Cから学ぶ大切なこと

ノーマル（最初から）

C言語とC++言語は
本当に違うもの？

ハード（応用）

全ての始まり
プログラムはなぜ動く

ベリーハード（より発展的）

C言語は
データ指向なプログラム

点と点を結ぼうという話 1

まず解説していく前に、「これは何でしょうか？」

点と点を結ぼうという話２

「リンゴ？」、「Apple？」どちらも正解です。

点と点を結ぼうという話３

変な質問ですが、大事な話です。

重要な所は、「何をもってして、リンゴ、またはAppleと読んでいるのか？」です。

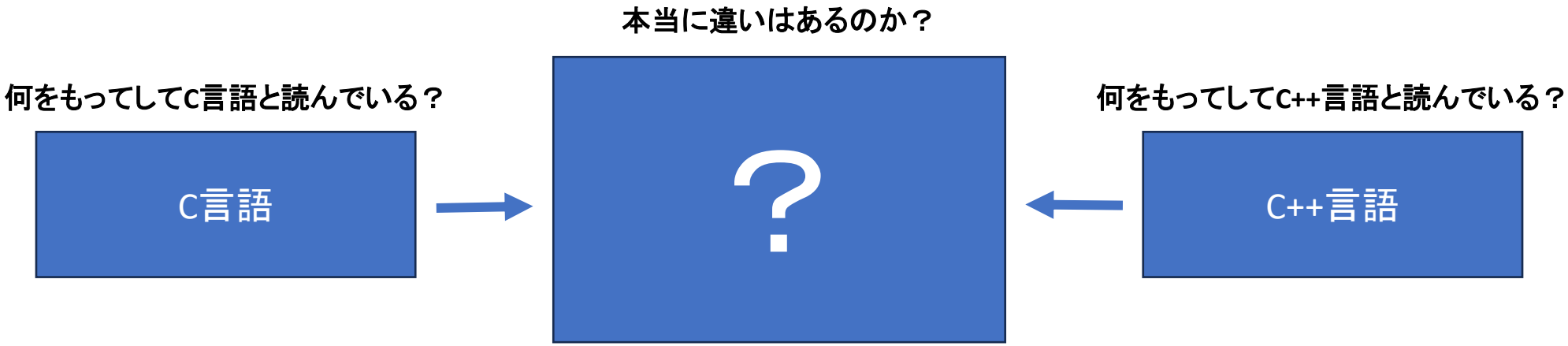
下記の問題は、「赤い丸いもの？」「緑もあるから丸いもの？」、何でしょうね。

点と点を結ぼうという話 4

これは、プログラムでも同様のことが言えます。

この話は「C++にはテンプレートがあって」とかそういう表面上の話ではありません。

もっと抽象的な考えで、「本当の境界線はどこか？」を知りましょうという話です。



もし、CSの知識がある方は、「アセンブリでしょ」とか言うと思いますが、、、

点と点を結ぼうという話５

本当の境界線を知ることで、

C言語とC++言語だけでなく、全てのモノへの理解が高まりますし、履修難易度も格段に下がります。

アニメーション、AI、グラフィック、ネットワーク、サウンド、プログラム設計、プログラム言語も全て。

勿論、個々で細かい計算とかは違いますが、考え方は同じように感じられるはずです。

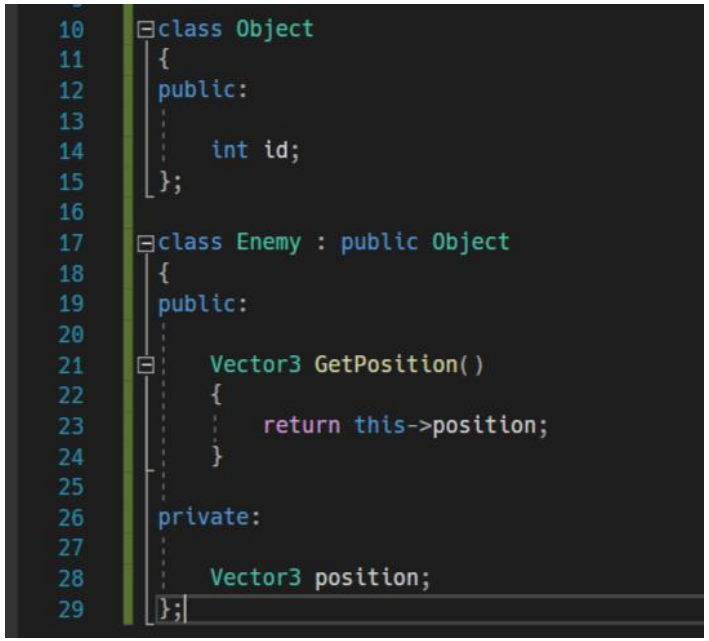
この話は、企画でも同じことが言えそうですが（場外乱闘が起きそうですし）、ここでは割愛します。

