Boost.勉強会 #9 つくば (2012-05-26)

C++ TIPS 4 CV修飾編

概要

- 主に cppll ML でご紹介してきた tips をC++の仕様をより掘り下げた形でまとめ直してみました。
- 今回はcv修飾にフォーカスした内容です。

概要

- cv修飾はC++のすばらしい機能のひとつではあるのですが、細部は残念ところが多々あり、注意が必要なのでしっかり復習しておきましょう。

C++ Tips

CV修飾

cv修飾ってなに？

- const 修飾と volatile 修飾を合わせて cv 修飾(cv-qualifiers)と呼ばれます。
  - constexpr はcv 修飾とは関係ありません。ヤツは 全然別物です。
  - cv 修飾は型とメンバー関数に対して付けることができます。
  - 元の型とは違う型と見なされ、オーバーロードの対象となります。

const

- const 修飾されたオブジェクトはJIS 的には定値オブジェクトと呼ばれます。
- const は厳密には二段階の意味があります。
  - 一つ目は読み込み専用(readonly)であること。
  - 二つ目は不変(immutable)であること。

const

- 読み込み専用(readonly)であること。
  - const なオブジェクトは書き換えられません。

const

- 不変(immutable)であること。
  - 生成時から const として作成されたオブジェクトは変更できません。
  - この場合、コンパイラはコンパイル時にオブジェクトを作ってしまうことができます。また、そういうオブジェクトをconst_castなどでconst外しを行い書き換えようとすると未定義動作となります。

const

- const 参照あるいはconstなオブジェクトを指し示すポインタは、その指し示すオブジェクトはreadonlyであるがimmutableであるとは限りません。

const

- readonlyだがimmutableではない例：

```
int a = 0;
const int & b = a;
printf("%d¥n", b); // 0
a = 1;
printf("%d¥n", b); // 1
b = 2;             // NG
```

volatile

- volatile 修飾されたオブジェクトはJIS 的には揮発性オブジェクトと呼ばれます。
- 最適化の抑止を意味します。
  - マルチスレッドプログラミングにおける諸問題の回避を行ってくれるわけではありません。
  - メモリマップドI/Oなどを想定した機能であり、実の所それ意外はほぼ要無しです。

文法

```
const int a = 0; // 前置
int const b = 0; // 後置
const volatile int e = 0; // const+volatile
volatile const int f = 0; // 順番は逆もOKで、意味は同じ。
const int const c = 0; // 重複はNG
const int volatile d = 0; // 重複しない前置+後置はOK
int * const g = 0; // ポインタに対する修飾は必ず後置
int * const * h = 0; // ポインタに対する修飾は必ず後置
const int * const * const i = 0; // ベース＋各ポインタに対して
```

文法

```
const int a = 0; // a は変更不可
const int * b = &a; // *b は変更不可, b は変更可
const int * const c = b; // c,*c は変更不可
const int * const * const d = &c; // d,*d,**d は変更不可
int e = 0; // e は変更可
int * const f = &e; // f は変更不可, *f は変更可
int * const * const g = &e; // g,*g は変更不可, **g は変更可
int * const * h = &e; // *h は変更不可, h,**h は変更可
```

文法

```
// 引数に対するcv修飾に応じて
// 適切にオーバーロードが機能します。
void func(int & a);
void func(int const & a);
void func(int volatile & a);
void func(int const volatile & a);
void func(int && a);
```

文法

```
// 参照やポインタでない場合は関数シグネチャ的には無視される。
void func(int a);
void func(int const a); // 宣言的には void func(X a); と等価。

// 定義的には違いがある。
void func(int a) {
  a = 0; // OK
}
void func(int const a) {
  a = 0; // constなので書き換えはNG
}
```

文法

```
class X {
  public:
    void func();
    void func() const;
    void func() volatile;
    void func() const volatile;
    void func() &&;
    // 呼び出し時の this に対するcv修飾に応じて
    // 適切にオーバーロードが機能します。
};
```

