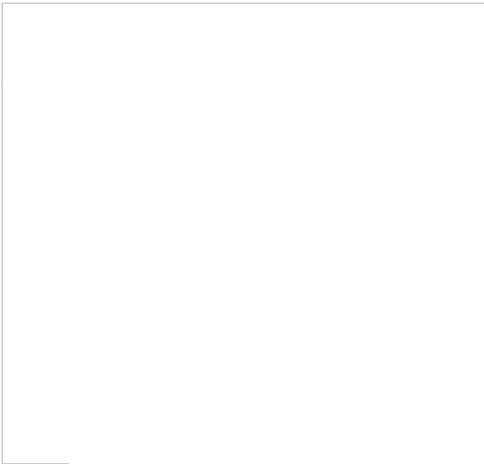


読者になる

yohhoyの日記 (別館)

もうちょい長めの技術的メモをしていきたい日記



2013-12-15

スレッドセーフという幻想と現実

C++

この記事は[C++ Advent Calendar 2013](#)の15日目にエントリしています。  
内容はC++標準ライブラリとスレッドセーフに関する解説になります。



flickr / [rennasverden](#)

もくじ

1. What's スレッドセーフ？
1. スレッドセーフという幻想
2. 基本型とデータ競合
3. C++標準ライブラリとデータ競合
4. C++標準ライブラリ：シーケンスコンテナ編
5. C++標準ライブラリ：連想コンテナ編
2. スレッドセーフ RELOADED
1. 基本的なスレッドセーフ保証
2. std::shared\_ptr<T>
3. std::rand()

プロフィール



[yoh \(id:yohhoy\)](#)  
もうちょい長めの技術的メモをしてい  
きたい日記

+ 読者になる 46

[このブログについて](#)

リンク

[yohhoyの日記](#)

カテゴリー

- [Boost \(3\)](#)
- [C \(4\)](#)
- [C++ \(13\)](#)
- [MSVC \(1\)](#)
- [POSIX \(1\)](#)
- [VB6 \(1\)](#)
- [Windows \(2\)](#)

最新記事

- [C++ MIX #5に参加しました](#)
- [C++ミューテックス・コレクシ  
ョン -みゅーこれ- 実装編](#)
- [C++ミューテックス・コレクシ  
ョン -みゅーこれ- 紹介編](#)
- [メモリモデル？なにそれ？おい  
しいの？](#)
- [条件変数 Step-by-Step入門](#)

検索

記事を検索

🔍

B!エントリー

4. `std::cout`

（本文のみ約9000字）

## はじめに

マルチスレッド対応の点では他言語に遅れを取っていたプログラミング言語C++ですが、C++11ではようやく標準ライブラリにスレッドサポートが追加されました。C++11スレッドサポートではスレッドクラス `std::thread` をはじめとし、ミューテックス `std::mutex` や条件変数 `std::condition_variable` などの基本的な同期機構、Future/Promiseパターンを実現する `std::future` , `std::promise` 、少しマニアックなところではアトミック変数 `std::atomic` などがC++標準ライブラリに追加されました。Happy Multithreading Programming!

という風に、C++11スレッドサポートの文脈では新規クラス追加ばかりが目立っています。このような華やかな表舞台の裏側では、以前からある `std::string` や `std::vector` といったクラス群に対して、スレッドサポートに関連する重要なルールが明確化されたことは知っていますか？マルチスレッドプログラム中でC++標準ライブラリを利用するとき、それらの**スレッドセーフ性(thread safety)\*1**について理解し、適切に使えているでしょうか？

本記事では、プログラミング言語C++におけるスレッドセーフの観点から、**C++11スレッドサポート追加クラス以外のC++標準ライブラリ**をいくつか紹介したいと思います。その中でも、たまに論争ネタになっている `std::shared_ptr` や `std::cout` のスレッドセーフ性についても説明を加えていきたいと思います。

## 1. What's スレッドセーフ？

複数スレッドを扱うプログラミングについて学び始めると、必ず「**スレッドセーフ(thread safe)**」という概念が出てくると思います。曰く、

- △△ライブラリはスレッドセーフだから、マルチスレッドプログラム中でも安全に使える。
- クラス○○はスレッドセーフじゃないから、スレッドと一緒に使えない。

どうやら「スレッドセーフなC++ライブラリ=マルチスレッド対応」という雰囲気のようなのです。スレッドセーフなんて簡単ですね！

...でも、本当にそんなに単純な話でしょうか。次の「スレッドセーフ」に関する質問に答えられますか？（Q&Aサイトでよく見かける質問の変形です）

- Q1. `std::map<K,V>` クラスはスレッドセーフですか？
- Q2. `std::vector<T>` クラスはスレッドセーフですか？
- Q3. `std::string` クラスはスレッドセーフですか？
- Q4. 基本型 `int` はスレッドセーフですか？
- Q5. そもそもC++言語での「スレッドセーフ」は何を意味しますか？