C言語とC++言語って本当に違うもの？ 1

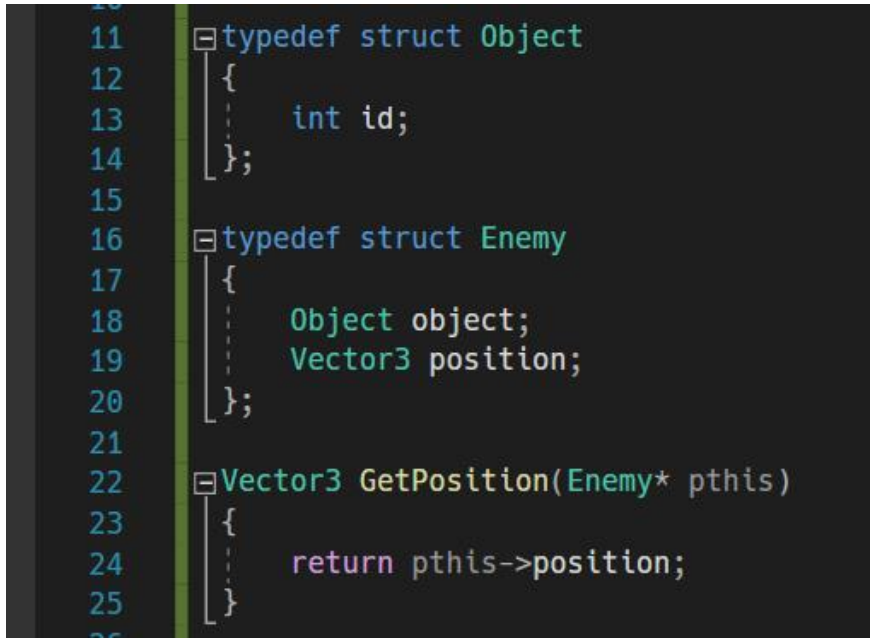
ここからは、C言語とC++言語の違いを見ながら、「抽象化するという考え」を実感してもらいます。

下記は、どちらもクラスを表現していますが、「本当にどちらもクラスなのでしょうか？」。

C++言語でのクラス表現



C言語でのクラス表現



C言語とC++言語って本当に違うもの？ 2

先ほどのモノを見ても、「これはリンゴです。」とリンゴのイラストを渡されたようにしか感じません。

なので、ここから更に使用コードと含めて見てみますが、下記はどちらも同じように見えますね。

C++言語でのクラス表現

```cpp
class Object
{
public:

    int id;
};

class Enemy : public Object
{
public:

    Vector3 GetPosition()
    {
        return this->position;
    }

private:

    Vector3 position;
};

int main()
{
    Enemy* enemy = new Enemy();

    // GetPosition(enemy); と同じ
    enemy->GetPosition();

    // アップキャスト
    Object* object = enemy;

    delete enemy;
}
```

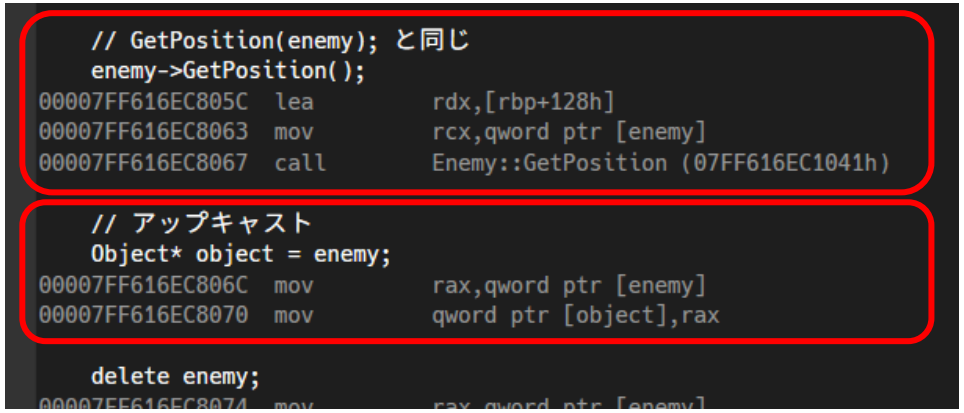
C言語でのクラス表現

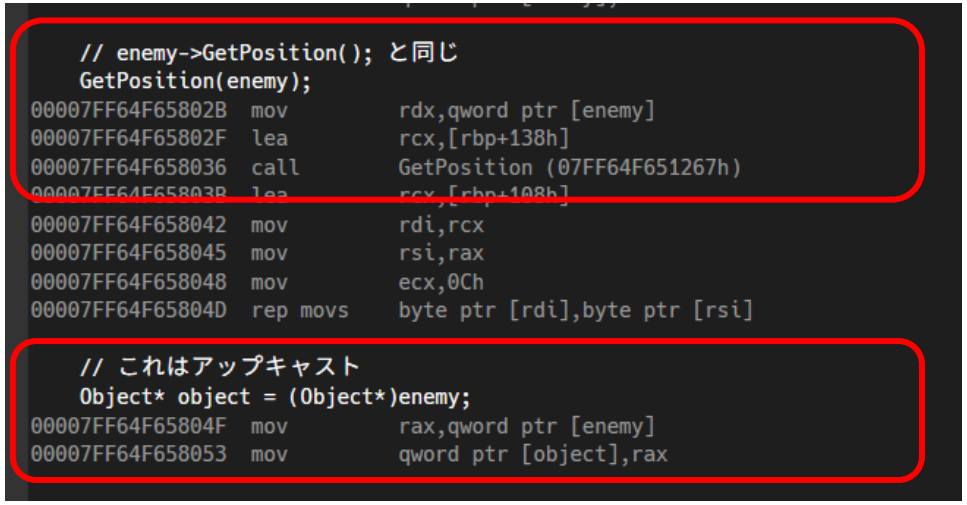
```c
typedef struct Object
{
    int id;
};

typedef struct Enemy
{
    Object object;
    Vector3 position;
};

Vector3 GetPosition(Enemy* pthis)
{
    return pthis->position;
}

int main()
{
    Enemy* enemy = (Enemy*)malloc(sizeof(Enemy));

    // enemy->GetPosition(); と同じ
    GetPosition(enemy);

    // これはアップキャスト
    Object* object = (Object*)enemy;

    free(enemy);
}
```

