Learning C part 2

このテキストは紅白の巫女さんと黒白のゴスロリさんによって書かれました[i]

ゼミＡで華麗にスルーした細かい C 言語の仕様等について説明します。知らなくても困ることはありませんが、知っておくと便利なことばかりです。

第一章　プリプロセッサ命令

C 言語の中でもちょこっと異質な存在の#[ii]の記号について説明します。

#から始まる命令のことをプリプロセッサ命令と言います。このプリプロセッサ命令というのは C 言語の仕様ですが、C 言語自体のプログラムの仕組みではありません。どちらかというとコンパイラ側の仕組みにあたります。つまりコンパイル時に処理し、プログラム実行時は何も干渉がない存在にあたります。だからといって何もしない存在ではなく、正しくコンパイルをするための道しるべとしてコンパイル時に役立っています。コンパイル時にプログラムは合っているのにエラーが出るといった場合はこのプリプロセッサ命令が正しく行われていないことも多々あります。

第一節　#include 命令

C 言語を学ぶ上で一番最初に学ぶプログラムであろう「hello wolrd!!」[iii]という表示プログラム。その最初に一行目には必ず#include<stdio.h>というものが書かれます。これは stdio.h というファイルを読み込む(インクルードする)ということを指します。stdio.h というのは Standard Input Output.Header を指し、標準入出力関数群のファイルのことを指します。stdio.h の中には printf 関数や scanf 関数などが含まれています。この#include<stdio.h>を書くことによってこれらの関数を使うことができます。逆にこの#include<stdio.h>とプログラムの冒頭に書かなかった場合は printf 関数などが使えなくなります(コンパイルエラーになる)。

stdio.h 以外にも有名なものとして、math.h や stdlib.h、string.h といったものがあります。これらは C 言語をコンパイルできる環境なら必ず入っているべきヘッダ群になります。またそれら以外にも任意でヘッダをインクルードすることができます。例えば WindowsAPI を使いのならば#include<windows.h>や DX ライブラリを使うならば#include"DxLib.h"というものがあります。インクルードする方法として不等号記号「<bracket.h>」という方法か「"quotation.h"」という二種類の方法があります。この違いは読み込むヘッダファイルが標準ライブラリのディレクトリを探すか、カレントディレクトリを探すかの違いです。

ここで#include 命令の厄介な点を説明します。それは一度読み込んだヘッダに対しても複数回読み込もうとする点です。複数回読み込んでしまうことにより既に宣言した大域変数や関数が再び宣言されてしまうためコンパイルエラーを起こしてしまいます[iv]。この問題は非常に厄介で、ヘッダ側で対処するための常套句はあるものの自ら作ったヘッダファイルに対しては全てにその常套句を適用する必要があります(具体的な方法については後述)。

---

i　脚注の権限は私がもらった

ii　#は井桁(いげた)と読む場合もある

iii　おそらく世界一有名なプログラム

iv　Objective-C では#import を使うことによりこの問題を解決している

第二節　#define 命令

#命令には他に#define 命令というものがあります。これは定数を定義することができます。下記のように使います

```
#include<stdio.h>
#define WITCH "Beatrice"
int main(void){
    printf("魔女 : %s", WITCH);
    return 0;
}
```

上記の例では「WITCH」という言葉を「Beatrice」という文字列に置き換えます。定数というのはこのように文字列に対しても有効です。

```
#define（マクロ名）（置き換える数字や文字列）
```

慣例として#define で定義されるものは全て大文字で記されます。理由として、他の変数と区別するためです。この#define 命令は自らで定義するものの他に、コンパイラが既に定義しているものがいくつかあります。TRUE や FALSE、NULL、EOF などがその例です。これらは既に当たり前のように使用していますが、全て stddef.h などで一つひとつ定義されているものです。

また、定数以外にも関数のような使い方ができます。

```
#include<stdio.h>
#define M_ABS(x) 0 < x ? x : -x
int main(void){
    int num;
    scanf("%d", &num);
    printf("num = %d, abs_num = %d", num, M_ABS(num));
    return 0;
}
```

