SWEST24
セッション s4b
「最近のCとC++を振り返る」

2022年9月2日（金）12:30～13:40

株式会社アイシン　間瀬　順一

セッションのねらい

組込みソフトの開発では、依然としてCを採用することが多いようです。また、自動車で採用が進んでいくことが予想される AUTOSAR Adaptive Platform の実装言語として、C++が採用されました。

これらを踏まえて最近のCとC++を取り巻く状況を概説します。形式は、ハイブリッド開催に配慮したレクチャー方式となります。

みなさまの参加について

- 基本的にレクチャー方式で進めますが、みなさまとの交流も重視したいと考えています。

- 質問やコメントがありましたら、ZOOMのチャットで気楽に送ってください。
- セッション後もSlackのチャンネルにコメントいただけたら、可能な範囲で対応します！
  https://app.slack.com/client/T03R79Z304A/C03UHDHC7FB

講師　自己紹介

氏名：間瀬　順一

所属：株式会社アイシン

バス・トラック向けAT制御ソフトの開発を担当していました。

今は、車載向けソフトウェアPF全般の開発に関わっています。

- 技術士（情報工学）
- 名古屋大学特任教授（2007/09/11/14/20）
- 自動車技術会 ソフトウェア更新分科会 委員

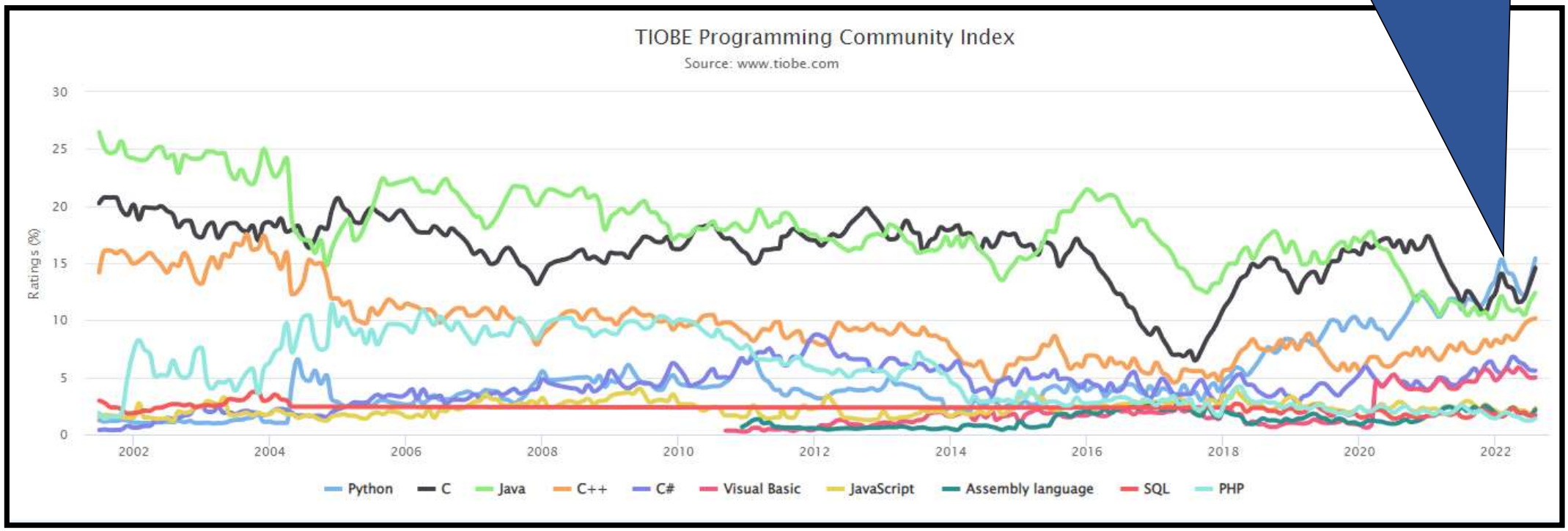
セッションの進め方（時間の目安）

- 現状把握　5分
- Cの現状について　20分
- C++の現状について　40分
- 質問やコメントの対応　5分

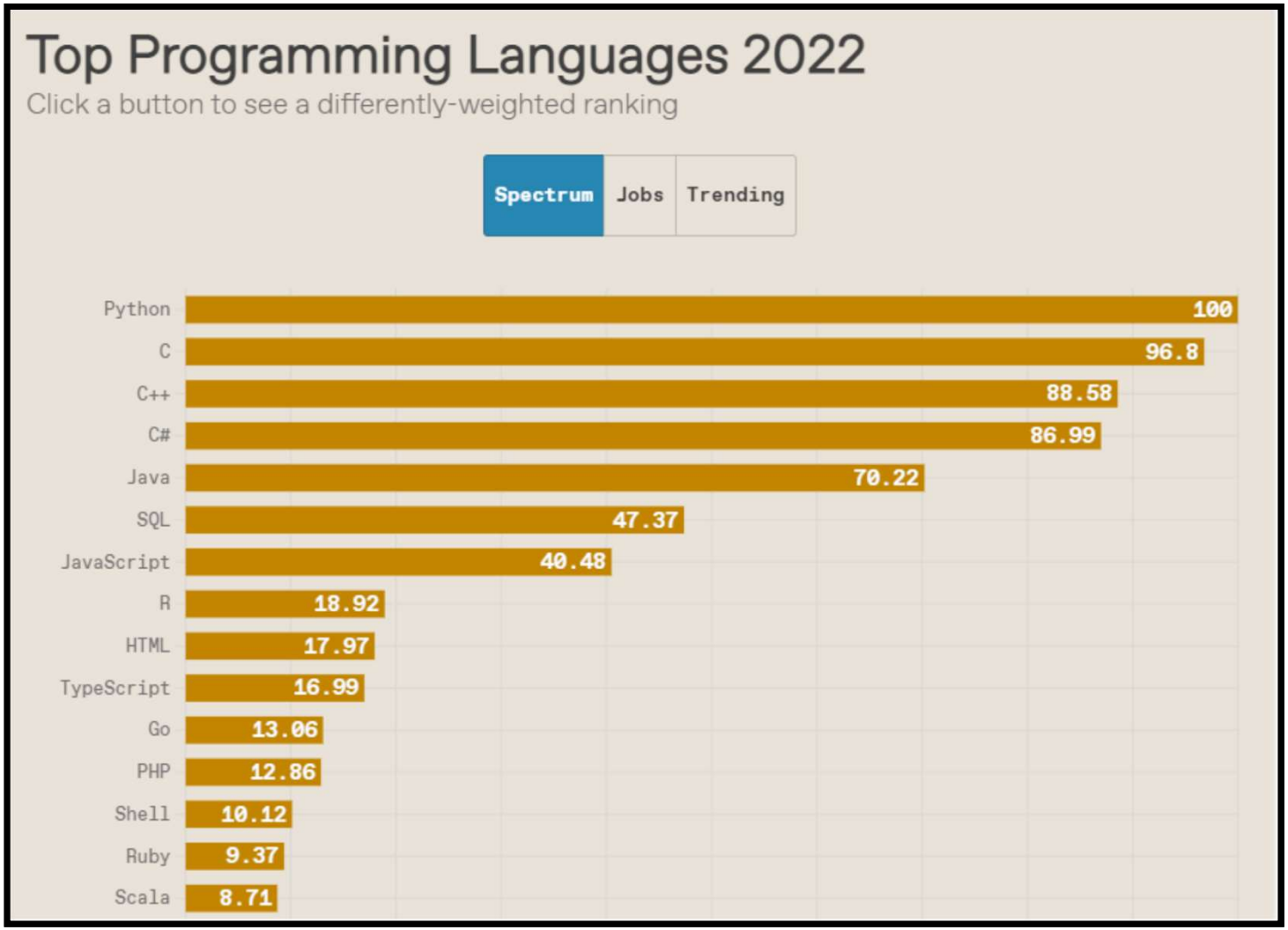
現状把握

TIOBE Index（ https://www.tiobe.com/tiobe-index/ ）

2021年10月版からPythonが首位に
Cは2位、C++は4位

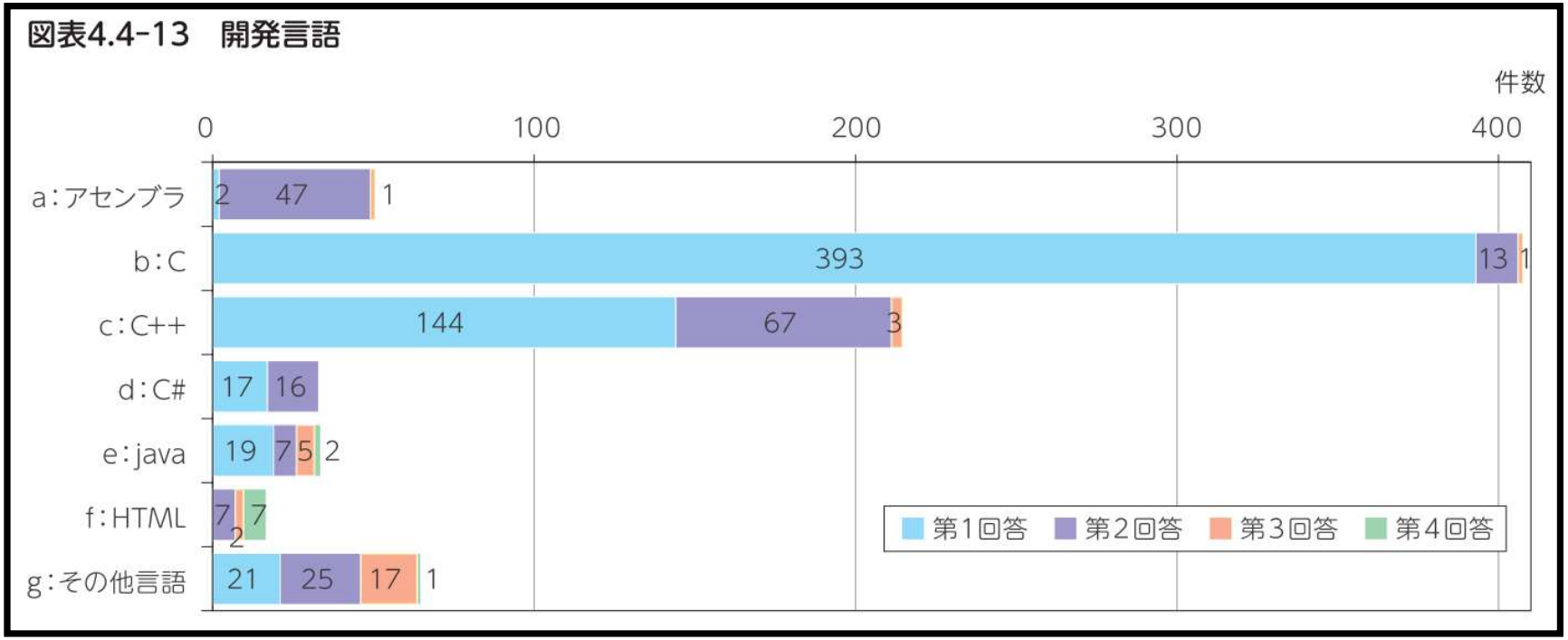


https://spectrum.ieee.org/top-programming-languages-2022



現状把握

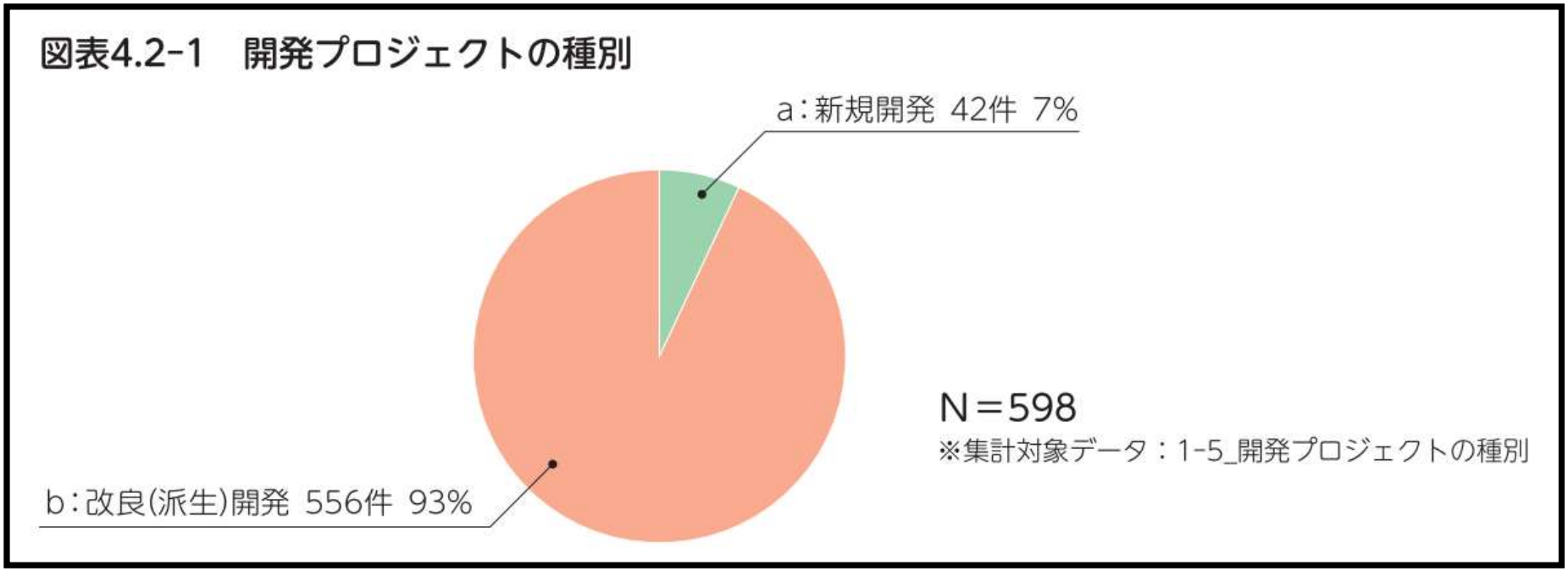
- 約2/3のプロジェクトで、Cが主力言語
- 次がC++、補助的な利用を含めると次がアセンブラ