C言語とC++言語って本当に違うもの？ 3

では、「実際に同じなのか？」ということを見てみましょう。

赤丸で囲っている所は、内容は分からなくても、どちらも同じですよね。





C言語とC++言語って本当に違うもの？ ４

ここまでの話で、

多分、私たちが境界線だと思っていた「クラスがあるのがC++言語」というのが無くなりましたよね。

では、「実際の境界線はどこなのか？」という話ですが、

最も大きな境界線は、「C言語では、スコープというものが無いという所です。」

（無理くりすれば出来るけど、意味が変わる）

そこそこ抽象化出来たと思いますが、ここから面白い所に繋がっていきます。

C言語とC++言語って本当に違うもの？ ５

#仮コード例

```python
# Pythonでは、型を宣言する必要がない
class Object :

    # 変数宣言
    id = 0

# Objectを継承したEnemyクラスです。
class Enemy(Object) :

    # 変数宣言
    _position = Vector3()

    # slef は this と同じ
    def GetPosition(self) :
        return self._position

def main() :
    enemy = Enemy()

    # 毎度お馴染み
    enemy.GetPosition()

    # アップキャスト
    obj = Object(enemy)
```

左はPythonコードです。

Pythonにも、C言語同様スコープという概念がありません。

しかし、先ほどまでの、Cのコードと何が違いますか？

ここまで読んで頂いていれば、同じものだと思います。

C言語とC++言語って本当に違うもの？まとめ

ザックリとでしたが、抽象化し、本当の境界線を知るという感覚は分かって頂けたましたか？

これを知っているだけで、

DirectX11、DirectX12、Vulkan、OpenGLなどといったモノも、どれも同じに感じてくるはずです。