これらに回答できるなら、プログラミング言語C++の文脈での「スレッドセーフ」という概念について、ちゃんと理解していると言っても差し支えないでしょう。本

[スレッドセーフという幻想と現実](#) **332 users**

[本当は怖くないムーブセマンティクス](#) **129 users**

[メモリモデル？なにそれ？おいしいの？](#) **109 users**

[条件変数 Step-by-Step入門](#) **21 users**

[volatile教、あるいはvolatile狂](#) **8 users**

### 参加グループ

 [プログラミング](#)

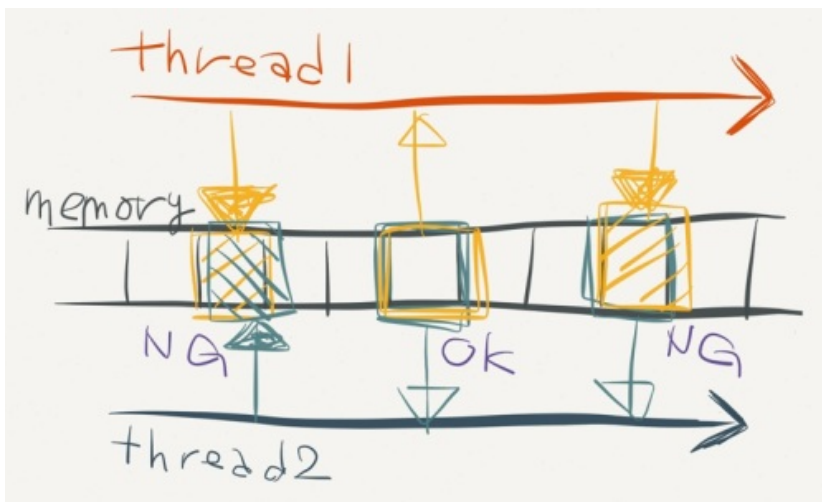
記事の前半では、逆順にてそれぞれの質問に答えていきたいと思います。

## 1.1. スレッドセーフという幻想

Q5. そもそもC++言語での「スレッドセーフ」は何を意味しますか？

いきなり出鼻をくじくようですが、プログラミング言語C++の言語仕様としては「スレッドセーフ」という用語は直接定義されていません。ただし、スレッドセーフを考える上で重要な用語「**データ競合(data race)**」は、C++言語仕様の一部として厳密に定義されます。C++11におけるデータ競合とは、次の3条件を全て満たすときに発生します。<sup>\*2</sup>

- (1) 同一メモリ位置に対するアクセスにおいて、
- (2) 少なくとも一方が**変更(modify)**操作であり、
- (3) 異なるスレッド上から**同時に行われる**とき。



そして、C++言語仕様では「プログラム中で何らかのデータ競合が発生した場合には、**未定義の動作(undefined behavior)**を引き起こす」と明言しています。つまり、ひとつでもデータ競合を含むマルチスレッドプログラムの動作結果について、C++言語仕様としては何も保証しないことを意味します。ここでの“何も保証しない”とは、本当に何も保証されていないことに注意してください。例えば、データ競合を含むプログラムは偶然正しく動くかもしれないし、運悪く期待する結果とならないかもしれないし、はたまたプログラム実行中に突然異常終了するかもしれないという意味です。<sup>\*3</sup>

残念ながら原典をあたっても「スレッドセーフ」についての明確な答えは得られませんでした。少なくともデータ競合を含まないプログラムとなっていればよい事は分かりました。

Q5. そもそもC++言語での「スレッドセーフ」は何を意味しますか？

A5. 厳密には「スレッドセーフ」は定義されない。一方、妥当なC++プログラムはデータ競合を一切含まないことが求められる。

と、このままでは話が終わってしまいますので、厳密なC++言語仕様から少し離れてみたいと思います。ある対象の「スレッドセーフ」について言及するときは、その利用者視点から議論するのが一般的と考えられます。本記事でもそれにならい“利用者側コードからどのように扱えばデータ競合が生じないか”、より具体的には“いつ

「排他制御すれば良いのか」という観点から、基本型やクラスの「スレッドセーフ」の議論を続けていきます。

## 1.2. 基本型とデータ競合

Q4. 基本型 `int` はスレッドセーフですか？

`char` や `int` などの基本型<sup>4</sup>は、C++プログラムを構成する最も原始的なデータ構造です。このような基本型変数は、一定サイズのメモリ領域にマッピングされます（例： `sizeof(int) == 4` な環境では `int` 型変数は連続した4バイトメモリ位置を占める）。つまり、基本型に関するスレッドセーフの議論は、前述「データ競合」の定義から単純に読み替えが可能です。