組込みソフトウェアデータ白書2019　独立行政法人 情報処理推進機構 2019年

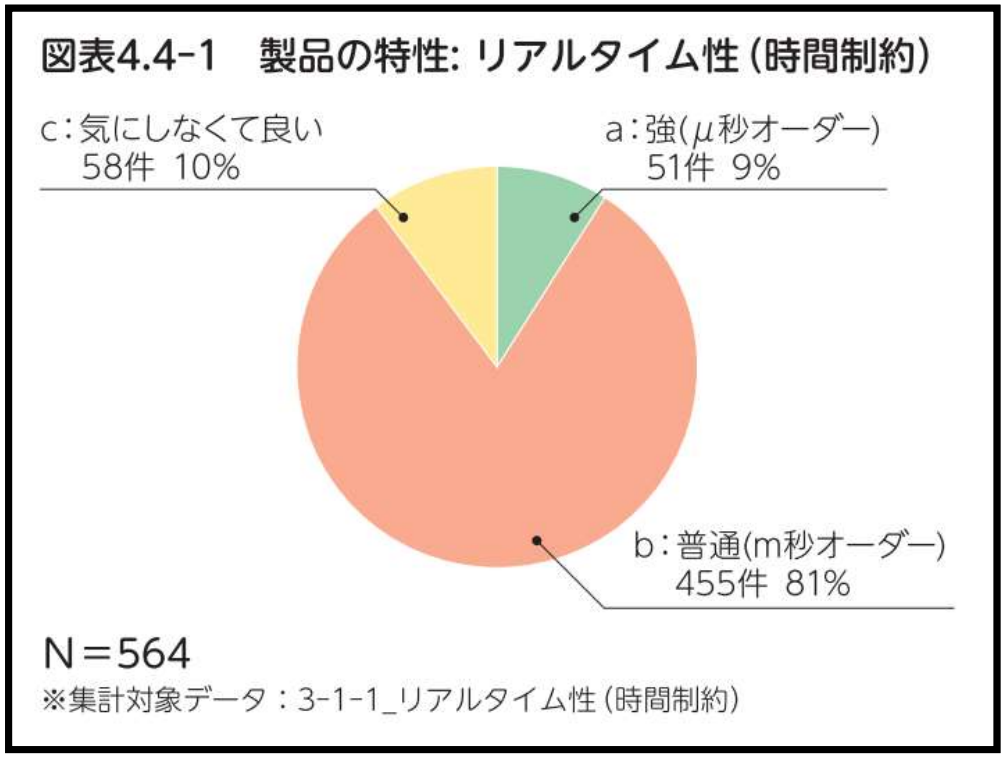
現状把握

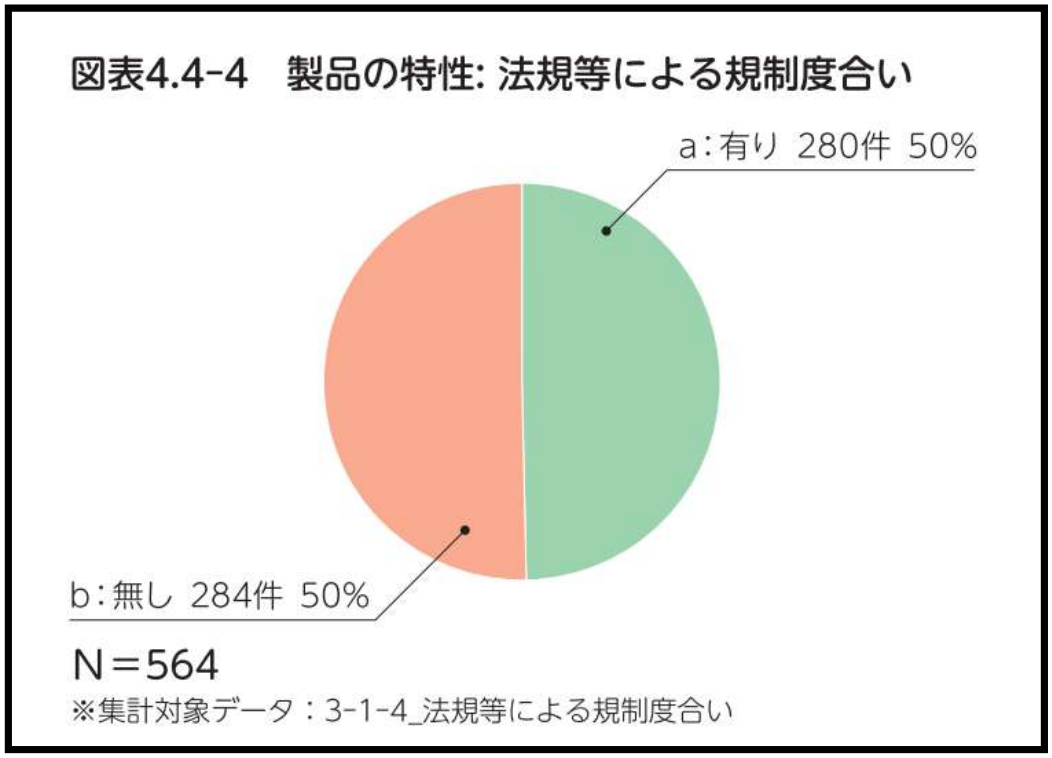
- 新規開発は7%
- 多くの開発プロジェクトは、ベースソフトがある。



組込みソフトウェアデータ白書2019　独立行政法人 情報処理推進機構 2019年

現状把握





組込みソフトウェアデータ白書2019　独立行政法人 情報処理推進機構 2019年

現状把握（まとめ）

CもしくはC++は、以下の理由で使われ続けているようです。

- 組込みソフト開発は流用することが多く、流用元が C で書かれているから、そのまま C を使っている。
- 組込み開発の成果は、「モノ」として販売される側面があり、計算力をむやみに増やすことが難しい。リアルタイム制約も厳しく、小回りが効く言語を使うことは合理性がある。
- 法規制や機能安全規格などへの対応を考えると、ナレッジが集約されたガイドラインが整備されていたり、コンパイラを含めた開発ツールが強く品質保証されているのは好ましく、CもしくはC++はこの面では蓄積がある。