本当の境界線を知ることで、全ての物が繋がり成長の機会が広がります。

私は、この考え方を「本当の意味で身につける」のに3年かかりましたが、皆さんなら大丈夫です。

Cから学ぶ大切なこと（応用）

1. そもそもプログラムはなぜ動く？

プログラムはどうやって動く？ １（ザックリ過ぎる解説）

ここからは、そもそも「プログラムがどうやって動いているのか？」ということを解説します。

今回は触りとなるイメージだけのお話しますが、

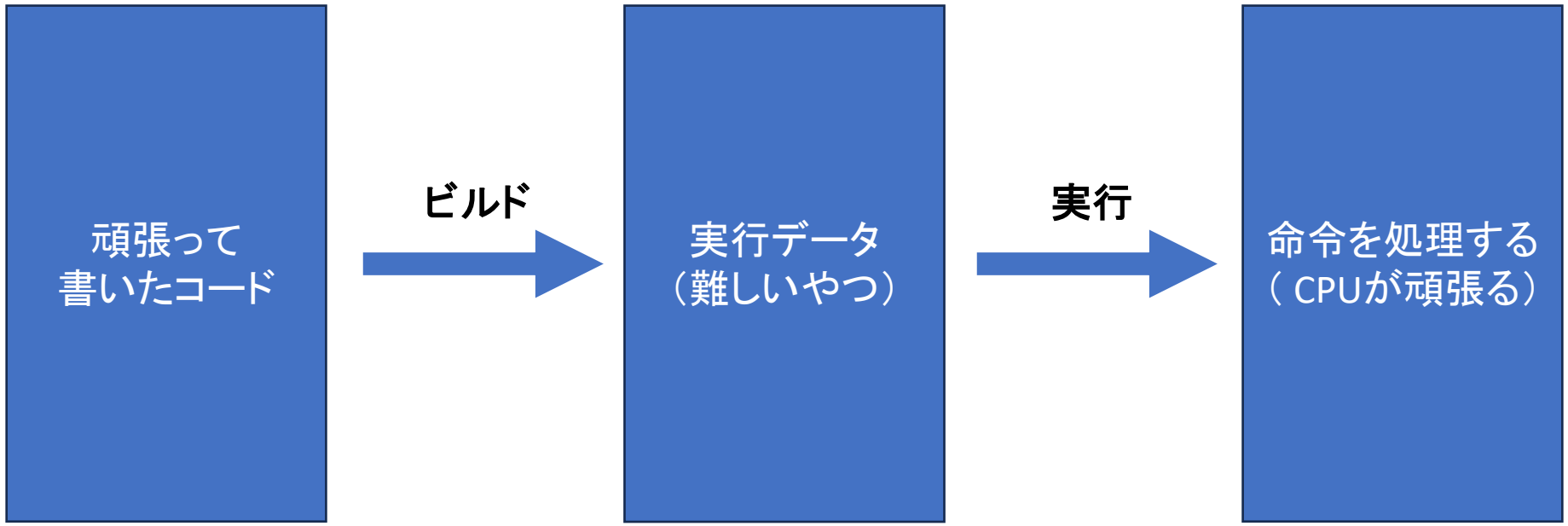
これを理解することでC言語問わず、全ての言語のコードの書き方が変わりますし、

どの言語だろうが、勉強だろうが、履修する難易度が段違いで変わってきます。

プログラムはどうやって動く？ 2（ザックリ過ぎる解説）

下記の図は、コードを書いて、ゲームを遊ぶまでの流れをザックリ図に表したものです。

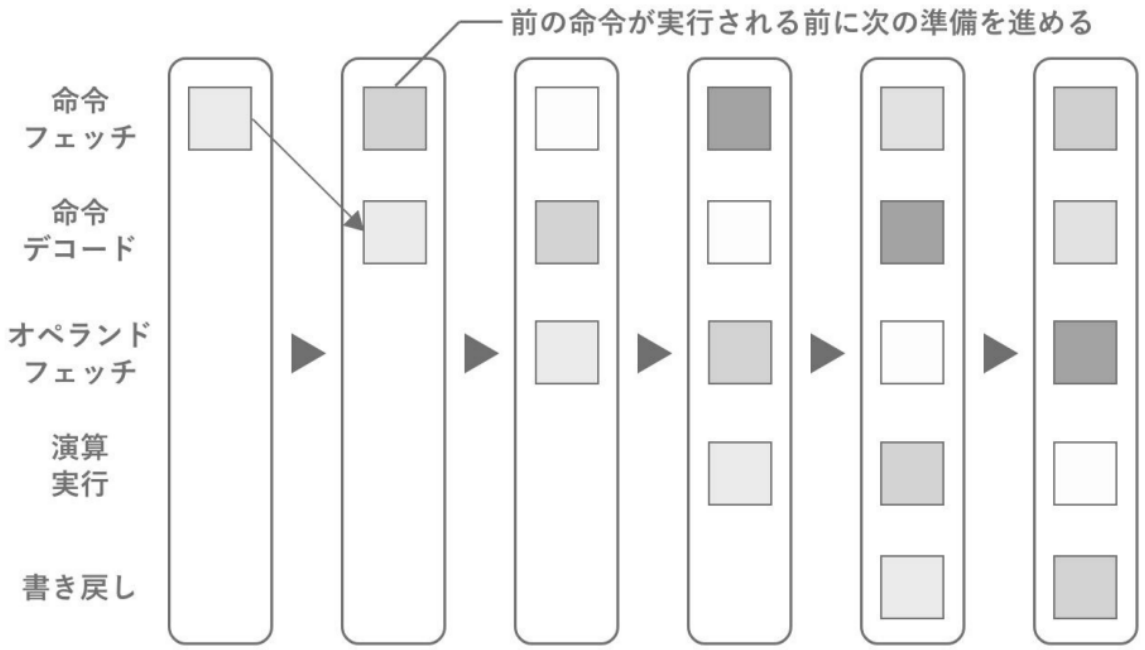
ここから、特に命令を処理する部分について解説していきます。



プログラムはどうやって動く？ 3（ザックリ過ぎる解説）

そして、CPUが命令を処理する流れのことを「パイプライン」と呼びます。

パイプラインでは、次に処理する命令を予測し、準備しながら動いています。

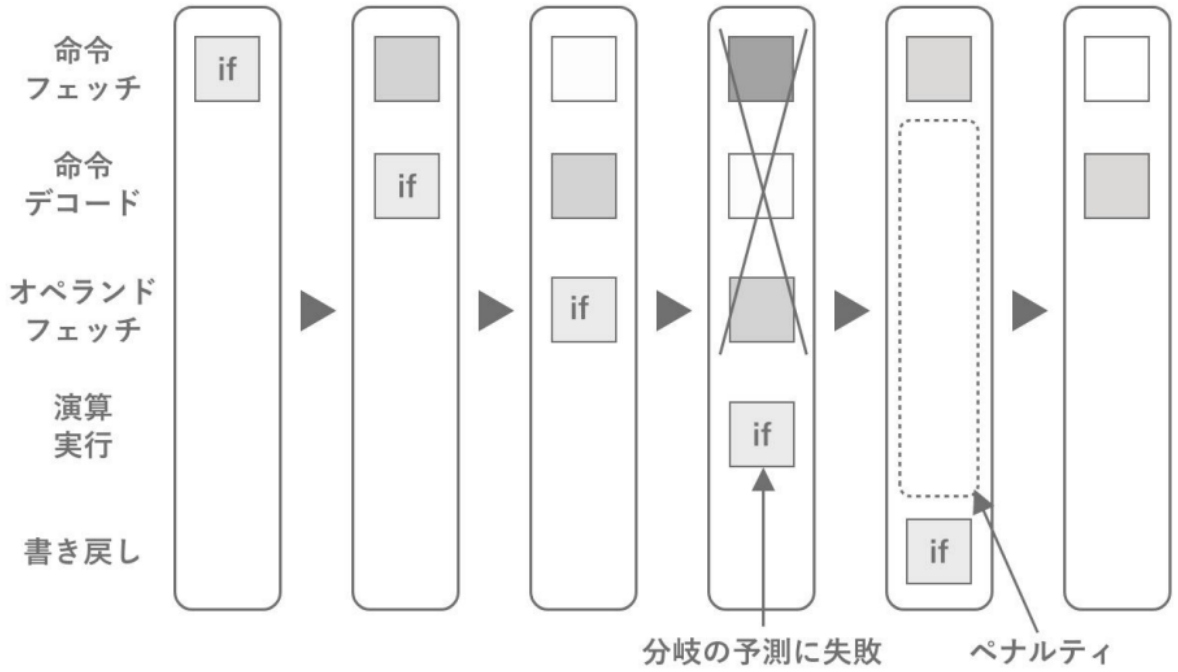


CPUのパイプライン・アーキテクチャ

プログラムはどうやって動く？ ４（ザックリ過ぎる解説）

しかし、分岐処理による予想の失敗など、もし予測されていない場合は、

「パイプライン・ストール」呼ばれる一時停止が発生し、リセットし直されます。



CPUのパイプライン・ストール

プログラムはどうやって動く？５（ザックリ過ぎる解説）

先ほどの例のように、

CPUはアプリケーションが終了するまで、処理の予測、実行を繰り返していきます。

非常にザックリした解説ではありましたが、イメージは出来たのではないでしょうか。