マクロ関数[i]と呼ばれます。上記の例[ii]では貰った値の絶対値を返すマクロ関数です。通常の関数ではないので行末にセミコロンは要りません(#命令には全て共通することですが)。このマクロ関数の利点として、引数の型を問わないという点です。上記のマクロ関数は int 型でも double 型でも問題なく処理することができます。

またマクロ関数は引数を二つ以上取ることも出来ます。

```
#include<stdio.h>
#define M_ADD(x, y) x + y
int main(void){
    int m, n;
    scanf("%d %d", &m, &n);
    printf("m + n = %d", M_ADD( m, n));
```

---

i　C++では似たようなもとのとしてインライン関数と呼ばれるものが用意されている

ii　2行目の「0 < x ? x : -x」とハテナマークとコロンを二つ使うものを三項演算子という

```
    return 0;
}
```

このような感じになります。#define 命令はコンパイル時に検索置換をしているだけなので、通常の関数より処理速度は速くなります。しかし、マクロ関数はあまり多様しすぎると実行ファイルが大きくなるので注意が必要です。

第三節　その他のプリプロセッサ命令

主に使うのは上記の二つですが、それ以外にも#から始まる命令はいくつかあります。それらの名前と使用方法を列挙します。

```
#if, #ifdef, #ifndef, #else, #elif, #endif
```

名前を見ただけでどういったことをするかはわかると思いますが、通常の if 文に相当する命令です。これらはデバッグ用とリリース用にコンパイルする時に使います。

```
#ifdef DEBUG
    printf("デバッグ¥n");
#else
    printf("リリース¥n");
#endif
```

これは初めに#define DEBUG という文があれば、「デバッグ」と表示され、定義されていなければ「リリース」と表示されます。デバッグ用とリリース用で大きくプログラムが異なる場合に使います。通常の if 文の場合だと分岐毎に逐一判定されるのですが、この場合はコンパイル時に判定が行われるので実行速度は落ちません[i]。

また、先ほど説明したヘッダファイルに対しては下記のようなプログラムがよく書かれます。

```
//witch.h
#ifndef ___witch
#define ___witch

    //省略　関数のプロトタイプ宣言など

#endif
```

#ifndef のところは____witch が定義されていなければ、という判定を行っています。その直後に____witch を定義しています。これによって、もう一度この witch.h が#include された場合でも実際の witch.h の中身は多重に読み込まれずにすみます。ちなみに各名前にアンダーバーが三つ重ねて置いてあるのはコンパイラが定義しているものと重複を防ぐため[ii]です。

#error (error_message)

コンパイル時にエラーを出すことができます。あまり使用する機会はないですが、強制的にコンパイルを止めたいときに使えます。

---

i　但しコンパイル速度は落ちます(実質影響はないですが)

ii　標準ライブラリ側もかなりの数の#ifndef ～#define を使っているので重複するとコンパイルエラーになる

第二章　繰り返し命令

プログラムでは欠かせない繰り返しの命令について説明します。
通常の繰り返し命令の他にジャンプ命令についても説明します。

第一節　ブロック

繰り返し命令を説明する前に繰り返しを囲む括弧について説明します。

```
int main(void){
    int sum, i;
    for( i = 0; i < 8; i++){
        sum += i;
    }
    return 0;
}
```

この for 文で回す範囲、「{ ... }」で囲まれたエリアを一つのブロックと呼びます。これは別に for 文や while 文を使わずとも任意にブロックを作ることができます。

```
int main(void){
    int x, y, z;
    {
        scanf( %d, &x);
        scanf( %d, &y);
        scanf( %d, &z);
    }
    printf( "x = %d, y = %d, z = %d", x, y, z);
    return 0;
}
```

上記のように、一つのプログラム的まとまりを一つのブロックとしておくとプログラムの可視性が上がり保守がしやすくなります。また、話は少しそれますが、C 言語においては変数の宣言位置はブロックの最初と決められています[i]。今までは関数の最初の部分と説明してきましたが、関数の最初以外にも for 文の最初や while 文の最初、先ほどのプログラムのように任意のブロックの最初の位置にも変数を宣言できます。そこで宣言した変数はブロックの終わりまで有効になります。一時的に使う変数などはその場で宣言するというのも一つの手です。

