3 章 C の文法を解き明かす

3-1 C の宣言を解読する

暗号の解読、むずかしい。

3-2 C の型モデル

3-2-1 基本型と派生型

「◯ 型のポインタへの配列への関数へのポインタへの...」とすることで無限に型を生み出せる。 「int の配列」も「char へのポインタへの配列」も結局は配列という意味で最後の型が重要である。

3-2-2 ポインタ型派生

「int のポインタ」も「char へのポインタへのポインタ」などを総称して「T へのポインタ」という。 ポインタに加算をすると T のサイズだけポインタが進む。

```
int array[3];
int (*array_p)[3];

array_p = &array; // 配列へのポインタ
```

この場合、ポインタに加算をすると `4*3` の 12 バイト進む。

3-2-5 C 言語には多次元配列は存在しない

多次元配列のように見えるのは「配列の配列」

3-2-6 関数型派生

関数型からはポイント型にしか派生できない。

3-2-7 型のサイズを計算する

後述の不完全型以外は型にサイズが決まっている。 どんな型でも `sizeof(型)` とすればサイズがわかる。

3-2-8 基本型

char, int, long, float などのこと

3-2-9 構造体と共用体

構造体と共用体は文法上は派生型

3-2-10 不完全型

型のサイズが決まらない型の中で、関数型以外のもの。

最終的に C 言語の型は以下の 3 つに分類される。

- オブジェクト型(char、int、配列、ポインタ、構造体など)
- 関数型
- 不完全型

相互に要素を参照する場合の例

```
typedef struct Woman_tag Woman; // 先にtypedef

typedef struct {
    Woman *wife; // 妻
} Man;

struct Woman_tag {
    Man *husband; // 妻
};
```

最初 Woman_tag が宣言された時点ではサイズがわからないので不完全型。 その後、Woman_tag の内容を定義した時点で不完全型ではなくなる。

3-3 式

3-3-1 式とデータ型

式は木構造になるよ。

> a + b * c は a も b + c も a + b * c も式になる。

そして、**あらゆる式は型を持つ**。

```
int index = 2;
char str[256];

printf(str);
"0123456"[index];
```

str も文字列リテラルも式の中では「char へのポインタ」になるのでこんな書き方もできる。

3-3-2 左辺値とは何か

- 式がどこかの記憶領域を意味しているものを左辺値(lvalue)
- 式が単なる値を意味しているものは右辺値(rvalue)

```
x = 10;
++y;
```

x と y は左辺値、 10 は右辺値

3-3-3 配列 → ポインタの読み替え

式の中で配列はポインタに読み替えられる。 ただし、以下の場合は例外である。

1. sizeof 演算子のオペランドの場合
    - sizeof ではポインタのサイズではなく、配列のサイズが返されるように読み替えが抑止される
2. &演算子のオペランドの場合
    - 配列に対して `&` をつけるとその配列全体へのポインタを返す
3. 配列初期化時の文字列リテラル
    - 3-5-4 で解説

3-3-4 配列とポインタに関係する演算子

間接演算子 `*`

- 単項演算子 `*` を間接演算子という
- `*` はポインタをオペランドとしてとり、指し示すオブジェクトまたは関数を返す
- オブジェクトを返す場合、 `*` の結果は左辺値を持つ
- T 型へのポインタ `t_p` に対して、 `*t_p` の型は T となる。

アドレス演算子 `&`

- 単項演算子 `&` をアドレス演算子という
- `&` は左辺値をオペランドとして取る
- T 型のオペランド `t` に対して、 `&t` の型は **T 型へのポインタ** となる

添字演算子 `[]`

- 後置演算子 `[]` を添字演算子という
- `p[i]` は `*(p+i)` のシュガーシンタックスでありそれ以外の意味を持たない

アロー演算子 `->`

- `->` 演算子はぽいん t 名を経由して構造体のメンバを参照するのに使う
- `p->hoge` は `(*p).hoge` のシュガーシンタックスである

3-3-5 多次元配列

多次元配列「もどき」である `array[i][j]` にアクセスした時に何が起きているか解説している。

3-4 続・C の宣言を解読する

3-4-1 const 修飾子

const は ANSI C で追加された型修飾子であり、「読み出し専用」であることを意味する。

```
// src は、charへの、読み出し専用のポインタである
char  * const src;

// src は、「読み出し専用のchar」へのポインタである
char const *src;
const char *src;
```