文法

```
class X {
  public:
    X(const int a) const;
    ~X() const;
    // コンストラクタとデストラクタに対するcv修飾はNG
    //（ コンストラクタの引数型に対するcv修飾はOK ）

    static void gunc() const;
    // 静的メンバー関数に対するcv修飾はNG
};
```

「よりcv修飾されている」

- cv修飾に関する文脈上「よりcv修飾されている(more cv-qualified)」という表現が現れることがありますが、これは規格上次の表のように定義されています。

「よりcv修飾されている」

修飾の順序関係		
cv修飾なし	＜	const
cv修飾なし	＜	volatile
cv修飾なし	＜	const volatile
const	＜	const volatile
volatile	＜	const volatile

const_cast

- cv修飾の状態のみを変換するキャストです。
- 名前は const_cast ですが、 volatile についても修飾を付けたり外したりできます。

const_cast

- よりcv修飾されている型へは暗黙の型変換が行われる為、明示的にキャストする必要はありません。
  - これは安全なキャストについては暗黙の型変換が行われるということであり、const_castが必要になるという事は安全でないキャストを行っているということであり、注意が必要です。

const_cast

```
int a = 0;
const int * b = &a;
int * c = b;                          // NG
int * d = const_cast<int *>(b); // const をー つ除去
const int * * e = &d;                 // NG(後述)
const int * * f = const_cast<const int * *>(&d);
                                      // const をー つ付加※
const int * const * g = ...;
int * * h = const_cast<int * *>(g); // const をー つ除去
```

修飾の変換・互換ルール

- cv修飾の差異による型の変換・互換ルールは「よりcv修飾されている型へは暗黙の型変換が行われる」ということさえ頭に入れておけば後は概ね直感的に理解しやすいものなのですが、厳密にはやや変則的で落とし穴が無くもないのでひとつひとつ見ていきましょう！

型の互換性：値型

```
int a = 0;
const int b = a; // コピー
int c = b;       // コピー
```

...値型の場合、cv修飾の状況の関わらずコピーされる。

※サンプルは代入ですが、関数の引数についてもルールは同じ。

型の互換性：参照型

```
int a = 0;
const int & b = a; // よりcv修飾されている型へはOK
int & c = b;       // 逆方向はNG
int & d = const_cast<int &>(b); // キャストすればOK
```

型の互換性：ポインター型

```
int a = 0;
const int * b = &a; // よりcv修飾されている型へはOK
int * c = b;                          // 逆方向はNG
int * d = const_cast<int *>(b); // キャストすればOK
int * * e = &b;                       // NG
int * * f = &d;                       // OK
const int * * g = &b;                 // OK
const int * * h = &d;                 // NG(後述)
int * const * i = &b;                 // NG
int * const * j = &d;                 // OK
const int * const * k = &b;           // OK
const int * const * l = &d;           // OK
```

直感的でない型の非互換性

- なぜ const int * * 型の値に int * * 型の値を代入できないのか？

```
int a = 0;
int * b = &a;
const int * * c = &b; // NG
```

直感的でない型の非互換性

- ポインタ部分を除けば const int 型と int 型であり問題はない。

```
int a = 0;
const int b = a; // OK(コピー)

int c = b; // 逆方向もOK(コピー)
```

直感的でない型の非互換性

- ポインタがあっても一段の const int * 型と int * 型であれば同じく問題はない。

```
int a = 0;
int * b = &a;
const int * c = b; // OK
int * d = c;       // NG
```

直感的でない型の非互換性

- しかし、const int ** 型の値に int ** 型の値の代入はエラーになる。

```
int a = 0;
int * b = &a;
const int * * c = &b; // NG
```

直感的でない型の非互換性

- もしエラーにならないとしたら？

```
int a = 0;
int * b = &a;
const int * * c = &b; // 本当はエラー
```

直感的でない型の非互換性

- もしエラーにならないとしたら？

```
int a = 0;
int * b = &a;
const int * * c = &b; // 本当はエラー
const int d = 0;
*c = &d;
*b = 1;
```