ところで「データ競合」は3つの発生要因から構成されますが、逆に言うと、これらの要因のうち1つでも排除できれば回避できます。

- (1) 異なる変数に対する同時アクセスは、データ競合とはなりません。
- (2) 同一変数に対して同時読み込み(read)操作のみならば、データ競合とはなりません。
- (3) 同一変数に対するアクセスが同時でなければ、データ競合とはなりません。

(1)はアクセスする変数自体が別々となるため、最も直感的に理解できるかと思います。（以降のコード例では、関数 `th1()`、`th2()` 等は異なるスレッドからそれぞれ同時に呼び出されるとします。）


```
// 異なる変数に対する同時アクセス
int x, y;
void th1() {
    x = 1; // OK
}
void th2() {
    y = 2; // OK
}
```

また(3)はマルチスレッドプログラムの定番、ミューテックス `std::mutex` などを用いた排他制御により同一変数へのアクセスが同時に生じないように制御します。これも、マルチスレッドプログラミングの基本ですね。

```
// 同一変数に対するアクセスが同時でない
std::mutex mtx;
int x;
void th1() {
    std::lock_guard<decltype(mtx)> lk(mtx);
    x = 1; // OK
}
void th2() {
    std::lock_guard<decltype(mtx)> lk(mtx);
    x = 2; // OK
}
```

一番馴染みが薄いのは(2)でしょうか？C++における「データ競合」の定義に従うと、同一変数に対する読み込み操作のみであれば安全に同時アクセスが可能なので（変数 `x`）。ただし、1つでも変更操作が発生するケース（変数 `y`）では、データ競合を回避するために排他制御が必要となることに注意してください。

```
// 同一変数に対して同時読み込み操作のみ
int x = 0;
void th1() {
    int r1 = x; // OK
}
```

 Hatena Blog yohhoyの日記（別館）  記事を書く  ダッシュボード 無料で新規登録  Hatena

```
int r2 = x; // OK
}
```

```
// 同一変数に対して片方が変更操作
int y = 2;
void th1() {
    int r2 = y; // NG: データ競合
}
void th2() {
    y = 42; // NG: データ競合
}
```

利用者視点で考えると、上記ルール“**全てが読み込み操作ならば排他制御は不要である**”というのが、C++基本型が提供するスレッドセーフ保証と解釈できます。

Q4. 基本型 `int` はスレッドセーフですか？

A4. 「同時アクセスが全て読み込み操作であれば安全」というスレッドセーフ性レベルが保証される。

### 1.3. C++標準ライブラリとデータ競合

Q3. `std::string` クラスはスレッドセーフですか？

さて基本型のスレッドセーフ保証は分かりましたが、C++標準ライブラリ提供のクラスではどうでしょう。例えば文字列型 `std::string` クラスでは、明らかにprivateなメンバ変数として複数の基本型（文字列ポインタやバッファ長など）を内包しているはずです。一方でC++11以前のプログラム動作との互換性を考慮すると、クラス内部実装にミューテックスなど排他制御の仕組みがこっそり追加されたとは考えにくいですね。そうなると、マルチスレッドプログラムからこれらのクラスを安全に利用するには、利用者側にて全てのメンバ関数呼び出しを排他制御しないとダメなのでは...？

```
std::string s = "hello";
void th1()
{
    // ここに排他制御は必要？
    bool b = s.empty();
}
void th2()
{
    // ここに排他制御は必要？
    char& c = s.at(0);
}
```

実は、C++11における重要な（そして地味な）スレッドサポートとして、C++標準ライブラリ提供クラスのデータ競合に関する基本ルールが明確化されました。

- クラスオブジェクトのconstメンバ関数呼び出しは、オブジェクトに対する読み込み操作とみなす。
- クラスオブジェクトの非constメンバ関数呼び出しは、オブジェクトに対する変更操作とみなす。（一部例外事項あり）

このルールはC++標準ライブラリ提供のクラスに対して適用される、最低限かつ基本的なスレッドセーフ保証となっています。また追加的な例外事項として、

`begin`、`end` や `at` などの一部の非constメンバ関数ではオブジェクト自身の内部状態を変更しないため、これらもデータ競合に関してはconstメンバ関数相当（＝読み込み操作）とみなします。

```
std::string s = "hello";
void th1()
{
    if ( !s.empty() )           // OK: constメンバ関数呼び出し
        printf("%s\n", s.c_str()); // OK: constメンバ関数呼び出し
}
void th2()
{
    size_t n = s.length(); // OK: constメンバ関数呼び出し
    char& c = s.at(0);      // OK: constメンバ関数相当呼び出し
}
```

つまり、データ競合を回避する3つの方法は次のように拡張できます。

- (1) 異なるオブジェクトに対する同時アクセス／メンバ関数呼び出しは、データ競合とはなりません。
- (2) 同一オブジェクトに対して同時読み込み操作＝constメンバ関数相当呼び出しのみならば、データ競合とはなりません。
- (3) 同一オブジェクトに対するアクセス／メンバ関数呼び出しが同時でなければ、データ競合とはなりません。

