2 章_実験してみよう

2-1 仮想アドレス

現在の PC に搭載される OS ではマルチタスク環境を提供しており、複数のプログラム(プロセス)を同時に実行できる。

例えば、Mac ならターミナルと iTerm2 を同時に起動して同じプログラムを実行する。 このとき、同じアドレスが保持する値を表示するプログラムを動かすと、同じアドレスなのに違う値が返ってくる。

つまり、今時の環境ではポインタを `printf()` で表示して見える値は物理的なメモリのアドレスそのものではない。 今時の PC 環境ならアプリケーションプログラムには、**プロセスごとに独立した**「仮想アドレス空間」が与えられる。 これは C 言語とは関係なく OS と CPU が連携して行う仕事である。 だから、未熟なプログラマが意図しない領域に書き込むようなバグを仕込んでも他のプロセスに迷惑がかかることはないのである。

> 今時の環境なら、アプリケーションプログラムに見えるのは、仮想アドレス空間だ。

2-2 C のメモリの使い方

2-2-1 C における変数の種類

C の変数は **スコープ**(scope : 有効範囲)と**記憶域期間**(storage duration)という 2 つの軸で分類することができる。

スコープ

1. グローバル変数

関数の外側で定義した変数はデフォルトでグローバル変数になる。 プログラムを複数のソースファイルに分けて分割コンパイルする場合、グローバル変数は宣言さえしていれば別のソースファイルからでも参照できる。

2. ファイル内 static 変数

グローバル変数のように関数の外側で定義した変数でも、 static をつけるとスコープはそのソースファイル内に限定される。

3. ローカル変数

関数の中で宣言した変数のこと。ローカル変数は宣言を含むブロック(`{}` で囲まれた内側の範囲)の中でだけ参照することができる。

ローカル変数は基本的に関数の先頭で定義するが、ブロックの頭でも宣言できる。 また、C99 からはブロックの途中でもローカル変数を宣言できるようになった。

記憶域期間

1. 静的記憶域期間(static storage duration)

グローバル変数、ファイル内 static 変数、static 指定をつけたローカル変数は**静的記憶域期間**をもつ。 これらの変数を総称して**静的変数**と呼ぶことがある。 これらの変数は、プログラムの開始から終了までの寿命を持ち、メモリの同一アドレス上に存在し続ける。

2. 自動記憶域期間(auto storage duration)

static 指定のないローカル変数は**自動記憶域期間**をもつ。 このような変数を**自動変数**と呼ぶ。 自動記憶域期間を持つ変数はそのブロックに入ると同時に領域が確保され、ブロックを抜けると解放される。 これには通常**スタック**というしくみが使われる。(スタックについては後述)

また、変数ではないが `malloc()` を使用して動的なメモリ確保を行うことができる。 `malloc()` で確保した領域は `free()` で解放するまでの寿命を持つ。

2-2-2 アドレスを表示させてみよう

```c
#include <stdio.h>
#include <stdlib.h>

int global_variable;
static int file_static_variable;

void func1(void)
{
    int func1_variable;
    static int local_static_variable;

    printf("&func1_variable .. %p\n", (void *)&func1_variable);
    printf("&local_static_variable .. %p\n", (void *)&local_static_variable);
}

void func2(void)
{
    int func2_variable;
    printf("&func1_variable .. %p\n", (void *)&func2_variable);
}

int main(void)
{
    int *p;

    // 関数へのポインタの表示
    printf("func1 .. %p\n", (void *)func1);
    printf("func2 .. %p\n", (void *)func2);

    // 文字列リテラルのアドレス表示
    printf("string literal .. %p\n", (void *)"abc");

    // グローバル変数のアドレス表示
    printf("&global_variable .. %p\n", (void *)&global_variable);

    // ファイル内static変数のアドレス表示
    printf("&file_static_variable .. %p\n", (void *)&file_static_variable);

    // ローカル変数の表示
    func1();
    func2();

    // malloc()により確保した領域のアドレス
    p = malloc(sizeof(int));
    printf("malloc address .. %p\n", (void *)p);

    return 0;
}
```

上記のコードを実行した結果

func1	**0x105914da0**
func2	0x105914de0
string literal	0x105914f69
&global_variable	0x105916020
&file_static_variable	0x10591601c
&func1_variable	0x7ffeea2eb25c
&local_static_variable	0x105916018
&func2_variable	0x7ffeea2eb25c
malloc address	0x7f996fc05970