3-4-2 const をどう使うか？どこまで使えるか？

「const がついたものが指すもの」は読み出し専用だが、「const がついたものが指すもののさらに先にあるもの」に関しては読み出し専用ではない。

3-4-3 typedef

typedef はある型に対して別名をつける機能である。たとえば、

```
typedef char *string;
```

と宣言すると、以後「char へのポインタ」という型に対して「String」という別名をつけることができる。

```
typedef struct {
    ...
} Hoge;
```

と宣言することで構造体に別名をつけられる。

3-5 その他

3-5-1 関数の仮引数の宣言(ANSI C 版)

関数の仮引数に配列を渡すことはできない。

```
void func(int a[])
```

としていても、**関数の仮引数の宣言では**型分類としての配列はポインタに読み替えられるので、

```
void func(int *a)
```

に自動的に読み替えられる。 仮引数に配列の要素数を書いても無視される。

> **重要**
>
> `int a[]` が `int *a` と同じ意味になるのは、唯一、関数の借り引数の宣言の場合だけである。

3-5-2 関数の仮引数の宣言(C99 版)

VLA の一環として「縦横可変の多次元配列」を関数に渡すことができるようになった。

```
void func(int size1, int size 2, int a[size1][size2])
void func(int size1, int size 2, int a[][size2]) // 配列のサイズは無視される
void func(int size1, int size 2, int (*a)[size2]) // 実態はポインタなのでこのように宣言しても良い
```

3-5-3 空の[]について

以下の例ではコンパイラが特別に解釈するため、空の `[]` を書くことができる。

1. 関数の仮引数の宣言

関数の仮引数の宣言において**最外周の配列に限り**、ポインタに読み替えられる。

2. 初期化子により配列のサイズが確定できる場合

以下のように、初期化子により必要な要素数をコンパイラが決定できる場合は、**最外周の配列に限り**、要素数を省略できる。

```
int a[] = {1,3,5,7};
char str[] = "abc";
double matrix[][2] = {{0,1},{1,0}};
char *color_name[] = {
    "red",
    "green",
    "blue",
};
```

3. グローバル変数を extern 宣言する場合
4. 構造体のフレキシブル配列メンバ

3-5-4 文字列リテラル

"" で囲まれた文字列のことを「文字列リテラル」という。 文字列リテラルの型は「char の配列」であり、式の中では「char へのポインタ」に読み替えられる。

```
char *str;
str = "abc";
```

ただし、char の配列を初期化する場合は例外である。

```
char str[] = "abc";
```

これは下と同義である。

```
char str[] = {'a', 'b', 'c', '\0'};
```

3-5-5 関数のポインタにおける混乱

- 関数は式の中では「関数へのポインタ」に自動的に変換される。ただし、アドレス演算子 & のオペランドであるときと、sizeof 演算子のオペランドである時は例外である
- 関数呼び出し演算子 ( ) は関数ではなく関数へのポインタをオペランドとする

3-5-6 キャスト

ある型を強制的に他の型に変換する演算子

```
(型名)
```

例として整数同士の割り算を小数で表す時に使う。

```
int a,b;
printf("a/b ... %f\n", (double)a/b);
```

また、ポインタをキャストする時にも使う。

3-5-7 練習 - 複雑な宣言を読んでみよう

複雑な宣言を読む練習をしている。

3-6 配列とポインタは別ものだ

3-6-1 なぜ混乱してしまうのか

C において、配列とポインタは別物です。

3-6-2 式の中では

式の中では、配列はその先頭要素のポインタに読み替えられる。

```
int *p;
int array[10];
p = array;  // pには、array[0]へのポインタが代入される
```

しかし、逆に

```
array = p;
```

のような書き方はできない。 array は式の中でポインタに書き換えられるが、あくまで `&array[0]` に読み替えられているというだけで、この場合のポインタは**右辺値**であるからだ。

`a + 1 = 10` という代入ができないのは、`a + 1`がそれに対応する記憶領域を持たない右辺値であるから、という理由と同じ。

3-6-3 宣言では

配列の宣言がポインタの宣言に読み替えられるのは、唯一、関数の仮引数の宣言の場合のみである。