次からは、実際のコードを見ながら更に考えていきましょう。

プログラムはどうやって動く？６（ザックリ過ぎる解説）

最初は、メモリについてです。

スタックとヒープという単語は一度は聞いたことあると思いますが、そちらの解説からです。

まずスタックとは、処理の実行に使用される専用の固定メモリ領域です。

関数の呼び出しなど行った際に、引き数やローカル変数などの分の、メモリ確保時に使用され、

関数が終了し、元の関数へ戻る際にメモリの解放を行います。

また、スタックメモリの容量はとても少なく、限られたデータしか管理しません。

プログラムはどうやって動く？ 7（ザックリ過ぎる解説）

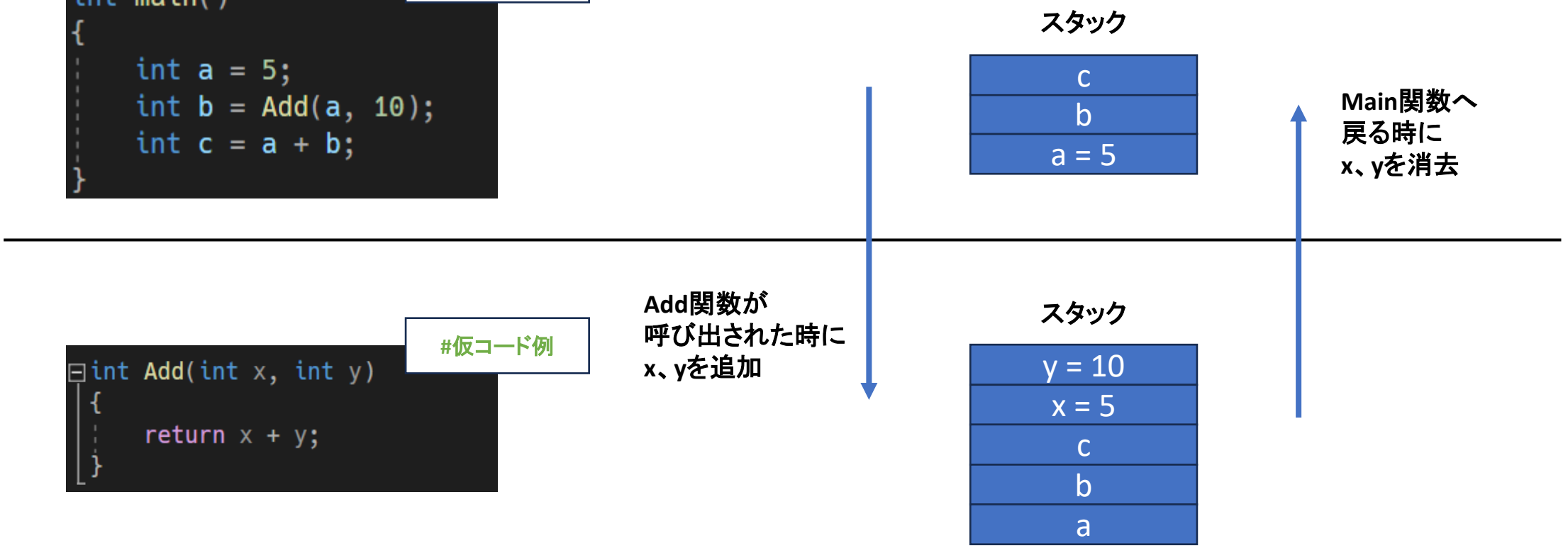
下記プログラムを実行した際の、スタックメモリのイメージ

#仮コード例

```
int main()
{
    int a = 5;
    int b = Add(a, 10);
    int c = a + b;
}
```

#仮コード例

```
int Add(int x, int y)
{
    return x + y;
}
```



プログラムはどうやって動く？ 8（ザックリ過ぎる解説）

次にヒープとは、実行中に自由に使えるメモリ領域です。

プログラムでメモリを確保する命令（C++ならnew）を出すと、メモリ領域を取得出来ます。

勿論、使い終われば、プログラム側でメモリを解放（C++ならdelete）することが必要です。

また、ヒープを使うことで実行時にメモリサイズが変動するデータや、

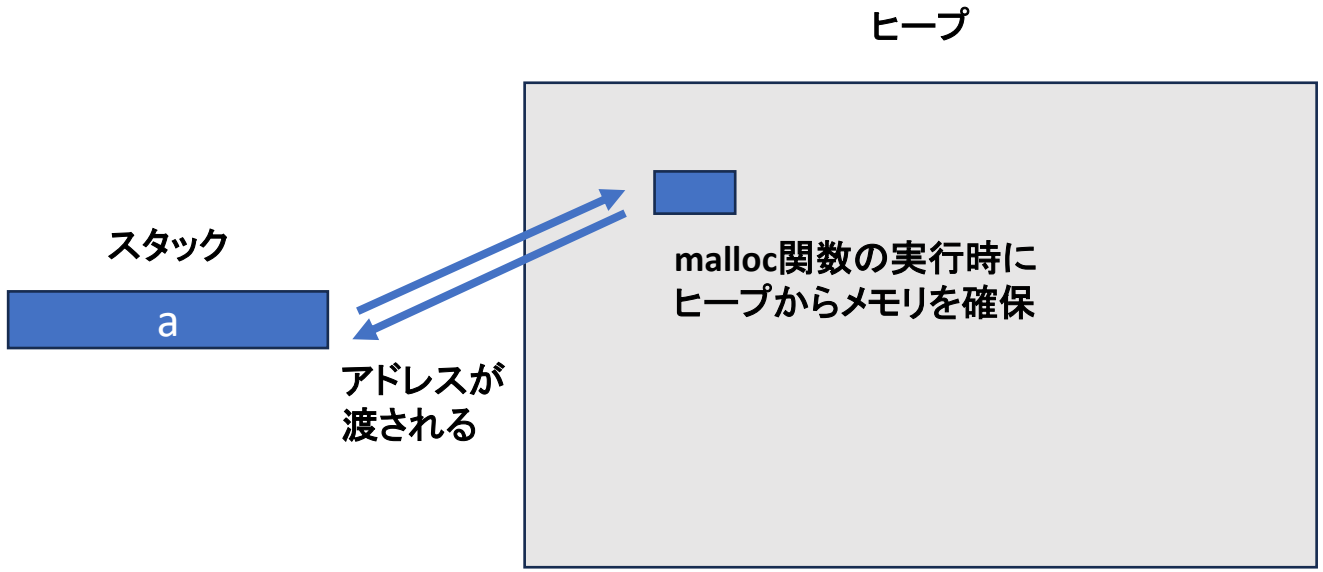
スタックでは扱えないほどの、大容量なデータも扱えるようになります。

プログラムはどうやって動く？ 9（ザックリ過ぎる解説）

下記プログラムを実行した際の、ヒープメモリのイメージ

#仮コード例

```
int main()
{
    int* a;
    a = (int*)malloc(sizeof(int));
    free(a);
}
```