こちらら利用者視点の表現では、上記ルール“全てconstメンバ関数相当呼び出しならば排他制御は不要である”というのが、C++標準ライブラリが提供するスレッドセーフ保証と解釈できます。

Q3. `std::string` クラスはスレッドセーフですか？

Q3. 「同時アクセスが全て読み込み操作（constメンバ関数相当呼び出し）であれば安全」というスレッドセーフ性レベルが保証される。

## 1.4. C++標準ライブラリ：シーケンスコンテナ編

Q2. `std::vector<T>` クラスはスレッドセーフですか？

他の型を内包するシーケンスコンテナ(sequence containers)クラスはどうでしょうか（説明のため `T=int` とする）。コンテナクラスのデータ競合を考える場合、コンテナオブジェクト( `vector<int>` )と格納された各要素( `int` )は異なるオブジェクトであると認識することが重要です。

コンテナ要素へアクセスする最も一般的な方式は `vector<T>::operator[]` メンバ関数呼び出しでしょう。この `operator[]` にはconst・非constメンバ関数の2つが存在しますが、前掲の「基本的なスレッドセーフ保証」により両者ともconstメン

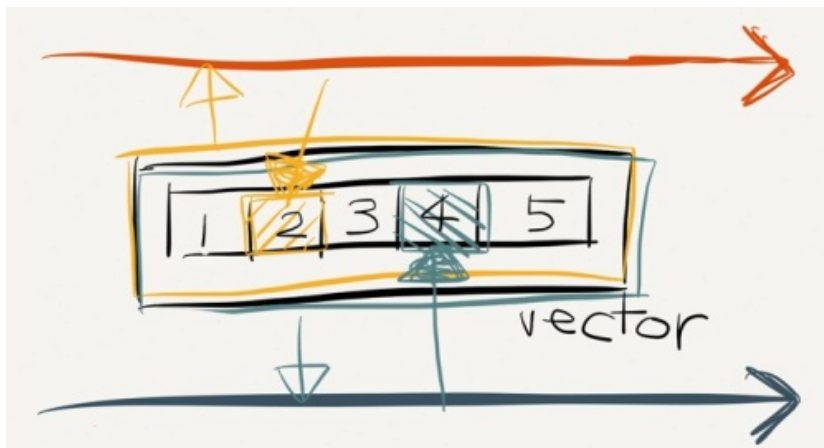


バ関数（相当）として扱われます。このため、コンテナオブジェクトの他`constメンバ関数`と同時に呼び出してもデータ競合とはなりません。

```
std::vector<int> v = { 1, 2, 3, 4, 5 };
void th1()
{
    size_t n = v.size(); // OK: v.size()は、vに対する読み込み操作
}
void th2()
{
    v[1] *= 2; // OK: v[1]つまりv.operator[](1)は、vに対する読み込み操作
              // v[1] *= 2は、vの2番目要素に対する変更操作
}
```

`operator[]` や他メンバ関数（イテレータ経由や `at` など）を介して得られた各コンテナ要素オブジェクトへのアクセスについては、既に説明してきたデータ競合の考え方と同じになります。つまり、異なるコンテナ要素オブジェクトへのアクセスであれば、同時に変更操作を行ってもデータ競合とはなりません。((このコンテナ要素アクセスに関するデータ競合に関して、1つだけ例外ケースが存在します。特殊化された `std::vector<bool>` だけは、異なる要素へのアクセスであってもデータ競合が発生する可能性があります。同テンプレート特殊化版の仕様が要請するメモリ空間最適化のために、異なる `bool` オブジェクトが“同一メモリ位置（バイト）”の異なるビットとして配置される可能性があるためです。))

```
std::vector<int> v = { 1, 2, 3, 4, 5 };
void th1()
{
    // v[1]つまりv.operator[](1)は、vに対する読み込み操作
    ++v[1]; // OK: vの2番目要素に対する変更操作
}
void th2()
{
    // v[3]つまりv.operator[](3)は、vに対する読み込み操作
    v[3] = 0; // OK: vの4番目要素に対する変更操作
}
```



Q2. `std::vector<T>` クラスはスレッドセーフですか？

A2. コンテナ自身 `std::vector<T>` および各要素 `T` に対して、個別に「同時アクセスが全て読み込み操作（`constメンバ関数`相当呼び出し）であれば安全」というスレッドセーフ性レベルが保証される。

