

プリプロセッサとデータの抽象化

#include 文

```
#include <stdio.h> /* 省略可 */  
#include <math.h>  
#include "myLib.h"
```

```
int main () {  
    ...  
}
```

- ファイルの読み込み
- stdio.h, math.h は特定のフォルダにあるテキストファイル（ヘッダファイルと呼ばれる）
- 自分が作成したヘッダファイル(myLib.h) は、“ ” で名前を囲み、見える（パスが通った）フォルダに置く
- ファイルがその位置にそのまま読み込まれる

ヘッダファイルの構成

- #define, typedef, enum, struct 等の宣言部を別のファイルに記述する → ヘッダファイル fileName.h の作成
- 関数は呼び出される前に宣言する必要がある
→ ヘッダファイルで入出力の仕様だけ宣言しておく

```
void getNumOfEachCarType (CarInfo *dp, void *numC);  
void listIterator (CarInfo *, void (*)(CarInfo *, void *), void *);
```

 - 末尾の「;」を忘れてはいけない！
 - 変数名は省略できる
- 作成したヘッダファイルは、それが置かれている場所を、実行時の場所から辿る(相対パス)か、ルートからの場所(絶対パス)を記述
例:

```
#include "../header/fileName.h" /* 相対パス */  
#include "/home/0/sk003/header/fileName.h" /* 絶対パス */
```
- C++では、ヘッダファイルが主役となる！

#define 文: マクロ

- 文字列の置き換え

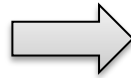
```
#define MAXCHAR 100  
#define MAXLINE 100
```

```
for (i = 0; i < MAXLINE; i++) {  
    for (j = 0; j < MAXCHAR; j++) {  
        process (i, j, ...);  
    }  
}
```

前
処
理

```
for (i = 0; i < 100; i++) {  
    for (j = 0; j < 100; j++) {  
        process (i, j, ...);  
    }  
}
```

数値だけでは、
その意味が伝わらない



```
#define MAXCHAR 100  
#define MAXLINE 10  
としたい場合、間違う可能性！
```

引数付マクロ

```
#define larger (x,y) (x>y ? x:y)
```

```
int main (void) {  
    printf ("Larger value is %d\n", larger (5,2));  
}
```

マクロ展開後

```
int main (void){  
    printf ("Larger value is %d\n", (5>2 ? 5:2));  
}
```

- 補足：？演算子

条件式？ 真の場合の値：偽の場合の値

引数付マクロ or 関数

- マクロにする場合
 - 命令部が少なく, 頻繁に使用する機能
 - 高速に処理したい場合
 - しかし, 関数化しても, それほど変わらない...
- マクロにしない場合
 - 頻繁に書き換える⇒バグを招きやすい
 - 処理が長い場合⇒読み辛くなる

enum を用いた数値化

- 文字列で表される集合の要素を数値的に扱いたい.
例えば色の3原色の要素名に数値を割り振る場合:

```
#define RED 0
```

```
#define GREEN 1
```

```
#define BLUE 2
```

- しかし, 0,1,2 の値には特別な意味が無いので, 自動的に割り振りたい → enum を使用する

```
enum Color { RED, GREEN, BLUE };
```

で同様に数値が割り振られる.

enum の使用例

```
enum Color { RED, GREEN, BLUE };
```

```
int main () {
```

```
    enum Color background, foreground;
```

```
    background = RED; /* OK */
```

```
    foreground = BLACK;
```

```
× if (background == WHITE)
```

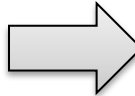
```
×     ...
```

「enum Color」が
型の様に使用できる！

enum Color の宣言に
含まれていないので
コンパイル時にエラーとなる！

typedef を用いた型定義

- 型に新たな名前を与えるために用いる
 - typedef <定義済みの型> <新たな型名>;

short color;		int color;
short *colorArray;		int *colorArray;

型名の定義

color の型と colorArray の型も
ポインタが参照する型は一致させる
必要がある → 自動的に一致させたい！

```
typedef short Color;  
Color color; /* 型名は大文字で始まる */  
Color *colorArray;
```