自動変数だけが大きく離れた位置に割り当てられていることがわかる。

2-3 関数と文字列リテラル

2-3-1 書き込み禁止領域

関数（プログラム）本体と文字列リテラルは隣接したアドレスに配置される。 これは、関数本体と文字列リテラルをまとめて 1 つの**書き込み禁止領域**に配置するため。

2-3-2 関数へのポインタ

関数へのポインタを格納するポインタの宣言は変態的

```
int func (double d);  // プロトタイプの関数

int (*func_p)(double); // 関数へのポインタ
```

サンプルプログラム

```
#include <stdio.h>

/* 引数に1.0を足して表示する関数 */
void func1(double d)
{
    printf("func1: d + 1.0 = %f\n", d + 1.0);
}

/* 引数に2.0を足して表示する関数 */
void func2(double d)
{
    printf("func2: d + 2.0 = %f\n", d + 2.0);
}

int main(void)
{
    void (*func_p)(double);

    func_p = func1;
    func_p(1.0);

    func_p = func2;
    func_p(1.0);

    return 0;
}
```

```
# 実行結果
func1: d + 1.0 = 2.000000
func2: d + 2.0 = 3.000000
```

「関数へのポインタ」を変数に格納するテクニックは以下のようなケースで使用する。

1. GUI のボタンを表示するときに「そのボタンが表示されたら呼び出される関数」をボタンに覚えておいてもらう。
2. 複雑な処理をライブラリするが、処理の一部をカスタマイズしたい場合、たとえばソートのプログラムについて比較処理だけを外部から与えるようにする。
3. 「関数へのポインタの配列」により処理を振り分ける。

2-4 静的変数

2-4-1 静的変数とは

静的変数はプログラムの開始から終了まで存在し続ける。よって（仮想）アドレス上で固有の領域を占有する。

2-4-2 分割コンパイルとリンク

C では、複数のソースファイルによりプログラムを構成し、それぞれ別々にコンパイルしてから結合することができる。

static 指定のない関数とグローバル変数については **名前が同じであれば** ソースファイルを跨いでも同じものとして扱われる。 この作業を行うのが **リンカ** と呼ばれるプログラムである。

リンカに名前を結合してもらうために、各オブジェクトファイルは **シンボルテーブル** という表を備えていることが多い。

2-5 自動変数(スタック)

2-5-1 領域の「使い回し」

> 自動変数の領域は、関数を抜けたら別の関数呼び出しで使いまわされる。 自動変数のアドレスは関数の呼び出しにより変動し、一定とは限らない。

2-5-2 関数呼び出しで何が起きるか？

> C では通常、自動変数はスタックに確保される。

main の内部で呼び出した関数、それがさらに呼びだした関数、はどんどん小さい方にアドレスを積み上げていく。 これが 2-2-2 で自動変数だけが大きく離れた位置に割り当てられている所以である。

2-5-3 自動変数をどのように参照するのか

アセンブリ言語を用いてローカル変数がベースポインタからのオフセットで参照されていることを解説している。

2-5-4 典型的なセキュリティホール -- バッファオーバーフロー脆弱性

配列の範囲を大きく超えて破壊すると、その関数の復帰情報まで破壊してしまい、その関数から戻ることができなくなる。 復帰情報を書き換えられるということは、悪意のある攻撃者が設定した任意のリターンアドレスから再開されるので、任意のプログラムを実行できてしまう。 これを **バッファオーバーフロー脆弱性** という。

2-5-5 可変長引数

C では可変長の引数を取る関数を作ることができる。例として `printf()` があげられる。

```
printf("%d, %s\n", 100, "str");
```

が参照する領域は小さい方から以下のように並んでいる。

- printf のローカル変数
- 復帰情報、リターンアドレスなど
- 第一引数（`%d, %s\n`）
- 第二引数（`100`）
- 第三引数（`str`）

printf のローカル変数から見て第一引数が常に同じ位置にあるようにするために、引数は後ろから順にスタックに積み上げられていく。

可変長引数の関数ではプロトタイプ宣言による引数の型のチェックが効かないなど、デバッグが難しくなるので可変長引数は、そうしないとソースが描きにくくてどうしようもない場合にのみ使うようにしよう。