第二節　for 文

プログラムの醍醐味である for 文について説明します。

for 文は配列で宣言された各要素にそれぞれアクセスする、一定の回数同じ処理をする、ある一定の条件が満たされるまで処理を行うなど沢山の使い道があります。まずは文法事項を確認します。

---

i　1999 年に改訂された C99 では宣言場所はどこでも可能となった

```
for(（初期値の設定）;（繰り返しの条件）;（ループ毎の処理）){ ... }
```

まず for 文は「初期値の設定」を行います。通常は「i = 0」と書くことが多い場所です。次に繰り返しの条件判定が行われます。ここで偽と判定されれば for 文の内側を一度もループしないこともあります。真と判定されれば、for 文で繰り返されるブロックの内側を実行します。ブロックの実行が終わったら「ループ毎の処理」を行います。そして「繰り返しの条件」を行い、元に戻ります。

これを while 文で書き換えると次のようになります。

```
（初期値の設定）
while((繰り返しの条件)){
    { ... }
    （ループ毎の処理）
}
```

for 文と while 文は相互に書き換えが可能ですが、for 文のほうが綺麗に書けることが多いです。

また変わり種として下記のようなこともできます。

```
int main(void){
    int i, j;
    for( i = 0, j = 0; i < 0 && 0 < j; ){
        scanf("%d %d", &i, &j);
    }
    return 0;
}
```

二変数以上の初期値の設定や、条件設定です。for 文の各要素には一つの式が入ります。ですがカンマ演算子「 , 」[i]を使うことによって複数の式を無理矢理入れることができます。これによって通常は一つの変数しか初期化できないのを複数の変数を同時に初期化することができます。また for 文第二項目は if 文と全く同じ働きとなるのでアンド演算子やオア演算子などを入れて複雑な条件にすることができます。

第三節　while 文

while 文も for 文とともに頻繁に出てくるループのための機能です。

また派生系として do-while 文というのもあります。while 文はループの中を一回も通らないという選択肢がありますが、do-while 文は必ずループを 1 ループしてから条件判定を行います。また判定が後に行われるため、最後にセミコロンが必要になるところが注意が必要です。

```
#include<stdio.h>
int main(void){
    int num;
    do{
        scanf("%d", &num);
    }while(num < 0);
```

i　カンマ演算子とは、カンマを挟んで左側の式を評価し、その後右側の式を評価する演算子

```
}
```

do-while 文は上記の用に必ず一回はキーボード入力を行わせる、と言った場合に有用です。キーボードからの入力が負の数だともう一度入力をさせるといったことができます。ちなみに上記の do-while 文のところを while 文に書き換えると

```
while(scanf("%d", &num), num < 0);
```

と、一行で済まそうと思えば済ますこともできます[i]。

第四節　ジャンプ命令

ジャンプ命令というのはプログラムは順々に流れていきますが、いきなり実行場所を変える命令を指します。ジャンプ命令は下記の四つがあります。

```
break
return
continue
goto
```

break はループから抜け出す役目があります。for 文、while 文、do-while 文、switch 文に対して有効です。

return は関数から抜け出す役目があります。一つの関数に対し複数個あることは構いませんが、return を通らないパターンは作ってはいけません[ii]。void 型の関数の場合は値を返す必要がないので return は必要ないと思われがちですが、値を返さないだけで return を使うことができます[iii]。

continue は break の逆の意味で次のループに行くという意味です。

```
#include<stdio.h>
int main(void){
    int i;
    for( i = 0; i < 10; i++){
        if( i % 2 == 0) continue;
        printf("i = %d¥n", i);
    }
    return 0;
}
```

上記のプログラムを実行すると、偶数の数字は continue 文を通るので for 文の最初まで戻されてしまいます。よって、奇数の数字だけ表示されます。

goto[iv]はまともな本では「多用するな」と書かれています[v]。

```
#include<stdio.h>
int main(void){
    int i, j, num;
```

---

i　但し人に見せるプログラムではない

ii　C 言語では Waring、C++では Error になります

iii　具体的には「return;」で終わらせる。return の後にすぐにセミコロンがくるのがポイント

iv　BASIC 経験者ならお馴染みであろう。しかし近年の言語では用意されていないことも多い

v　もしくは紹介すらされない

```
        for( i = 0; i < 10; i++){
            for( j = 0; j < 10; j++){
                scanf("%d", &num);
                if(num == 0) goto loop_finish;
            }
        }
        loop_finish:
        return 0;
    }
```