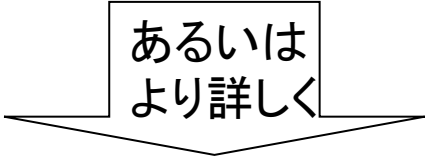
typedef を用いた配列定義

```
#define ROW 3
```

```
#define COL 3
```

```
typedef float Matrix[ROW][COL];
```

あるいは
より詳しく



```
typedef float Scalar;
```

```
typedef Scalar Vector[COL];
```

```
typedef Vector Matrix[ROW];
```

使用方法:

```
int main () {  
    Matrix mat;  
    for (i=0; i<ROW;...)  
        for (j=0; j<COL;...  
            mat[i][j] = ...  
            ...  
}
```

ポインタのポインタ

```
typedef float Scalar;  
typedef Scalar *Vector;  
typedef Vector *Matrix;  
Vector v;  
Matrix m;  
m = (Matrix) malloc (sizeof (Vector) * col);  
for (i = 0; i < col; i++) {  
    p = (Vector) malloc (sizeof (Scalar) * row);  
    ;  
}
```

構造体の抽象化

- typedef を用いて型として扱う
 - struct の指定が省ける

省略可能

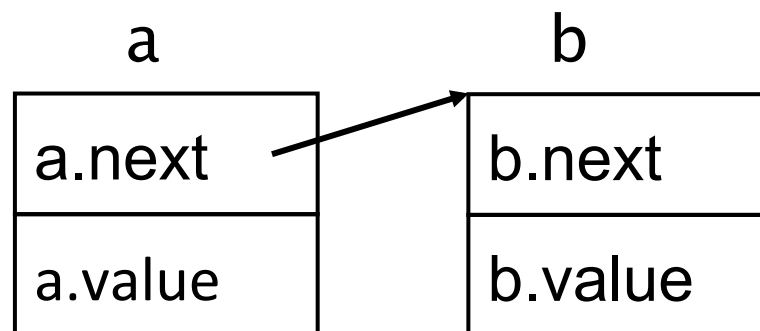
```
typedef struct word {  
    char    *string;  
    int     num;  
} Word;
```

Word w; // struct word w; と同じ
w.string = "tut";

再帰的な構造体

- 中身に自分の型へのポインタを持つ
 - ポインタを辿ると同じ型のデータがある
- 単方向リスト
 - 多数の同じ型のデータを扱うが、配列より並び替えが容易
 - 但し、頭から探すため手間がかかることもある

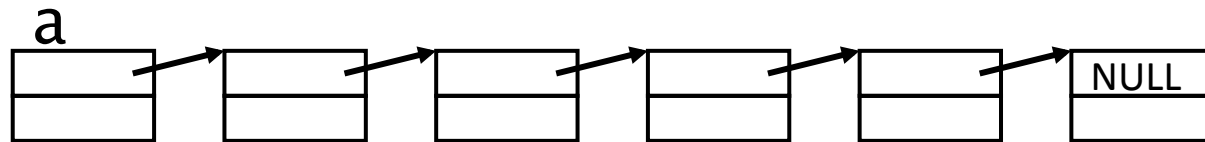
```
typedef struct list {  
    struct list *next;  
    int          value;  
} List;  
List a, b;  
a.next = &b;
```



**(a.next) が b になる*

リスト構造

- 端から探すときにリストの終わりは？
 - 次が無ければよい
 - 値が0のポインタ →ヌルポインタ
 - stdio.h に NULLの値が定義されている



```
List *p;  
for(p=&a; p != NULL; p = p->next) {  
    ...  
}
```

リスト構造の利点と欠点

- データ全体の規模を考慮する必要が無い
 - 構造体の単位毎にデータの追加や削除が可能
 - 配列や malloc を用いたメモリの動的割付けでは、確保する領域の大きさを確定する必要がある
- データを効率よく挿入／削除できる
 - 配列では、順番（インデックス）の総入替が発生
 - × メモリ領域とデータを順番に辿る効率は落ちる
- C++、Javaでは標準ライブラリを利用できる
 - 概念はしっかりと習得しておく必要がある