## 1.5. C++標準ライブラリ：連想コンテナ編

Q1. `std::map<K,V>` クラスはスレッドセーフですか？

最後に、より複雑な連想コンテナ(associative containers)クラスをみていきましょう（説明のため `K=int` , `V=std::string` とする）。既に説明してきた通り、コンテナオブジェクト(`map<int,string>`)と格納されるキー(`int`)–値(`string`)をそれぞれ異なるオブジェクトと考える必要があります。これらのクラス／型はそれぞれで基本的なスレッドセーフ保証を提供しますから、もう説明しなくても分かりますよね。

```
std::map<int,std::string> m = { {1,"a"}, {2,"b"} };
void th1()
{
    size_t n = m.size(); // OK: m.size()は、mに対する読み込み操作
}
void th2()
{
    m.at(2) += "x"; // OK: m.at(2)は、mに対する読み込み操作
    // m.at(2) += "x"は、mの要素{2,"b"}の値に対する変更操作
}
```

ところで、上記コードでは意図的に `map<int,string>::operator[]` の利用を避けました。実は“連想コンテナクラスの `operator[]` メンバ関数は変更操作”と解釈するため、下記コードへ単純に置き換えるとデータ競合を引き起こしてしまうのです((連想コンテナ `std::map` , `std::unordered_map` では非const版 `operator[]` メンバ関数のみが提供され、そもそもconst版 `operator[]` メンバ関数は存在しません。))。少々不便に見えるかもしれませんが、このメンバ関数は“与えられたキー値が存在しなければ新要素を追加する”という動作仕様となっており、これはコンテナ自身の内部状態を変更しないことには実現できないからです。

```
std::map<int,std::string> m = { {1,"a"}, {2,"b"} };
void th1()
{
    size_t n = m.size(); // NG: データ競合
}
void th2()
{
    m[2] += "x"; // NG: データ競合
    // m[2]つまりm.operator[](2)は、mに対する変更操作
}
```

つまり `map<int,string>::operator[]` を安全に使うには、ミューテックスによる排他制御が必要となるのです。

```
std::mutex mtx;
std::map<int,std::string> m = { {1,"a"}, {2,"b"} };
void th1()
{
    std::lock_guard<decltype(mtx)> lk(mtx);
    size_t n = m.size(); // OK
}
void th2()
{
    std::lock_guard<decltype(mtx)> lk(mtx);
```



```
m[2] += "x"; // OK
}
```

C++標準ライブラリでは、連想コンテナ `std::map<K,V>` ,  
`std::unordered_map<K,V>` の2つで本ルールが当てはまります。

Q1. `std::map<K,V>` クラスはスレッドセーフですか？

A1. コンテナ自身 `std::map<K,V>` および各要素キー `K` と値 `V` に対して、別個に「同時アクセスが全て読み込み操作（constメンバ関数相当呼び出し）であれば安全」というスレッドセーフ性レベルが保証される。ただし `operator[]` はコンテナに対する変更操作であることに注意。

## 2. スレッドセーフ RELOADED

いかがでしょう。5つの質問には答えられましたか？ここでは、改めてC++標準ライブラリの基本的なスレッドセーフ保証について整理し、またこの考え方が特段目新しいものでないことを紹介します。さらに、しばしばスレッドセーフ論争を引き起こすクラスや関数をいくつか挙げて、追加的な解説を行いたいと思います。

### 2.1. 基本的なスレッドセーフ保証

前節の繰り返しとなりますが、C++言語仕様およびC++標準ライブラリでは次の「**基本的なスレッドセーフ保証**」を提供します。これは、C++標準ライブラリ提供のクラス／関数が提供する、最低限のスレッドセーフ性の保証となっています。ちなみに、ここで“最低限の”と表現しているのは、C++標準ライブラリの一部クラスではもっと強いスレッドセーフ保証（＝同時に変更操作を行っても安全）を提供するためです。

- 同一オブジェクトにする同時アクセスが全て読み込み操作であれば、排他制御なしでもデータ競合を引き起こさない。
- C++標準ライブラリ提供のクラスでは、constメンバ関数（およびオブジェクトを変更しない非constメンバ関数）呼び出しは読み込み操作とみなす。
- これ以外のケース（少なくとも片方が変更操作）の場合、同一オブジェクトへの同時アクセスを排他制御する必要がある。

さて、この「基本的なスレッドセーフ保証」ですが、実はC++11で新たに作られた考え方ではありません。C++98以前から存在しており、また現C++標準ライブラリの源流でもあるSGI STLライブラリでも、同等のスレッドセーフ保証を提供すると明言していました。[\\*5](#)

The SGI implementation of STL is thread-safe only in the sense that simultaneous accesses to distinct containers are safe, and simultaneous read accesses to shared containers are safe. If multiple threads access a single container, and at least one thread may potentially write, then the user is responsible for ensuring mutual exclusion between the threads during the container accesses.