Cの現状について

C の現状について、以下の順で解説をいたします。

- 歴史
- ガイドライン（MISRA-C）について
- C99 で導入された機能の紹介
- C11 で導入された機能の紹介

歴C（※1）を振り返る

C言語

今年でちょう50年です。

- 1972年　D. Ritchie がコンパイラを開発
- 1979年　B. Kernighan と D. Ritchie による公刊書籍発行
- 1989年　ANSI C 標準発行（C89）
- 1990年　ISO/IEC 9899:1990発行（通称 C90）
- 1999年　ISO/IEC 9899:1999発行（通称 C99）
- 2011年　ISO/IEC 9899:2011発行（通称 C11）
- 2018年　ISO/IEC 9899:2018発行（通称 C17※2）

※1「エキスパート C プログラミング」 Linden著、梅原系訳 アスキー 1996年から表現を拝借
※2 2018年発行だが、__STDC_VERSION__ は、201710L に指定されている。

ガイドラインの必要性について

例）C では、if 文の本文は、単文でも複文でもよい。

```
if (a > b)
    a = b;
```

一方、本文を単文で書いた場合、文を追加すると、インデント（見た目）と実際の動作が合わなくなる。

```
if (a > b)
    a = b;
    c = b; /* 必ず実行される */
```

if 文に続く本文は、必ず複文にしておく、というガイドラインを導入しておくと、以下となる。

```
if (a > b) {
    a = b;
}
```

この書き方であれば、本文に文を追加しても意図通り動作します。

MISRA-C

C の好ましくない書き方についてノウハウが集約されており、その成果が MISRA-C としてまとめられた。
当初は、自動車業界を中心に活用されていたが、安全性が必要な分野にも適用されているようです。

- MISRA-C:1998　初版
- MISRA-C:2004　2版
- MISRA-C:2012　3版　現行これが普及している。

MISRA-C のようなガイドラインが確立していることが、C の普及を促している（寿命を延ばしている）と考えています。

車載開発における Cバージョン

- C90（JIS X 3010:1993）
  - 多くの車載開発で現役
  - MISRA C:1998、MISRA C:2004、MISRA C:2012 が対象としている。
- C99（JIS X 3010:2003）
  - 車載でよく採用されている機能の例
    - 1行コメント （// コメント）
    - inline 関数
  - MISRA C:2012 が対象バージョンに加える。
- C11（JIS 未発行）
  - 車載向けのコンパイラは未対応のものが多い。
    ARMコンパイラは対応しているが、COMMUNITY 扱い。
  - MISRA C:2012 Amendment2 が対象バージョンに加える。

制御系の車載ソフト開発は、30年前に仕様策定された
言語仕様に従って開発を行っている。

多くの開発では C90 を採用

- 車載開発では、多くのプロジェクトで C99 以降の機能を使っているとは、言い難い。
- C99 で導入された新機能で、比較的よく使われている機能
  - 1行コメント（// コメント）
  - Inline による関数定義
- C11 は普及しているとは言えない。

C99で導入された機能（1）

- 関数の引数などで指定した個数のローカル配列（可変長配列）が宣言できる。

```
int function(int n)
{
    int data[n];
    /* … */
}
```

  - この機能を使うと、想定した配列の最大容量を超えてしまうリスクが減る。
  - スタック使用量が最適化できる。
  - この機能の導入により sizeof 演算子の評価タイミングがコンパイル時（C90）から実行時（C99）に変更された。

C99で導入された機能（2）

- for 文の制御式の中で変数が宣言できる。

```
for (int i = 0; i < N; i++) {
    /* … */
}
```

  - for 文で使うループカウンターのスコープを限定できる。for 文からはずれた場所で参照ができないので安全と言える。
  - C++ では初期から、この書き方が導入されているため、この書き方が普及している（逆に関数冒頭でループカウンタを定義することはあまり見られない）。C としては、あまり見ない。

C99で導入された機能（3）

- ローカル変数の宣言をブロックの先頭で行う必要はない。

```
int function(void)
{
    int data1;
    /* data2 はここで使わない */
    /* … */

    int data2;
    /* data2 はここから使う */
    /* … */
}
```

  - 変数を参照する直前に宣言ができるため、書き方を工夫すれば、理解しやすいコードが実現できる。

2007年発行の書籍から引用

「Lepton先生のCの強化書」から引用

間瀬の感覚もほぼ同じ。

C89制定後
やっとCの標準規格ができたかぁ。これでコンパイラごとに仕様の差によって泣かされることも減るかなぁ。

私の経験と記憶によれば、1995年頃までにはK&R Cでの開発はほぼなくなり、新規開発についてはC89準拠に移行したと思います。

C99制定後
おお、Cの新しい規格かあ。便利そうな機能もあるな。でもいまのCに慣れちゃってるし、別にどうしても新機能を使わないと書けないものでもないしなぁ。なんなら他の便利な言語を使ってもいいし。

ごく一部の新機能を除いては、ほとんど使われていない。

C11で導入された機能

- アライメントの仕様
- _Noreturn 関数指定子
- _Generic キーワードを使用した型
- マルチスレッドのサポート（pthread に近い）
- 境界チェックインターフェース
- 無名構造体と無名共用体
- 静的アサーション

C11で導入された機能（続き）

- 2020年2月発行の MISRA C:2012 Amendment 2 が発行されて、C11も対象に加わったが、以下のガイドラインが追加された。

  Rule 1.4 Emergent language features shall not be used.

