (ちょっとだけ日本語)

    http://boost.cppll.jp/HEAD/libs/spirit/

- **Boost C++ Libraries プログラミング 第2版**

  http://www.kmonos.net/pub/BoostBook/

- **コンパイラの構成と最適化**

  http://www.k.hosei.ac.jp/~nakata/aCompiler/aCompiler.html

# おわり