[Standard Template Library Programmer's Guide, Thread-safety for SGI STL](#)

さらにC言語の時代にまでさかのぼると、POSIXシステムにおいても同等のスレッドセーフ保証が明言されていました。（こちらは対象がC言語のためconstメンバ関数は存在しません。）

（略） They usually cannot determine when memory operation order is important and generate the special ordering instructions. Instead, they rely on the programmer to use synchronization primitives correctly to ensure that modifications to a location in memory are ordered with respect to modifications and/or access to the same location in other threads. Access to read-only data need not be synchronized. The resulting program is said to be data race-free. [The Open Group Base Specifications Issue 6, IEEE Std 1003.1, 2004 Edition](#)

## 2.2. std::shared\_ptr<T>

C++11では所有権共有スマートポインタ `std::shared_ptr<T>` がC++標準ライブラリ入りしました。ここでは同一 `int` オブジェクトを参照する2つのスマートポインタが存在し、同時に `p1`, `p2` を操作するケースを考えてみます（説明のため `T=int` とする）。各スレッド上での操作により参照カウント更新が同時に行われるため、これまでの延長線上で考えると排他制御が必要に思えます。

```
std::shared_ptr<int> p1 = std::make_shared(42);
std::shared_ptr<int> p2 = p1;
// p1とp2は同一オブジェクトを参照
void th1() {
    // ここに排他制御は必要？
    p1.reset(); // 参照カウントを-1
}
void th2() {
    // ここに排他制御は必要？
    auto q = p2; // 参照カウントを+1
}
```

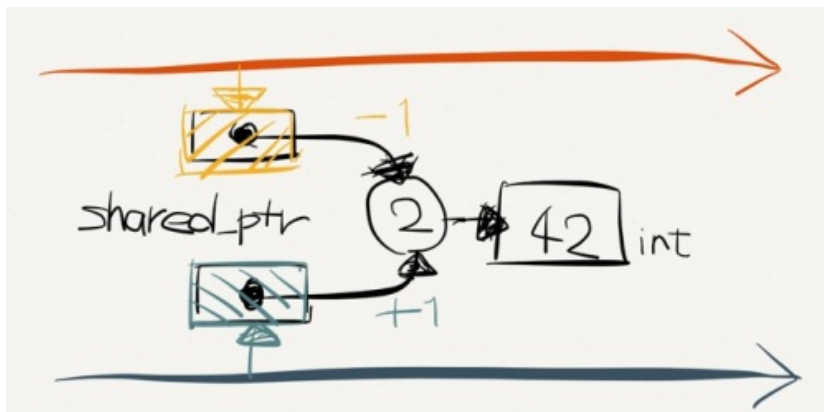
一方で、局所的な観点では `p1` と `p2` はあくまで異なる `std::shared_ptr<int>` オブジェクト”です。仮に両スマートポインタが異なる `int` オブジェクト／異なる参照カウントを参照するなら、C++標準ライブラリの基本的なスレッドセーフ保証より明らかに排他制御は不要です。とはいえ、その参照先が同一か否かで必要有無が変わるようでは、結局は保守的に排他制御をせざるを得なくなります。

```
std::shared_ptr<int> p1 = /* ??? */;
std::shared_ptr<int> p2 = /* ??? */;
// p1とp2は同一オブジェクトor異なるオブジェクトを参照
void th1() {
    // ここに排他制御は必要？
    p1.reset(); // 参照カウントを-1
}
void th2() {
    // ここに排他制御は必要？
    auto q = p2; // 参照カウントを+1
}
```

この問題に対して、C++標準ライブラリでは「異なる `std::shared_ptr` オブジェクトに対するメンバ関数呼び出しは、その参照先に関わらずデータ競合を引き起こさない」と明確に保証しています。つまり、利用者からは見えないオブジェクト内部状態を気にすることなく、基本的なスレッドセーフ保証の通り異なるオブジェク

トへの同時アクセスは安全”と扱ってして良いのです。これは裏を返すと、スマートポインタ `std::shared_ptr` の内部実装では、参照カウント更新はatomic変数操作（または排他制御あり変数更新）にて行われることを意味します。

```
std::shared_ptr<int> p1 = std::make_shared(42);
std::shared_ptr<int> p2 = p1;
// p1とp2は同一オブジェクトを参照
void th1() {
    p1.reset(); // OK: 参照カウント-1は安全に行われる
}
void th2() {
    auto q = p2; // OK: 参照カウント+1は安全に行われる
}
```



ただし、両スマートポインタ `p1`、`p2` が同一オブジェクトを参照している状況で、`shared_ptr<int>::operator*` メンバ関数を介して得られた参照先 `int` オブジェクトにアクセスする時には、`int` に対する基本的なスレッドセーフ保証に留意する必要があります。

```
std::shared_ptr<int> p1 = std::make_shared(42);
std::shared_ptr<int> p2 = p1;
// p1とp2は同一オブジェクトを参照
void th1() {
    *p1 += 1; // NG: 参照先においてデータ競合
    // 参照先オブジェクトに対する変更操作
}
void th2() {
    int r = *p2; // NG: 参照先においてデータ競合
    // 参照先オブジェクトに対する読み込み操作
}
```

```
std::shared_ptr<int> p1 = std::make_shared(42);
std::shared_ptr<int> p2 = p1;
// p1とp2は同一オブジェクトを参照
void th1() {
    int r = *p1; // OK: 参照先オブジェクトに対する読み込み操作
}
void th2() {
    int r = *p2; // OK: 参照先オブジェクトに対する読み込み操作
}
```