  以下の機能は、ガイドラインとして非推奨にしている。
  - アライメントの仕様
  - _Noreturn 関数指定子
  - _Generic キーワードを使用した型
  - マルチスレッドのサポート
  - 境界チェックインターフェース
- C11で導入された境界チェックインターフェースについては、拙速な仕様であるとの意見がでていて、次の版で、非推奨または削除するという提案がでている。
  https://www.open-std.org/jtc1/sc22/wg14/www/docs/n1969.htm

C の現状について（まとめ）

- C はISO化以前は、方言が多くて、業界として困っていた。よいタイミングでの ISO化と考えられていた。
  一方、C90 の機能で満足してしまった。
- C99/C11 でいくつかの機能が導入されたが、C++コメント（//コメント）、インライン関数以外は、ほとんど使われていないのが実情
- 言語仕様が安定していることは、長期間で使う場合には、メリットとも考えられる。
- アセンブラ言語に近い立場で、生き残っていくと考えています。

C++の現状について

Cは、ISO（言語仕様）の改訂が行われているものの、業界として、新しい機能の取り込みに積極的とは言えない。

一方、C++ の現状について、以下の順で解説をいたします。

- C++ 紹介（メリット、デメリット）
- C++ のガイドライン
- C++11 で導入された機能の紹介
- C++14 で導入された機能の紹介

C++の歴史

- 1983年　Bjarne Stroustrup が公開
- 1998年　ISO/IEC 14882発行（C++98）
- 2003年　C++03　C++98と同一視することが多い。
- 2011年　C++11
- 2014年　C++14　AUTOSAR Adaptive Platform で採用
- 2017年　C++17
- 2020年　C++20
- 2023年　C++23　審議が開始されている。

新しい実装技術に対応するために、言語仕様自体を
3年毎に改訂する計画になっている。

C++導入のメリット

- オブジェクト指向が導入しやすい。
- 型については、言語文法として C より厳密になっている側面がある。
- C は、配列と構造体くらいしかないが、C++ は、連想配列（map）や stack、queue、などが標準として用意されている（STL : Standard Template Library）。
- 新しい言語の考え方が導入されている（特にC++11以降）。
- 多くの C コンパイラは、C++ コンパイラとしても使用可能
- ガイドラインも整備されている。

C++の負の面

言語仕様が巨大

- Stroustrup が記した “C++ Programing Language Third Edition” 1997年発行でも、910ページある。
- ISO 化される前にもっとも権威があった「注解C++リファレンスマニュアル」（通称 ARM） も596ページある。
- 参考）K&R 付録AがARMと同じ密度で文法仕様を記載しており、ページ数は69ページ

  単純に考えて、言語仕様が8倍ある。
- C11（N1570）は、701ページ、C++14（N4140）は、1365ページ、ライブラリを含むため、言語仕様の解説より差は少ないが、それでも倍以上の分量がある。

ガイドライン　MISRA C++:2008

- Guidelines for the use of the C++ language in critical systems
- 2008年7月 MIRA Limited 発行
  - 初版から改版されていない。
- C++のベースは、C++03 （ISO/IEC 14882:2003）
- 228 ガイドライン
  - MISRA C:2004　141 ガイドライン
  - MISRA C:2012　173 ガイドライン
  - C++の言語仕様の大きさがガイドライン数に反映されている。
  - C と共通のガイドラインも多いが、記述内容は2004年版と2012年版の中間と読み取れる記述が見られる。
- MISRA Forum によると、新しい版がレビュー中

ガイドライン　MISRA C++:2008

- ガイドラインのカテゴリ
  - Required　203ガイドライン
    - 準拠を強く求めているガイドライン
    - 逸脱する場合は、正式な逸脱手続きが必要
  - Advisory　16ガイドライン
    - 準拠することを推奨しているガイドライン
    - 逸脱する場合でも、正式な逸脱手続きを求めていない。
  - Document　9ガイドライン
    - 文書化を求めているガイドライン
    - 逸脱することは認められていない。

参考
MISRA C:2004
Required と Advisory の2段階
MISRA C:2012
Mandatory と Required と Advisory の3段階

ガイドライン　MISRA C++:2008

- Rule Number のつけ方
  - aa-bb-cc の形式をとる。
  - aa-bb は、ISO/IEC 14882:2003 の章立てに従っている。
  - 0-bb は、規格の特定記述に紐づかないガイドラインが該当する。
  - aa-0 は、section に紐づくが、それより細かい記述に紐づかないガイドラインが該当する。
- AUTOSAR C++14 もこのRule Number のつけ方を踏襲している。

ガイドライン　MISRA C++:2008

0 Language independent issues
1 General
2 Lexical conventions
3 Basic concepts
4 Standard conversions
5 Expressions
6 Statements
7 Declarations
8 Declarators
9 Classes
11 Member access control
12 Special member functions
14 Templates
15 Exception handling
16 Preprocessing directives
17 Library introduction
18 Language support library
19 Diagnostics library
27 input/output support

ガイドライン　AUTOSAR C++14

- Guidelines for the use of the C++14 language in critical and safety-related systems
  - R17-03/R17-10/R18-10/R19-03 が発行されている。
- C++のベースは、C++14 （ISO/IEC 14882:2014）
- いくつかのガイドラインを参照しているが、ベースとしているのは、MISRA C++
- Rule Number のつけ方
  - MISRA C++ を踏襲したもの Maa-bb-cc の形式
  - MISRA C++ を変更したもの、もしくは新設したものは、Aaa-bb-cc の形式

ガイドライン　AUTOSAR C++14 感想

- MISRA C++ もガイドラインとしては、まとまっているが、AUTOSAR C++14 は、MISRA C++ をベースとしているが、より良くなっている印象を受けた。
  - 遵守することによるメリットが少ないガイドラインのいくつかは廃止された。
  - カテゴリの見直し（Required ←→ Advisory）
  - C++11/C++14の新機能を積極的に取り入れている。（後述）

C++11 で導入された機能

- 以下 URL 記事から抽出した
  https://cpprefjp.github.io/lang/cpp11.html
- 一般的な機能（cpprefjp の区分）
  - Auto（型推論）
  - Decltype
  - 範囲for文
  - 初期化子リスト
  - 右辺値参照・ムーブセマンティクス
  - ラムダ式
  - Noexcept
  - Constexpr
  - Nullptr
  - インライン名前空間
  - ユーザー定義リテラル
- 上記に含まれていないが、組込みでも関係すると思われる機能
  - テンプレートの右山カッコ
  - コンパイル時アサート

C++11 は、大きな
レベルの仕様追加が多く
影響が大きい機能を
中心に紹介する。
（赤字の機能）

Auto（型推論）

- 変数宣言時に具体的な型名のかわりに auto キーワードを指定する事によって、変数の型を初期化子から推論できるようになった。

```
auto i = 0;                         // i は int 型
const auto l = 0L;                  // l は const long 型
auto& r = i;                        // r は int& 型
auto s = "";                        // s は const char* 型
```