上記の場合、通常は１０×１０の１００回ループになりますが、キーボードから０と入力された場合はループから抜け出すことができます。C 言語では多重ループを抜け出す手段が用意されていません[i]。なのでこのように goto 文を使うか、わざわざ変数を用意し if 文を使うなど若干の面倒さが伴います。goto は非常に強力なのでループ以外の場所でも使うことができますが、可視性が非常に落ち、いわゆるスパケティコード[ii]化するのでごく限られた場所でのみ使用すべきだと言われています。

i　Java では用意されている

ii　スパゲッティのように絡まったコードのこと

第三章　配列

何かと使う機会が多い配列について説明します。通常の宣言のやり方では配列の要素数を可変することはできませんでした。そこには配列とポインタの無駄に深い関係があるからです。云々。

第一節　普通の配列

まずは基本事項のおさらいをします。

```
#include<stdio.h>
int main(void){
    int array[4] = { 4, 3, 2, 1};
    int i;
    for(i = 0; i < 4; i++){
        printf("array[%d] = %d¥n", i, array[i]);
    }
    return 0;
}
```

上のプログラムを実行すると、各要素に入った数字が表示されます。配列は int　array[4]と宣言された場合は四つの int 型の変数が作られます。array[0]～ array[3]までです。0 が数字の最初となるので 4 - 1 の 3 が四つ目の配列となります。

つづいて二次元配列の説明に入ります。

```
#include<stdio.h>
int main(void){
    char array[3][16] = {"Beatrice", "Bernkastel", "Lambdadelta"};
    int i;
    for(i = 0; i < 3; i++){
        puts( array[i]);
    }
    return 0;
}
```

上記のプログラムは 3 行 16 列の配列となります。ここではせっかくの配列なので文字列[i]を入れてみました。char　array[3][16]と宣言した場合は 3 × 16 の 48 個の char 型の配列が生成されます。

第二節　動的配列の確保

下記のようなプログラムを書こうと思ったことはありませんか？

```
#include<stdio.h>
int main(void){
    int num;
```

i　文字列の扱いについては前回のテキスト参照

```
        scanf("%d", &num);
        {
            int array[num];
            //以下何かの処理
        }
        return 0;
    }
```

おそらくコンパイルエラーがでます[i]。また配列を int array[100000]くらい宣言したとしてもコンパイル時にエラーが起きます。C 言語では一定領域を確保する関数 malloc[ii] というものが用意されています。

```
    #include<stdio.h>
    #include<stdlib.h>
    int main(void){
        int num;
        scanf("%d", &num);
        {
            int *array;
            array = (int*) malloc( sizeof(int) * num);
            //何かの処理
            free(array);
        }
        return 0;
    }
```

malloc 関数は stdlib.h の中に入っている関数なので、上記のプログラム二行目で stdlib.h をインクルードしています。また malloc 関数を対をなすものとして free 関数というのも使います。まず、ポインタを宣言します(int *array)。このポインタは作る配列が doulbe 型だったら double 型のポインタ、char 型だったら char 型のポインタを宣言します。そして、malloc 関数は int 型の引数をとります。その値が領域として確保されます。例えば malloc(4)と書いた場合は 4byte の領域が確保されます。上記の場合は malloc(sizeof(int) * num)[iii]となっているので int 型の変数 1 つ分× num となります。具体的にすると num = 4 とすると int 型一つあたりの大きさは 4byte なので 16byte の領域が確保されることになります。また malloc 関数の前に(int*)と書いているのは malloc 関数は void 型のポインタを返すので、キャストしなければ int 型のポインタとして使うことができません。そして malloc は戻り値がその領域の先頭を返すので、先ほど宣言したポインタで参照します。最後に、その配列を使い終わったら領域を開放しなければなりません。free 関数を使います。今の OS はプログラム終了と同時にそのプログラムが使っていた領域