## 2.3. std::rand()

続いて、C言語時代の標準ライブラリから引きついだ乱数生成関数 `std::rand()` を取り上げてみましょう((例示コードでは整数0～9を得るために剰余演算(`%`)を利用

していますが、この実装では望ましい一様分布とはなりません。定数RAND\_MAXを利用すればもう少しまともな一様分布が得られますが、C++11以降では後述の乱数ライブラリ<random>を利用すべきです。))。C++11では、この関数を複数スレッドから同時に呼び出した場合、データ競合を引き起こすか否かは処理系定義(implementation-defined)と定めています。つまり、コンパイラやライブラリに任せるとしか言っていないのです。

```
void th1() {
    // 処理系によっては排他制御が必要
    int r1 = std::rand() % 10;
}

void th2() {
    // 処理系によっては排他制御が必要
    int r2 = std::rand() % 10;
}
```

(スレッドセーフの議論に関わらず) C++11以降では、C++標準ライブラリに追加された乱数ライブラリを用いるべきでしょう。この新しい乱数ライブラリ<random>であれば乱数生成器をスレッド別に持つことができるため、データ競合について悩む必要がなくなります。((例示コードでは乱数生成エンジンstd::mt19937を引数なしで構築しているため、常に同一の疑似乱数列が得られることに注意してください。実行のたびに異なる乱数列が必要な場合、std::random\_deviceなどで適当なシード値を与える必要があります。))

```
void th1() {
    std::mt19937 rng;
    std::uniform_int_distribution<int> dist(0, 9);
    int r1 = dist(rng); // OK
}

void th2() {
    std::mt19937 rng;
    std::uniform_int_distribution<int> dist(0, 9);
    int r2 = dist(rng); // OK
}
```

もちろん、下記コードのように単一の乱数生成器を複数スレッド間で共有し、アクセス時には排他制御を行うという実装でもOKです。ただし、この設計では“(a)高頻度で乱数生成を行う状況で、排他制御により他スレッドが停止するためプログラム処理速度に悪影響を与える”、“(b)各スレッドで取得する乱数列は実行時(OS)スレッドスケジューリングに依存するため、再現性のある乱数列を生成できない”という問題が生じます。最終的にはアプリケーションの目的によりませんが、一般にはスレッド別に乱数生成器を保持する設計の方が好ましいと思います。

```
std::mutex mtx;
std::mt19937 rng;
std::uniform_int_distribution<int> dist(0, 9);
// 単一の乱数生成器を排他制御ありで利用
int next_rand() {
    std::lock_guard<decltype(mtx)> lk(mtx);
    return dist(rng);
}

void th1() {
    int r1 = next_rand(); // OK
}

void th2() {
```

```
int r2 = next_rand(); // OK
}
```

今回は `rand()` 関数を取りあげましたが、C++標準ライブラリに取り込まれたC標準ライブラリの関数のうち、`strtok()`、`localtime()`、`setlocale()` などの一部関数は複数スレッドから同時に呼び出すとデータ競合となる可能性があります。できるだけC++標準ライブラリを使いましょう！

## 2.4. std::cout

最後に、おそらく一番論争を引き起こすであろう、標準出力ストリーム

`std::cout` のスレッドセーフ保証について整理します。C++標準ライブラリの基本的なスレッドセーフ保証に照らして考えると、各スレッドから同時に

`std::cout` へ出力するとき排他制御は必要でしょうか？

```
void th1() {
    // ここに排他制御は必要？
    std::cout << "I'm #1 thread." << std::endl;
}
void th2() {
    // ここに排他制御は必要？
    std::cout << "Hello, Multithreading World!" << std::endl;
}
```

この質問に対する回答は、2段階で考える必要があるでしょう。まず、C++11の標準ライブラリ仕様では「標準入出力ストリームオブジェクトに対する入力／出力操作は、排他制御を行わなくてもデータ競合を引き起こさない」と定めています。という訳で、上記コードはデータ競合なしに正常動作することが保証されています。つまり、標準入出力ストリームオブジェクト `std::cin`、`std::cout`、`std::cerr`、`std::clog`（と対応するwide文字版）では、C++標準ライブラリの基本的なスレッドセーフ保証よりも強いスレッドセーフ性を提供しています。（このような強いスレッドセーフ保証が行われるのは、あくまで標準入出力ストリームオブジェクトのみです。`std::fstream` や `std::stringstream` では、基本的なスレッドセーフ保証しか提供しないのでご注意ください。))