- 上記の例では、それほどメリットを感じないが、STL（Standard Template Library）を使うと、型の記述が長くなる傾向があり、その場合はメリットが大きい。

Auto（型推論）　続き

- STL を使ったコードの例

```
std::map<int, int> T;
// T を格納
for (std::map<int, int>::iterator it = T.begin(); it != T.end(); it++) {
    // *it の処理
}
```

- auto を使うと以下になる。

```
std::map<int, int> T;
// T を格納
for (auto it = T.begin(); it != T.end(); it++) {
    // *it の処理
}
```

```
/* 本質的には以下と同じ */
for (int i = 0; i < N; i++) {
    /* array[i] の処理 */
}
```

Auto（型推論）　続き

コーディングガイドラインの扱い

- Rule A7-1-5 (required)

  The auto specifier shall not be used apart from following cases:

  (1) to declare that a variable has the same type as return type of a function call,

  (2) to declare that a variable has the same type as initializer of non-fundamental type,

  (3) to declare parameters of a generic lambda expression,

  (4) to declare a function template using trailing return type syntax.

Auto（型推論）　続き

- Example から

```
auto x1 = 5;         // Non-compliant
auto x2 = 0.3F;      // Non-compliant
auto x3 = {8};       // Non-compliant

std::vector<std::int32_t> v;
auto x4 = v.size();                                    // Compliant
auto lambda1 = []() -> std::uint16_t { return 5U; };  // Compliant
```

- Rationale には、fundamental type の場合に型推論を使わない理由は明記されていないが、メリットが少ない、と判断したのかもしれない。
- ガイドラインでは、禁止する場合を記しているが、積極的に活用していきたいと考えている。
  - 関数型言語では、型推論の活用が前提となっており、型推論の精度があがっていると考えられている。

Decltype

- オペランドで指定した式の型を取得する機能である。

```
int i = 0;
decltype(i) j = 0;     // j は int 型
decltype(i)* p = &i;   // p は int* 型
decltype((i)) k = i;   // k は int& 型（変数名 i の周りの余分な丸括弧に注意）
```

- auto があれば、decltype は必要ないように感じるが、関数テンプレートの戻り値でこの機能が必要

  https://cpprefjp.github.io/lang/cpp11/decltype.html

- ガイドラインでは、推奨も禁止もしていない。auto と decltype の両方を導入すると混乱するかもしれない。上述のユースケース以外は、auto での記述を推奨とするのがよいと考えている。

範囲for文

- 範囲for文（The range-based for statement）は配列やコンテナを簡潔に扱うためのfor文の別表現である。

```
std::vector<int> v;

/* v に値を代入する */

for (const auto& e : v) {
  std::cout << e << std::endl;
}
```

- ガイドラインでは禁止していない。A6-5-1のRationaleには、”that reduces the amount of code”との記述があり、範囲for文を推奨していると受け取れる記述がある。

初期化子リスト

- ユーザー定義型のオブジェクトに対して、波カッコによるリスト初期化を使用できるようにするようオーバーロードする機能である。

```
int ar[] = {1, 2, 3};
std::vector<int> v1 = {1, 2, 3};
std::vector<int> v2 {1, 2, 3};
```

- 変数の初期化は以下の方法でできる。

```
int a(0);       // (1)
int b = 0;      // (2)
int c{ 0 };     // (3)
int d = { 0 };  // (4)
```

- ガイドラインでは推奨する記法はない。Scott Meyers “Effective Modern C++” では、使える局面が多い (3) を推奨している。

右辺値参照・ムーブセマンティクス

- ムーブセマンティクスはコピーコストの削減を主な目的としており、また所有権の移動を実現する。
- 後ほど、スマートポインタと合わせて解説します。

ラムダ式

- 簡易的な関数オブジェクトをその場で定義するための機能である。

```
auto plus = [](int a, int b) { return a + b; };
int result = plus(2, 3); // result == 5
```

- C++の STL では多くのアルゴリズムライブラリが定義されており、関数を引数に取るものが多く、これらのアルゴリズムを使いやすくするための言語サポートとして導入された。

ラムダ式（続き）

コーディングガイドラインの扱い

- Rule A5-1-8 (advisory)
  Lambda expressions should not be defined inside another lambda expression.
  入れ子では使わない（複雑になりすぎる）。
- Rule A5-1-9 (advisory)
  Identical unnamed lambda expressions shall be replaced with a named function or a named lambda expression.
  名前なしラムダ式は、使いまわさない（少し読みにくい）。
- ラムダ式を禁止しているわけではない。

Noexcept

- 用法その1
  - 関数がどの例外を送出する可能性があるかを列挙するのではなく、例外を送出する可能性があるかないかのみを指定する。例外を送出する可能性がある関数にはnoexcept(false)を指定し、例外を送出する可能性がない関数にはnoexcept(true)もしくはnoexceptを指定する。

```
class Integer {
  int value_ = 0;

public:
  // getValue()メンバ関数は、例外を送出しない
  int getValue() const noexcept
  {
    return value_;
  }
};
```

Noexcept

- 用法2
  - 式が例外を送出する可能性があるかどうかを判定する演算子

```
Integer x;
static_assert(noexcept(x.getValue()), "getValue() function never throw exception");
```

- 関数宣言で noexcept が付いていると、例外を送出しないことを表すことができ、最適化できた実装が選択できる。
- ガイドラインは例外送出しない関数に対して、noexcept を付けることを要求している。
- Rule A15-4-4 (required)

  A declaration of non-throwing function shall contain noexcept specification.

Constexpr

- 汎用的に定数式を表現するための機能である。コンパイル時に値が決定する定数、コンパイル時に実行される関数、コンパイル時にリテラルとして振る舞うクラスを定義できる。

```
constexpr int square(int x)
{
  return x * x;
}
```

- 上述の square は、コンパイル時に値が決まるのであればコンパイル時に呼びだされ、そこで決まらなければ実行時に呼びだされる。

Constexpr（続き）

コーディングガイドラインの扱い

- Rule A7-1-1 (required)

  Constexpr or const specifiers shall be used for immutable data declaration.

- Rule A7-1-2 (required)

  The constexpr specifier shall be used for values that can be determined at compile time.