直感的でない型の非互換性

- もしエラーにならないとしたら？

```
int a = 0;
int * b = &a;
const int * * c = &b; // 本当はエラー
const int d = 0;
*c = &d; // *c も &d ともに const int * 型
*b = 1;
```

直感的でない型の非互換性

- もしエラーにならないとしたら？

```
int a = 0;
int * b = &a;
const int * * c = &b; // 本当はエラー
const int d = 0;
*c = &d; // *c も &d ともに const int * 型
*b = 1; // おかしくね？ *b が指してんの d だよ！
```

直感的でない型の非互換性

- const int * * 型の値に int * * 型の値の代入を許してしまうと結果として明示的にキャストを使ってもいないのにconst外しを許してしまうことになってしまう！
  - とまぁ、そんなことにならないようにC++のコンパイラはエラーとして弾いてくれます。

伝播性

- ある変数をcv修飾するとその変数を参照あるいはポインタ経由でアクセスするコードが全て同じくcv修飾されていることが要求されるようになります。

伝播性

- オブジェクトがcv修飾されると同様な効果がそのデータメンバーとメンバー関数にも及びます。データメンバーについてはさらにそれらのデータメンバーのデータメンバーにも伝播が連鎖します。

伝播性

- 最終的に末端部分で呼び出す外部から提供されるライブラリの関数(API)が適切にcv修飾されていない場合、必要に応じてconst_cast を使用してcv修飾を外すことになりますが、本当に外してもよいのか注意する必要があります。

mutable

- オブジェクトがconst修飾されている場合でもデータメンバーにmutableを指定することでconst修飾を伝播させないこともできます。
  - const修飾の伝播のみを無効化するもので、mutableが指定されていてもvolatile修飾は伝播します。

mutable

```
class X {
  public:
    mutable int a;
    int b;
    X() { }
};

const X x;
x.a = 42; // mutableが効いているので代入可能
x.b = 42; // constなのでNG
```

伝播性の抜け・・・

```
class X {
  public:
    int a;
    int * b;
    int & c;
    X() :a(0), b(&a), c(a) { }
};
```

```
const X x;
//x.a = 42; これはさすがにNG
printf("%d¥r¥n", x.a); // 0
*x.b = 1; // これはまだしも・・・
printf("%d¥r¥n", x.a); // 1
x.c = 2; // しれっと代入可能・・・
printf("%d¥r¥n", x.a); // 2
```

伝播性の抜け・・・

- ほかにもコンストラクタおよびデストラクタにはcv修飾がかからない為、コンストラクタおよびデストラクタ内では本来 *this がconstな状態であってもcv修飾がかかっていないものとして動作します。
- const修飾されているオブジェクトであっても delete で破棄できちゃったり・・・

伝播性の抜け・・・

```
class X {
  public:
    int value;
    X() {
        value = 0;
        func();
    }
    ~X() {
        value = 42;
        func();
    }
    void func();
};
```

```
const X x;

const X * y = new X();
delete y;
```

一時オブジェクトの延命

- const且つ参照で受け取ることで一時オブジェクトをその受け取った側の変数と同じスコープで延命できます。

```
std::string func();

std::string name1 = func();          // コピー
const std::string name2 = func();    // コピー
const std::string & name3 = func(); // 一時オブジェクトの延命
std::string & name4 = func();        // NG
```

volatileメンバー関数でロック

- ちょっとトリッキーなvolatile修飾の使い方として、スレッド間で共有して使われることがあるオブジェクトをvolatile修飾によってマークし、そのメンバー関数が呼び出される際にvolatile修飾されている場合にのみロックを行う手法があります。

volatileメンバー関数でロック

```
class X {
  public:
    int func() { ... }
    int func() volatile {
        return auto_lock(), const_cast<X *>(this)->func();
    }
};

X a;
a.func(); // スレッド間共有しないインスタンスではロックせずに処理
volatile X b;
b.func(); //スレッド間共有するインスタンスではロックして処理
```

C++ Tips

質疑応答

C++ Tips

ご清聴ありがとうございました。