ただし、C++標準ライブラリでは「データ競合は起こさないが、複数スレッドからの出力が混ざる(interleave)可能性がある」とも言及しています。例えば標準出力には次のような文字列が出力されるかもしれませんが、これもプログラムが正常動作した結果なのです。

```
I'm #1
Hello,thread Multit
hreading World!
```

一般的には前掲コードを書いたプログラマが期待するのは、下記いずれかのように「行単位で出力されること」と考えられます。結局のところ標準出力ストリーム `std::cout` であっても、実用上は排他制御を行わない限りはまともな出力結果が得られないのです。

```
I'm #1 thread.
Hello, Multithreading World!
```

```
Hello, Multithreading World!  
I'm #1 thread.
```

## まとめ

これで今回の解説はおしまいです。最後にもう一度、本記事での要点をまとめてみましょう。

- C++11における基本型とC++標準ライブラリ提供クラスでは、少なくとも下記「基本的なスレッドセーフ保証」を提供します。
- 単一オブジェクトに対する同時アクセスが、全て“読み込み操作”=“const関数相当呼び出し”のみであれば、複数スレッド間での排他制御は不要です。
- このルールは、複数オブジェクト間で内部共有されるデータがあったとしても維持されます。（例：`std::shared_ptr`）
- `std::cout` 等の標準入出力ストリームを使う場合も、複数スレッド間での排他制御が必要になります。

*I Hope You Have A Nice Multithreading C++ Programming Life!!*

[C++ Advent Calendar 2013 は 16日目 disktnk さんの記事へと続きます。](#)



flickr / [applebymatt](#)

## 参考記事

本文中では次の記事をもとにした解説を行いました。より詳細についてはそれぞれの記事もご参考ください。

- [vector要素アクセスとスレッド安全性](#)
- [C++11のcoutとスレッド安全性](#)
- [C++11標準ライブラリのスレッド安全性](#)
- [shared\\_ptrとスレッド安全性](#)
- [std::random\\_shuffle関数はスレッド安全?](#)
- [「pthreadサポート」の意味するところ](#)
- [const, mutableキーワードとスレッド安全性](#)

はしがき：本文中でゆるふわ手書き画像を使っていますが、当初はInkscapeで描こうと思いつつ心が折れた結果です。字が汚いことを横に置けば、手書きも悪くないですね！

\*1: 本記事中では thread safe = スレッドセーフ との対比から thread safety = スレッドセーフ性 と記述します。なお、後者の対訳語としては「スレッド安全性」の方が一般的かと思いま



す。

\*2: 厳密にはもう1つの条件“(0) 対象がatomic変数でないとき”が追加されます。これを裏返すと、atomic変数に対するアクセスはデータ競合を引き起こさないことを意味します。本文中ではatomicでない通常の変数のみを対象とするため、この条件については言及していません。

\*3: 「データ競合」によるプログラム異常終了というのは直感に反するかもしれませんが、これは「未定義の動作」の結果としてあり得る動作の一つです。ちなみに、マルチスレッド分野でC++言語に大きな影響を与えたJava言語では、もう少し穏やかな定義「データ競合が生じると何らかの状態になる（が異常停止はしない）」となっています。

\*4: 組込み型やプリミティブ型と呼ばれることもあります。なお、C++言語仕様での正式名称はスカラ型(scalar type)です。

\*5: [id:y-hamigaki](#)さんのHamigaki C++ライブラリでは、SGI STLより明快到[スレッドセーフ保証について記載](#)しています。

yoh (id:yohhoy) 9年前



## 関連記事

[2019-09-06](#)

[C++ MIX #5に参加しました](#)

2019/9/4に開催されたイベント C++ MIX #5 にて、“20分くらいで...

[2014-09-23](#)

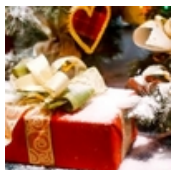
[条件変数 Step-by-Step入門](#)

多くのプログラミング言語では、マルチスレッド処理向け同期ブ...

[2013-05-11](#)

[volatile教、あるいはvolatile狂](#)

かつてのMicrosoft Visual Studio .NET 2003のC/C++コンパイラ(...



[2012-12-15](#)

[本当は怖くないムーブセマンティクス](#)

この記事はC++ Advent Calendar 2012の15日目にエントリーしてい...

[2012-07-06](#)

[UnVisualBasic 6.0](#)

この記事は2001年頃に書いた文章をそのまま転記し、はてなプロ...

[« gistにソースコードを放っておいたらOSSプ...](#)

[volatile教、あるいはvolatile狂 »](#)

はてなブログをはじめよう！

yohhoyさんは、はてなブログを使っています。あなたもはてなブログをはじめてみませんか？

はてなブログをはじめる（無料）

[はてなブログとは](#)

 [yohhoyの日記（別館）](#)  
Powered by [Hatena Blog](#) | [ブログを報告する](#)