- ガイドラインでは、constexpr を使うことを促している（禁止していない）。

Nullptr

- nullptr は、ヌルポインタ値を表すキーワードである。
- 歴史的な事情
  - C++では、void * から他の型へのポインタ代入はキャストを必要とした（C では必要ない）。
  - そのため NULL は C と C++ で異なる定義となった。

```
#define NULL (void *)0  /* C */
#define NULL 0          // C++
```

- 一方、NULL を 0 とすると、型推論がうまく働かない。

```
auto pTest = NULL;   // pTest は int
```

- ライブラリではなく、言語仕様としてキーワード nullptr が導入された。

Nullptr（続き）

MISRA C++ の扱い

- Rule M4-10-1 (required)
  NULL shall not be used as an integer value.
- Rule M4-10-2 (required)
  Literal zero (0) shall not be used as the null-pointer-constant.

AUTOSAR C++ で以下のガイドラインが追加された。

- Rule A4-10-1 (required)
  Only nullptr literal shall be used as the null-pointer-constant.

言語仕様とガイドラインと合わせて、すっきりとした。

インライン名前空間

- 名前空間内の機能に透過的にアクセスするための機能である。inline namespaceによって定義した名前空間の機能には、その名前空間を指定しなくてもアクセスできる。

```
namespace my_namespace {
  inline namespace features {
    void f() {}
  }
}

int main()
{
  my_namespace::features::f();
  my_namespace::f();           // features名前空間は省略できる
}
```

- ガイドラインでは推奨も禁止もしていない。

ユーザー定義リテラル

- リテラルに対して付けられるサフィックスをオーバーロードできるようにすることで、ユーザーがリテラルに意味を持たせられるようにする機能である。

```
std::string operator"" s(const char* str, std::size_t length)
{
  return std::string(str, length);
}

auto x = "hello"s; // xの型はstd::string
```

- AUTOSAR C++ R17-03 では、当初禁止であった。
- Rule A13-1-1 (required)
  User-defined literals shall not be used.
- R18-03 では、A13-1-1 が削除された。経緯は不明

テンプレートの右山カッコ

- C++03では、2つ以上連続する右山カッコが出現する場合には、間にスペースを入力する必要があった

```
// C++03 の場合
vector<basic_string<char> >; // OK
vector<basic_string<char>>;  // コンパイルエラー : >>は右シフト演算子と見なされる
```

- C++11から
  - 左山カッコがアクティブな場合(対応する右山カッコがまだ現れていない場合)、丸カッコ内を除いて、>>は右シフト演算子ではなく2つの連続する右山カッコとして扱われる。
  - この仕様により、C++03とC++11で異なるふるまいをするプログラムは存在する。
- ガイドラインでは推奨も禁止もしていない。

コンパイル時アサート

- static_assert宣言は、指定した定数式が真であることを表明するための機能である。
- static_assert宣言では、新たな型やオブジェクトは宣言しない。また、実行時にサイズや時間コストは発生しない。

```
template <class T, std::size_t N>
struct X {
  static_assert(N > 0, "number of array elements must greater than 0");
  T array[N];
};
```

コンパイル時アサート（続き）

コーディングガイドラインの扱い

- Rule A14-1-1 (advisory)
  A template should check if a specific template argument is suitable for this template.
  テンプレートのチェックでは、static_assert が使える（前スライドのコード例）。
- Rule A16-6-1 (required)
  #error directive shall not be used.
  #error ではなく static_assert を使うことを推奨している。
  A16-0-1 でも #error は含まれていない。

C++14 で導入された機能

- 以下 URL 記事から抽出した
  https://cpprefjp.github.io/lang/cpp14.html
- C++14 は、C++11のマイナーアップデートと捉えられている。
- 一般的な機能（cpprefjp の区分）
  - 2進数リテラル
  - 通常関数の戻り値型推論
  - Decltype(auto)
  - 後置戻り値型をプレースホルダーにすることを許可
  - ラムダ式の初期化キャプチャ
  - ジェネリックラムダ
  - 変数テンプレート
  - constexprの制限緩和
  - 宣言時のメンバ初期化を持つ型の集成体初期化を許可
  - ネストする集成体初期化における波カッコ省略を許可
  - [[deprecated]]属性
  - 数値リテラルの桁区切り文字
  - サイズ付きデアロケーション
  - 動的メモリ確保の省略の許可

細かな仕様追加が多く
AUTOSAR C++14
組込み開発に
影響が大きい
機能のみ紹介する。
（赤字の機能）

2進数リテラル

- 整数リテラルのプレフィックスとして0bもしくは0Bを付けることで、2進数を表す値を記述できる。

```
int x = 0b1010; // 2進数の値1010を表す。xの値は、10進数の値10となる
```

- Java、Pythonで同じ構文が採用されている。
- ガイドラインでは推奨も禁止もしていない。

通常関数の戻り値型推論

- 関数宣言の構文において、先頭の戻り値型をautoもしくはdecltype(auto)とすることで、戻り値の型が関数のreturn文から推論される。

```
// 関数f()の戻り値型はint
auto f()
{
  return 42;
}
```

- ガイドラインでは禁止している。（3.1の記述より）

ジェネリックラムダ

- C++11のラムダ式を拡張して、パラメータにテンプレートを使用できるようにした機能である。

```
auto plus = [](auto a, auto b) { return a + b; };
```

- Template と同じような機能と思えるが事実、以下のような関数呼び出し演算子を持つ関数オブジェクトを生成する。

```
template <class T1, class T2>
auto operator()(T1 a, T2 b) const
{
  return a + b;
}
```

- ガイドラインでは禁止していない。（前述A7-1-5 （3））

変数テンプレート

- 変数定義時のテンプレート指定を可能にする。

```
template <class T>
constexpr T pi = static_cast<T>(3.14159265358979323846);
```

- 従来であれば、テンプレート関数 pi() を定義して対応していたが、コピーのコストが高いものについては、関数テンプレートよりも、変数テンプレートを使用した方が効率がよくなる。
- ガイドラインでは推奨も禁止もしていない。

数値リテラルの桁区切り文字

- 整数リテラルと浮動小数点数リテラルには、途中にシングルクォーテーション(')を入力することで、値を読みやすくできる。桁区切り文字は、数値を読みやすくするためだけにあり、それによる値への影響はない。

```
int price = 1'000'000; // 100万円
int binary_value = 0b1000'1111;
```

- 区切り文字案としては、カンマ、アンダースコアーなどもあったが、既存仕様と両立しなかった。
  https://cpprefjp.github.io/lang/cpp14/digit_separators.html

数値リテラルの桁区切り文字（続き）

- Rule A13-6-1 (required)

  Digit sequences separators ’ shall only be used as follows:
  (1) for decimal, every 3 digits,
  (2) for hexadecimal, every 2 digits,
  (3) for binary, every 4 digits.
- 使い方を規定している（使うことを強制しているわけではない）。(2)は多少違和感がある。16進数は、2または4桁区切りにする手もあるのでは。