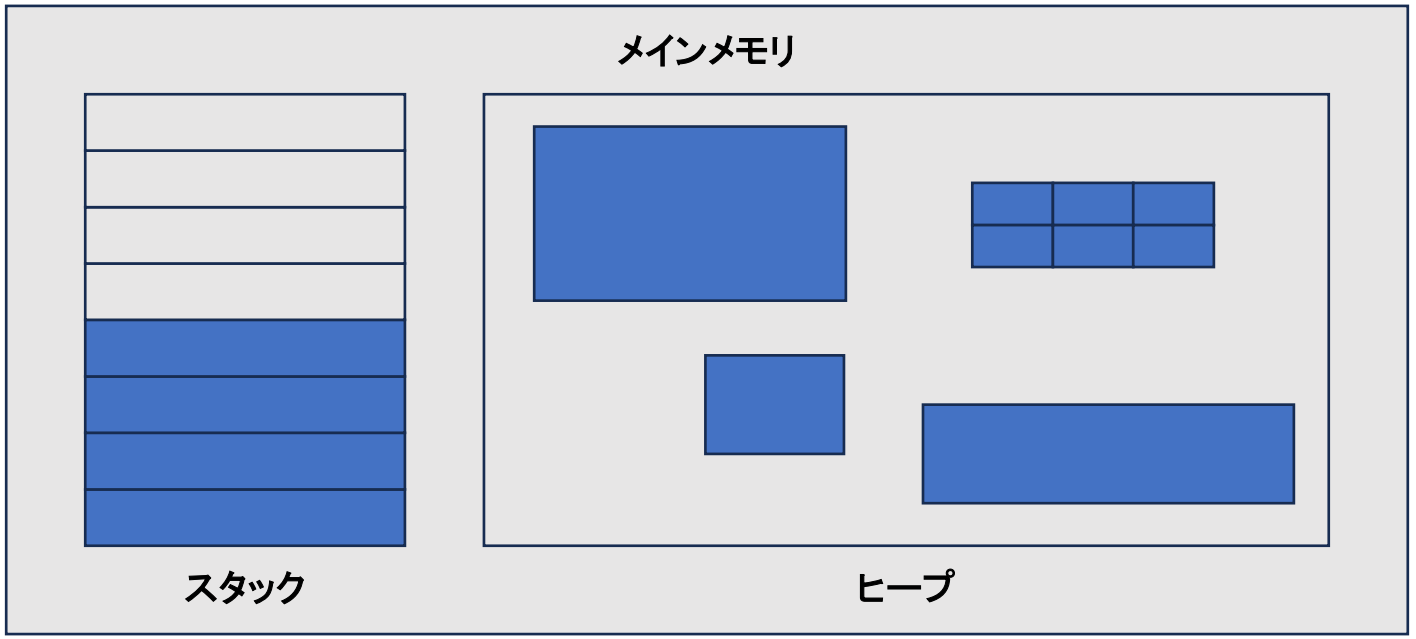


勿論、free関数の実行時に、
ヒープ領域のメモリを解放する。

プログラムはどうやって動く？ 10（ザックリ過ぎる解説）

先ほどまでの、スタックとヒープは、下記の様にメインメモリという場所に所属しています。

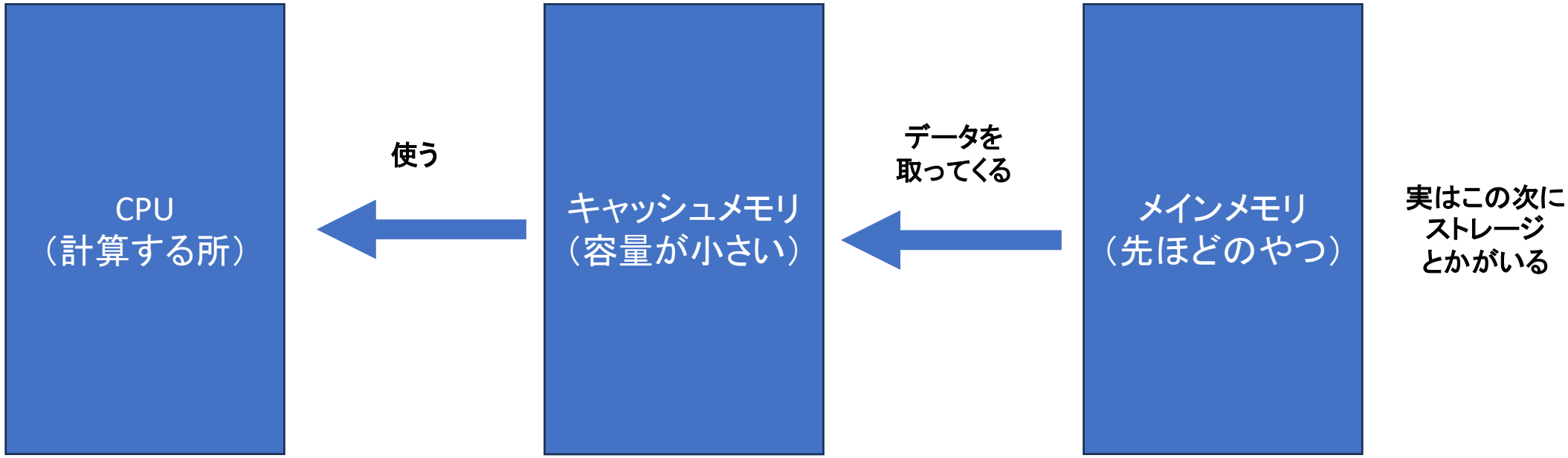
しかし、CPUから、このメインメモリへのアクセスは非常に遅い処理なのです。



プログラムはどうやって動く？ 11（ザックリ過ぎる解説）

そのため、CPUは計算に必要なデータを、予めキャッシュメモリに格納しておき、高速化させています。

もし、キャッシュメモリにデータがなければ、メインメモリに都度取りに行くという感じです。



データを取りに、右に行けば、行くほど遅くなります。

プログラムはどうやって動く？ １２（ザックリ過ぎる解説）

先ほどまでの例のように、

キャッシュメモリやメインメモリでは、メモリの確保と解放が繰り返されていきます。

非常にザックリした解説ではありましたが、イメージは出来たのではないでしょうか。

次はいよいよ動作の流れを見てみましょう。

プログラムはどうやって動く？ 13（ザックリ過ぎる解説）

ここからは下記のコードを元に、

実行中どのようにメモリにアクセスし、処理を実行しているのか（実際の動作の流れ）を見てみましょう。

```
struct Test
{
    int data;
};

int main()
{
    Test* test = NULL;
    test = (Test*)malloc(sizeof(Test));
    test->data = 0;
    free(test);
}
```

プログラムはどうやって動く？ 14（ザックリ過ぎる解説）

```
1
2
3
4
5
    int main()
    {
    00007FF67B482880  push        rbp
    00007FF67B482882  push        rdi
    00007FF67B482883  sub         rsp,108h
    00007FF67B48288A  lea         rbp,[rsp+20h]
    00007FF67B48288F  lea         rcx,[__3D044E8B_Source@cpp (07FF67B49D06Dh)]
    00007FF67B482896  call        __CheckForDebuggerJustMyCode (07FF67B4817BCh)
        Test* test = NULL;
    00007FF67B48289B  mov         qword ptr [test],0
        test = (Test*)malloc(sizeof(Test));
    00007FF67B4828A3  mov         ecx,4
    00007FF67B4828A8  call        qword ptr [__imp_malloc (07FF67B49B428h)]
    00007FF67B4828AE  mov         qword ptr [test],rax
        test->data = 0;
    00007FF67B4828B2  mov         rax,qword ptr [test]
    00007FF67B4828B6  mov         dword ptr [rax],0
        free(test);
    00007FF67B4828BC  mov         rcx,qword ptr [test]
    00007FF67B4828C0  call        qword ptr [__imp_free (07FF67B49B430h)]
    }
```

上記は、先ほどの処理のアセンブリコードです。

下記は赤丸を翻訳してみた処理内容です。（ちょっと変ですが）

1	rbpとrdiをスタックに積んでいる。
2	ローカル変数で使われるメモリ領域を確保。
3	rsp+20のアドレス番号をrbpに格納します。
4	おまじない。
5	おまじない。

プログラムはどうやって動く？ 15（ザックリ過ぎる解説）

```
