使いたい標準C++機能がない環境でいかに実装・設計するか

高橋 晶 (Akira Takahashi)
faithandbrave@gmail.com
Preferred Networks, Inc.
2024/02/09 (金) C++ MIX #9

開発現場で使われているC++バージョンは？

- 2024年初頭の現在、開発現場で使われているC++のバージョンはどれでしょう？
  - 2023年末、Boost 1.84.0でC++03のサポートが終了しました
- C++14やC++17を使っているケースが多いかもしれませんね

使いたい標準ライブラリの機能がいま使えない

- 「C++17の`std::optional`を使いたいけど、
  開発環境はC++14なので使えない」
- しかし継続開発なので (リリースしておわりではない) いずれコンパイラをバージョンアップして使えるようになるはず

- そんな方向けに、
  - 標準ライブラリを実装できる力をつけよう
  - いずれ標準機能に差し替えることを見越した設計をしよう

というお話をしようと思います

実装例は時間の都合でひとつだけ紹介します

- C++17の`std::optional`
- 実装方針として、
  - フル実装を目指さない
  - 使う機能だけ実装すればよい

C++17 std::optionalとは

- 有効値か無効値どちらかが入る型

```
std::optional<int> opt; // optは無効値をもつ
opt = 3; // 有効値を代入
if (opt) { // 有効値をもっているか判定
  int r = opt.value(); // 有効値を取り出す
}

opt = std::nullopt; // 無効値を代入
```

optionalの実装ポイント2つ

1. 無効値という特殊な状態を表現する
2. むだに動的メモリ確保をしない
   - optional内でnew / mallocしない

無効値の表現

- 空の型 (タグ型と言ったりもする) nullopt_tを定義し、
  その (唯一の) 変数としてnulloptを定義する

```
struct nullopt_t {};
const nullopt_t nullopt{};
```

- optionalクラスでは、nullopt_t型が代入されたら値をクリアする

```
optional& operator=(nullopt_t) {
  reset();
  return *this;
}
```

- これタグディスパッチと呼ばれる手法で、オーバーロード解決のためだけの
  空の型・値は、標準ライブラリやBoostでたくさん使われている

ヒープを使わない有効値の表現 1/4

- 有効値・無効値でまっさきに思いつくのはポインタ

```
T* p = new T(value); // 有効値を代入
...
p = nullptr; // 無効値を代入
```

- ▲
- 頻繁に使う小さなユーティリティのために
  動的メモリ確保はしたくない

ヒープを使わない有効値の表現 2/4

- 有効値とフラグをもてばよいのでは？

```
T value;
bool has_value;
```

- ▲
- 有効値が代入されていないのに、型Tのオブジェクトが作られるのは避けたい

ヒープを使わない有効値の表現 3/4

- 配置new (placement new) すればよいのでは？

```
char value[sizeof(T)];
bool has_value;

// 有効値の代入
T* p = new (value) T(x);
has_value = true;

// 無効値の代入
p->~T();
has_value = false;
```

- ○
- C++03まではこれでよかった。C++11以降はもっとかんたん

ヒープを使わない有効値の表現 4/4

- 共用体を使おう

```
union {
  T value;
  bool null_state;
};
bool has_value;
```

```
// 有効値の代入
new(&value) T{x};
has_value = true;

// 無効値の代入
value.~T();
has_value = false;
```

- ◎
- できた。C++11からは共用体にクラスオブジェクトを入れられる

完成コード 1/2

```
#include <utility>
#include <stdexcept>

struct nullopt_t {};
const nullopt_t nullopt{};

template <class T>
class optional {
  union {
    T _value;
    bool _null_state = true;
  };
  bool _has_value = false;

public:
  optional(T&& x)
    : _has_value{true} { new (&_value) T{x}; }
```

完成コード 2/2

```
  optional& operator=(nullopt_t) {
    if (_has_value) {
      _has_value = false;
      _value.~T();
    }
    return *this;
  }

  explicit operator bool() const {
    return _has_value;
  }
  const T& value() const {
    if (_has_value) {
      return _value;
    }
    throw std::runtime_error("nullopt exception"); // 仮
  }
};
```

使用例

```
optional<int> opt = 3;
if (opt) {
  std::cout << opt.value() << std::endl;
}

opt = nullopt;
if (!opt) {
  std::cout << "nullopt" << std::endl;
}
```

将来の標準機能と差し替えられるようにしよう

- std名前空間に自作機能を入れるのはよくない (場合によってはコンパイルエラーになる) ので、stdexとかの名前空間に入れる

```
namespace stdex {
  struct nullopt_t {};
  template <class T> optional { … };
}
```

将来の標準機能と差し替えられるようにしよう

- 開発環境を更新してoptionalが使えるようになったら、stdex名前空間でstd::optionalを使えるようにする

```
#include <optional>

namespace stdex {
  using nullopt_t = std::nullopt_t;

  template <class T>
  using optional = std::optional<T>;
}
```

この設計は外部ライブラリにも使える

- 外部ライブラリを使う際に、ダイレクトに使わず自作名前空間やクラスでラップしておくと、多様な環境に対応させやすい

```
namespace ext {
#if defined(__ios)
  using Purchase = ios::Purchase;
#elif defined(__android)
  using Purchase = android::Purchase;
#endif
}
```

- 外部ライブラリを使う際の中間レイヤーを用意し、そこで環境差を吸収する共通インタフェースを作っておくと、あとあとラクができる

完全置き換えではなくラップする設計でもよい

- 標準ライブラリのインタフェースに縛られると実装がたいへんな場合
- 自作インタフェースで作って、将来的には標準ライブラリをラップしてインタフェースを合わせることもできる
- この方針だと、拡張機能をメンバ関数として作ることもできる (thenとか)

```
namespace stdext {
  template <class T>
  class Optional {
  public:
    Optional<R> then(F f) const {
      if (*this) return f(value());
      return {};
    }
  };
}
```

まとめ

- C++標準ライブラリの機能でも、ユーティリティ的なものは実装しやすいです
  - 実装してみた系の記事は昔からたくさんある
- 標準ライブラリは、とてもよく考えられた設計・実装なので、学ぶことで開発力が向上します
- リリースしておわりではない継続開発の現場が増えたので、将来のコンパイラバージョンアップも想定した設計ができるといいですね