C++11/C++14 まとめ

- C++11/C++14 は、以下の観点で優れた仕様変更を多く含んでいると評価している。
  - 現代的な言語仕様の取り込み
  - 可読性の向上
  - 性能の最適化
- ガイドラインも新機能について、好意的に取り扱っている場合が多い（C の場合と対照的）。
- 一方、もともと大きな言語仕様が更に大きくなってしまった。
  - Stroustrup が記した「プログラミング言語C++ 第4版」（2015年発行）は、1300ページを超えた。

C++11/C++14 の世間動向

- アルゴリズムの教科書は、古典が多く、C++ の場合、C++03 で記載されていることが多かったが、最近のアルゴリズムの教科書では、C++11で記載されているものがでてきた。
- 「問題解決力を鍛える！アルゴリズムとデータ構造」
  - 大槻 兼資 著/秋葉 拓哉 監修
  - 2020年　講談社

  コード例は、C++11 で記述されている。

CがC90に留まっていることと対称的

動的なメモリ管理をどうするか

- 動的なメモリ管理の必要性
  - 静的なメモリモデルでは、実現しにくいソフトウェアが存在する。例えば、Windows GUI開発、オブジェクト指向のデザインパターンも動的なメモリ管理が前提になっているパターンがある。
  - Cでは、標準ライブラリで、malloc/freeを提供している。
  - C++では、言語仕様として、new/deleteを提供している。
- デメリット
  - 動的なメモリ管理では、メモリリークが発生する可能性がある。
    - 使い終わったメモリの解放を忘れる。（狭義のメモリリーク）
    - 解放したメモリにアクセスする。
    - 解放したメモリをもう一度解放してしまう。
  - フラグメンテーションによりメモリが獲得できない可能性がある。

動的なメモリ管理をどうするか（続き）

- ガーベージコレクション
  - メモリリークを避けるために、プログラマがメモリを管理するのではなく、使われていないメモリを環境側で判断して解放する仕組みが取り入れられた。この機能をガーベージコレクション（Garbage Collection; 以下、GC）と呼ぶ。
  - 例えば、Javaには、キーワードnewは存在するが、キーワードdeleteは言語に存在しない。メモリの解放は、JVMが行う。
- 現代的な言語は、GCを採用している言語の方が多い。
- ただし、ガーベージコレクションしている間は、通常のプログラムは動作できない場合が多い。例えば、JVMでは、ガーベージコレクションしている間は、すべてのコアで（Java としての）アプリケーションプログラムが実行できない。
- これは、リアルタイム制約が強いシステム（ハードリアルタイム制約を持つシステム）では問題になる。

C++ 言語ガイドラインと動的なメモリ管理

- MISRA C++:2008 Rule
  - Rule 18-4-1 (Required) Dynamic heap memory allocation shall not be used.
- AUTOSAR C++14
  - MISRA C++:2008 Rule 18-4-1 を Reject している。”Dynamic heap memory allocation usage is allowed conditionally” として、3つのガイドラインを導入している。(A18-5-1、A18-5-2、A18-5-3)
    - A18-5-1 (required)

      Functions malloc, calloc, realloc and free shall not be used.

      C-Styleの関数は、型チェックが甘いので使わない
    - A18-5-2 (required)

      Non-placement new or delete expressions shall not be used.

      次スライドへ
    - A18-5-3 (required)

      The form of the delete expression shall match the form of the new expression used to allocate the memory.

      new と delete、new[] と delete[] を混乱しないこと（意訳）

スマートポインタの導入

- A18-5-2 ではスマートポインタの導入を要求している。
- C++03の実装
  - これはT*型のポインタを保持し、ディストラクタ時に自身が所有権を持つメモリが存在すれば、 delete を実行する。
  - 「コピー先にメモリの所有権が移動する」という問題があった。このため普及せず。
- 所有権を持つポインタがただひとつであることを保証する
  unique_ptr
- 複数の所有権を持つポインタ
  shared_ptr
- 循環参照を対策したポインタ
  weak_ptr

スマートポインタはコピーする
のではなく、所有権をムーブする。

スマートポインタの導入（続き）

以下から事例を借用しました。

- https://qiita.com/hmito/items/db3b14917120b285112f

```
#include<iostream>
#include<memory>
class hoge{
private:
   std::unique_ptr<int> ptr;
public:
   hoge(int val_):ptr(new int(val_)) {}
   int getValue()const {return *ptr;}
};

int main(){
   //hogeのコンストラクタでint型を動的に確保しunique_ptrに委ねる
   hoge Hoge(10);

   return 0;
}  //auto_ptr同様、ディストラクタで自動的にメモリ解放
```

実際の C++ 動的メモリ管理運用

C++の動的なメモリ管理を以下の運用を前提として許容する。

- 従来のやり方（new/delete）を使う。ただし、A18-5-2 には違反する。
  - メモリリークを検出できるツールで検出する。
    - ツールの検出能力頼み。年々能力は向上している。
  - メモリリークが起こりにくいコーディングの書き方を推奨する。
    - 例えば、同じレベルで new と delete をするなど
  - フラグメンテーションの対策として、高負荷試験、長時間ランニング試験などの耐久試験を行う。
- C++11以降で定義されたスマートポインタを使う。
  - メモリリークは発生しない。（フラグメンテーションは発生する。）
  - 知っている技術者が少ないという懸念がある。

現代的な感覚では、
スマートポインタに移行するべき

結局、C++を何に使うのか？

- システムの制御によっては、動的なメモリ確保の必要性は少ないかもしれない。
- そもそもその領域では、C++は必要ない（Cで書ける）。
- 一方、以下のような要素が絡んでくると C では記述能力が足りない。
  - GUI
  - 高度なアルゴリズムの応用（画像認識、AI）
  - C++の既存資産が多い分野
  - 動的なコンポーネントの結合

C++ の現状について（まとめ）

- C++は、新しいパラダイムを言語仕様として、積極的に取り入れており、ガイドラインも新しい言語仕様の利用を促している。
- これから、C++ について、以下の利用が考えられる。
  - Better C として活用する。
    - オブジェクト指向を積極的に使う or 使わない。
    - STL を積極的に使う or 使わない。
  - C++11 以降の機能（Modern C++）を積極的に活用する。
    - C++14 も組として使うことはあり得る。
    - C++17/C++20 については、開発プロジェクト次第

最後に

このセッションでは、CとC++の現状について、わたくしの理解を元に紹介しました。

セッション後でも、コメント、お気づきの点がありましたら、Slack でお知らせください。

間瀬

END

・・・ありがとうございました。