int main()
{
00007FF67B482880  push        rbp
00007FF67B482882  push        rdi
00007FF67B482883  sub         rsp,108h
00007FF67B48288A  lea         rbp,[rsp+20h]
00007FF67B48288F  lea         rcx,[__3D044E8B_Source@cpp (07FF67B49D06Dh)]
00007FF67B482896  call          CheckForDebuggerJustMyCode (07FF67B4817BCh)
    Test* test = NULL;
1 00007FF67B48289B  mov         qword ptr [test],0
    test = (Test*)malloc(sizeof(Test));
00007FF67B4828A3  mov         ecx,4
2 00007FF67B4828A8  call        qword ptr [__imp_malloc (07FF67B49B428h)]
3 00007FF67B4828AE  mov         qword ptr [test],rax
    test->data = 0;
4 00007FF67B4828B2  mov         rax,qword ptr [test]
5 00007FF67B4828B6  mov         dword ptr [rax],0
    free(test);
6 00007FF67B4828BC  mov         rcx,qword ptr [test]
00007FF67B4828C0  call        qword ptr [__imp_free (07FF67B49B430h)]
}
```

上記は、先ほどの処理のアセンブリコードです。

下記は赤丸を翻訳してみた処理内容です。（ちょっと変ですが）

1	変数testに0を代入してください。
2	引き数値は4にして、malloc関数を呼び出す。
3	malloc関数の戻り値を変数testに代入します。
4	変数testの実体へアクセス出来るようにする。
5	変数testの実体が持つdataに0を代入します。
6	引き数値はtestにして、free関数を呼び出す。

プログラムはどうやって動いてた？

ここまで、めちゃくちゃザックリとした解説でしたが、

プログラムがどのように処理され、メモリを確保しという流れはイメージ出来たのではないでしょうか。

最後の部分は、難しいと思ってしまうかもしれませんが、

パソコンがどうプログラムを動かしているのか一番直感的に理解出来るので大事だと思っています。

本当は、他にも面倒くさい話が沢山あるのですが、それは追々皆さんが学んでください。（丸投げ）

この内容さえ頭にあれば、そこまで難しくはありません。（多分、、、）

Cから学ぶ大切なこと（より発展的）

1. C言語はデータ指向なプログラム

データ指向ってなに？

データ指向とは？

名前の通り、データを中心にプログラムを組んでいく考え方です。

データ、処理を分けて、データがどのようにメモリに配置され使われるか、

処理はどのようにデータを読み込み、書き込むのかを意識した考えです。

オブジェクト指向とどう違うの？ 1

#仮コード例

```
97  class Enemy
98  {
99  public:
100
101     void UpdateAI(float deltaTime)
102     {
103         // 何かの処理
104     }
105
106     void UpdatePosition(float deltaTime)
107     {
108         position += velocity * deltaTime;
109     }
110
111 private:
112
113     // Translate
114     Vector3 position;
115
116     // Move
117     Vector3 velocity;
118
119     // AI
120     EnemyState state;
121     float elapsedTime;
122 };
```