---

i 可変長配列に対応している C99 対応コンパイラならエラーはでない

ii 似たようなものとして calloc 関数というのもある

iii sizeof というのはその型が何バイトか使っているかを調べる演算子

も自動的に開放してくれるはず[i]なのですが、基本的にプログラム側が確保した変数は解放する必要があります[ii]。

ざっとの流れは説明しましたが、ポインタも挟むのでいくつか具体例を挙げましょう。

```
#include<stdio.h>
#include<stdlib.h>
int main(void){
    int num;
    double *array;
    int i;
    printf("いくつ配列を作る？¥n");
    scanf("%d", &num);
    array = (double*) malloc(sizeof(double) * num);
    for(i = 0; i < num; i++){
        printf("%d 番目の配列の値を入力してください¥n", i + 1);
        scanf("%d", &array[i]);
    }
    free(array);
    return 0;
}
```

特に何をするってプログラムではありませんが、double 型の配列を作るプログラムです。

また二次元配列は下記のように書きます。

```
#include<stdio.h>
#include<stdlib.h>
int main(void){
    int i;
    int x, y;
    int **tip;
    printf("X座標の配列の個数を決めてください¥n");
    scanf("%d", &x);
    printf("Y座標の配列の個数を決めてください¥n");
    scanf("%d", &y);
    tip = (int**) malloc(sizeof(int) * x)
    for( i = 0; i < x; i++){
        tip[i] = (int*) malloc(sizeof(int) * y);
    }

    //領域が確保された後の処理
```

---

i　この機能が甘いとどんどん PC のメモリが圧迫されていき、再起動しなければいけないようになる

ii　そうは言っても結構忘れやすいもの。しかも解放のタイミングが結構難しい

```
    for( i = 0; i < x; i++){
        free(tip[i]);
    }
    free(tip);
    return 0;
}
```

多次元配列の場合はより書き方が煩雑になります。上記の例は二次元配列ですが、二次元配列というものは配列の配列です。なのでポインタのポインタ[i]を使わなければなりません。

第三節　new 演算子、delete 演算子

せっかくなので C++で追加された機能、new 演算子と delete 演算子について説明します。

C 言語では malloc 関数と free 関数を使って配列を実現していましたが、関数という形なので言語の仕様とはいえない存在でした。C++になって malloc と free に代わる存在として new 演算子と delete 演算子というものが追加されました。

```
#include<stdio.h>
int main(void){
    int num;
    int *array;
    scanf("%d", &num);
    array = new int[num]
    //処理
    delete []array;
    return 0;
}
```

new と delete を使うので上記のプログラムの拡張しは「.cpp」にしてください。sizeof 演算子を使わなくなった分、幾分か綺麗なソースコードになりました。やっていることは先ほどの malloc と free と同じです。new が malloc、delete が free の役目にあたります。delete []array というところが不思議な書き方をしていますが、これで array ポインタの指し示した先の解放を行っています。

また二次元配列も malloc 関数と同じようになります。

```
#include<iostream>
using namespace std;
int main(void){
    int x, y;
    int** tip;
    cout << "配列のXとYの値を入れてください" << endl;
    cin >> x >> y;
```

i　三次元以上となるとポインタのポインタのポインタが登場する。例によってアスタリスクが三つ並ぶ

```
        tip = new int*[x];
        for(int i = 0; i < x; i++) tip[i] = new int[y];
        //何らかの処理
        for(int i = 0; i < x; i++) delete []tip[i];
        delete []tip;
        return 0;
    }
```

少し見慣れない文法がありますが気にしないでください[i]。new を使う場合も malloc と同じくポインタのポインタを使って配列を確保します。また領域の開放を行う場合も free と同じように for 文を使って一つずつ配列の解放を行っていきます。

一般に C++の環境では malloc は使わず、new と delete を使います。malloc 関数は領域を確保するだけの関数なので、初期化はは行いません。C 言語の場合だとあまり不具合は生じませんが、C++のクラスを使う場合、malloc で領域を確保するとコンストラクタが実行されません。また free にも同じ事が言えてデストラクタが働きません。なので new 演算子が使える環境なら malloc 関数を使う理由は特にありません。

i　せっかく C++の機能を使うんだから、他のところも C++でやってみた