2-5-6 再起呼び出し

C では通常、自動変数の領域をスタックに確保する。これは領域を使いまわしてメモリを確保する他に、**再起呼び出し**を可能にするという重要な意味がある。 再帰呼び出しの例でフィボナッチ数列がよくあるが実用的じゃないので、ここでは順列の数え上げ（3 連単の買い方）を再起呼び出しを用いて実装する。

```
#include <stdio.h>
```

```c
// nの最大数
#define N_MAX (100)

// 数字を使用したら、その添字の要素を1にする
int used_flag[N_MAX + 1];

int result[N_MAX];
int n;
int r;

void print_result(void)
{
    int i;
    for (i = 0; i < r; i++)
    {
        printf("%d ", result[i]);
    }
    printf("\n");
}

void permutation(int nth)
{
    int i;
    if (nth == r)
    {
        print_result();
        return;
    }

    for (i = 1; i <= n; i++)
    {
        if (used_flag[i] == 0)
        {
            result[nth] = i;
            used_flag[i] = 1;
            permutation(nth + 1);
            used_flag[i] = 0;
        }
    }
}

int main(int argc, char **argv)
{
    sscanf(argv[1], "%d", &n);
    sscanf(argv[2], "%d", &r);
    permutation(0);

    return 0;
}
```

2-5-7 C99 の可変長配列(VLA)におけるスタック

自動変数に限り配列を可変長にできる。なぜなら自動変数はスタックに確保されるから。

2-6 malloc()による動的な領域確保（ヒープ）

2-6-1 malloc の基礎

C では `malloc()` を使って動的に領域を確保できる。 `malloc()` は引数で指定したサイズのメモリの塊を確保して、その先頭へのポインタを返す関数である。

```c
p = malloc(size);
```

メモリ確保に失敗した場合（メモリが足りない場合）、`malloc()`は`NULL`を返す。 `malloc()`で確保した領域は使い終わったら `free()` により解放する。

```c
free(p);
```

このように、動的にメモリを割り当てて任意の順序で解放できる記憶領域のことを**ヒープ**と呼ぶ。

malloc()の主な使用例

1. 構造体を動的に確保する

2. 実行時までサイズがわからない配列を確保する

2-6-2 malloc()は「システムコール」か？

ちょっと脱線した話題。

> **システムコール**
>
> OS に何かしてもらおうとするよう要求する特別な関数群。

例えば `printf()` は最終的には `write()` というシステムコールを呼び出す。

`malloc()` は標準他ライブラリの関数ではあるが、システムコールではない。

2-6-3 malloc()で何が起きるのか？

malloc()の動作原理について知識を持ち合わせていないとデバッグができなかったり、効率の悪いプログラムを書いてしまう。

> malloc()は魔法の関数ではない。

2-6-4 free()したあと、その領域はどうなるのか？

free()したからといって、**その領域が即座に OS に返されるわけではない。**

ポインタ A とポインタ B が同じ領域を参照していて、 ポインタ A 側から`free()`したとしても、しばらくはポインタ B 側から以前と同じ値を見ることができる。 どこか別のところで `malloc()` が実行されてこの領域が割り当てられて初めて内容が壊れる。

2-6-5 フラグメンテーション

ランダムな順序でいろいろなサイズの領域の確保、開放を繰り返すと、メモリが分断されて細かい空きブロックができることになる。 このような領域は **事実上使用できない。** このような現象を **フラグメンテーション** という。 C ではアプリケーションプログラムに仮想アドレスを直接渡しているため、ライブラリ側から勝手にブロックを移動させて空き領域を詰めることはできない。

2-6-6 malloc()以外の動的メモリ確保関数

calloc()

```
#include <stdlib.h>
void *calloc(size_t nmemb, size_t size);
```

`calloc()` は `malloc()` と同じ方法で、 `nmbmb * size` だけの領域を確保し、その領域をゼロクリアして返す。

realloc()

すでに `malloc()` で割り当てている領域のサイズを変更するための関数。

```
#include <stdlib.h>
void *realloc(void *ptr, size_t size);
```

`realloc()` は `ptr` の指す領域サイズを `size` に変更し、新しい領域へのポインタを返す。

> realloc() は使い方に気をつけよう。

2-7 アラインメント

手作業でパディングを入れても移植性を高めることにはならない。

2-8 バイトオーダー

> 整数にしろ浮動小数点にしろ、メモリ上での表示形式は環境によってバラバラ

2-9 言語仕様と実装について

-- ごめんなさい、ここまでの内容はかなりウソです

C は確かに低級言語かもしれないが、高級言語らしく使おうと思えば結構使える。