オブジェクト指向（OOA）は、

データと、データに対する処理（プロセス）を、

一つの塊として定義します。（つまりカプセル化）

データは常にオブジェクトの中に隠されています。

雑に設計すると、凄く複雑なコードになってしまいます。

オブジェクト指向とどう違うの？ 2

#仮コード例

```
3   typedef struct Position
4   {
5       float x;
6       float y;
7   };
8
9   typedef struct Velocity
10  {
11      float x;
12      float y;
13  };
23  typedef struct EnemyAI
24  {
25      EnemyState state;
26      float elapsedTime;
27  };
```

```
54  void UpdatePosition(Position* positions, Velocity* velocities, int count, float deltaTime)
55  {
56      for (int i = 0; i < count; ++i)
57      {
58          positions[i].x = velocities[i].x * deltaTime;
59          positions[i].y = velocities[i].y * deltaTime;
60      }
61  }
62
63  void UpdateEnemyAI(EnemyAI* enemyAIs, int count, float deltaTime)
64  {
65      // 何らかのの処理
66  }
```

逆に、データ指向（DOA）は、

データと、データに対する処理（プロセス）を、

分離させて定義します。（つまりカプセル化しない）

データは常にオープンな状態になりますが、

システム（設計）が複雑にならないメリットがあります。

オブジェクト指向とどう違うの？３（メモリ配置の違い）

#仮コード例

```
123
124 int main()
125 {
126     Enemy enemies[ENEMY_CNT];
127
128     for (int i = 0; i < ENEMY_CNT; i++)
129     {
130         enemies[i].UpdateAI(DELTA_TIME);
131         enemies[i].UpdatePosition(DELTA_TIME);
132     }
133 }
```

※構造体やクラスは先ほどまでの仮コードと同様です。

#仮コード例

```
45
46     Position positions[ENEMY_CNT];
47     Velocity velocity[ENEMY_CNT];
48     EnemyAI enemyAI[ENEMY_CNT];
49
50     // メインループ
51     for (int i = 0; i < 100; ++i)
52     {
53         UpdatePosition(positions, velocity, OBJECT_CNT, DELTA_TIME);
54         UpdateEnemyAI(enemyAI, ENEMY_CNT, DELTA_TIME);
55     }
56 }
```

※構造体やクラスは先ほどまでの仮コードと同様です。

オブジェクト指向の、メモリ配置例

position	velocity	state	...
position	velocity	state	...
position	velocity	state	...
position	velocity	state	...

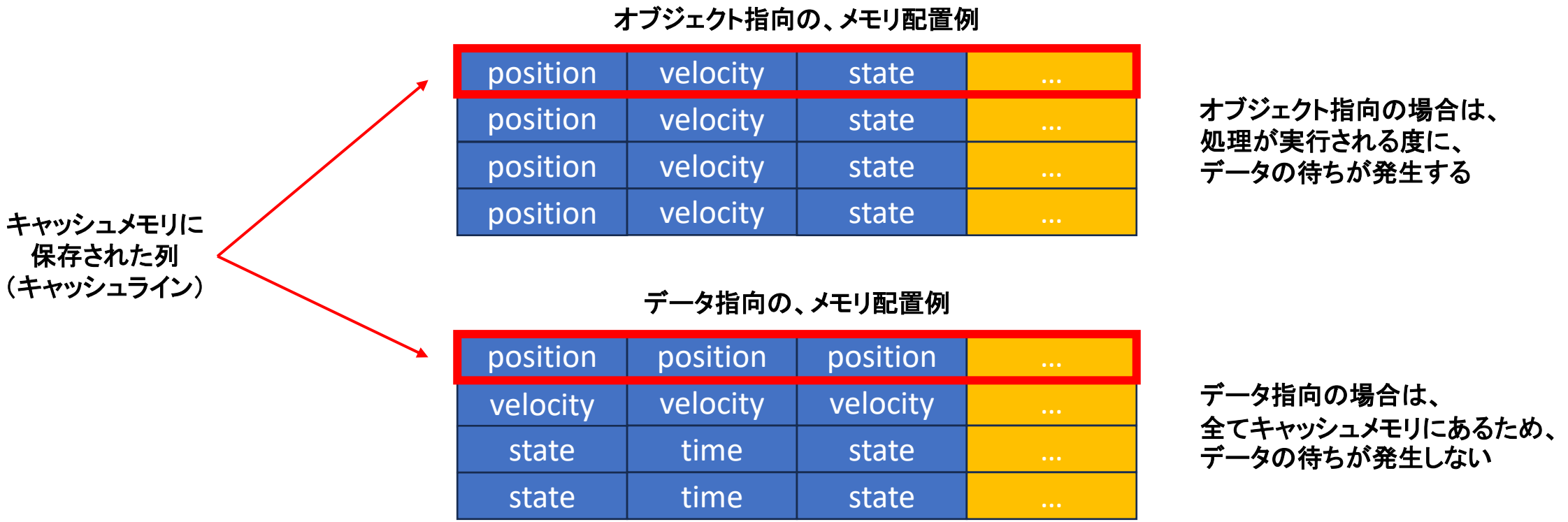
データ指向の、メモリ配置例

position	position	position	...
velocity	velocity	velocity	...
state	time	state	...
state	time	state	...

違いは分かったけど、結局、データ指向は何が良いの？

特に良くなるのはキャッシュミスが減ることでしょう。（= キャッシュメモリにデータがある確率が上がる）

下記の例では、全てのPositionを更新する際などは、データ指向の方が処理が速くなります。



データ指向についてまとめ

ここまでザックリと解説してきて、

オブジェクト指向とデータ指向の違いについては理解出来たのではないでしょうか。

元々クラスという概念がないので、C言語はデータ指向な設計だと言えると思います。

（無理くりすれば、意図的にオブジェクト指向にも出来ますが）

「これを理解して何するんだ？」と思うかもしれませんが、ECSなどの履修難易度は格段に下がります。

また、メモリマップを理解することは、ゲーム制作に大きく影響します。（この最高峰がゼルダです。）