Today's contents

- Review for "structure"
- Review for "linked list"

- How to make a library
  - make
  - Static library and dynamic library
  - Header file, to link a library

- Basic knowledge for this week's exercise
  - Postscript

今日の内容

- 構造体のおさらい
- 線形リストのおさらい

- ライブラリの作り方
  - make
  - 静的ライブラリと動的ライブラリ
  - ヘッダファイル、ライブラリのリンク

- 課題用基礎知識
  - Postscript

Review for Structure

- Array
  - Gathering and treating plural data
  - But, only for the same type data

- Structure
  - Gathering and treating different type data
  - You can also make it an array of structure.

構造体のおさらい

- 配列
  - 複数のデータをまとめて扱うことができる
  - ただし、同じ型のデータだけ
- 構造体
  - 異なるデータ型をまとめて扱う
  - 構造体自体を配列にすることもできる

Declaration of structure template

- Declare a structure template combining plural data types (structure members)
  - This is just “declaration” (not “definition”) so that the real memory area is still not allocated.

≪Format≫

```
struct Identifier_Name {
  Data_type1 Variable_name1;
  Data_type2 Variable_name1;
  ...
  Data_typeN Variable_nameN;
};
```

≪Example≫

```
struct Student {
  int gender;            // 0:Male, 1:Female
  char id[10];           // Student ID
  char name[20];
  char bloodType[3];
  double weight;
  double height;
};
```

構造体テンプレートの宣言

- 複数のデータ(構造体メンバー)をまとめて、1つの構造体テンプレート(構造体のデータ構造)を宣言する
  - 「宣言」であるため、まだ実体はない(「定義」ではない)

≪書き方≫

```
struct 構造体識別子 {
  データ型1 変数名1;
  データ型2 変数名2;
  ...
  データ型n 変数名n;
};
```

≪例≫

```
struct Student {
  int gender;              // 性別 0:男, 1:女
  char id[10];             // 学籍番号
  char name[20];           // 氏名
  char bloodType[3];       // 血液型
  double weight;           // 体重
  double height;           // 身長
};
```

Definition of structure variable

- By using the structure template, define an actual structure variable (allocate actual memory for the variable)
  - The scope of a structure template is the same as for variables.

≪Format≫

```
struct  Identifier_Name  Variable_name1, Variable_name2, ....., Variable_nameN;
```

≪例≫

```
struct  Student  student1;          // define the structure variable named “student1”
struct  Student  student2[20];      // define the array of structure named “student2”
```

構造体変数の定義

- 構造体テンプレートを使って、構造体変数の実体を定義（実際にメモリを確保）
  - 構造体テンプレートの有効範囲は変数のスコープと同じ

≪書き方≫

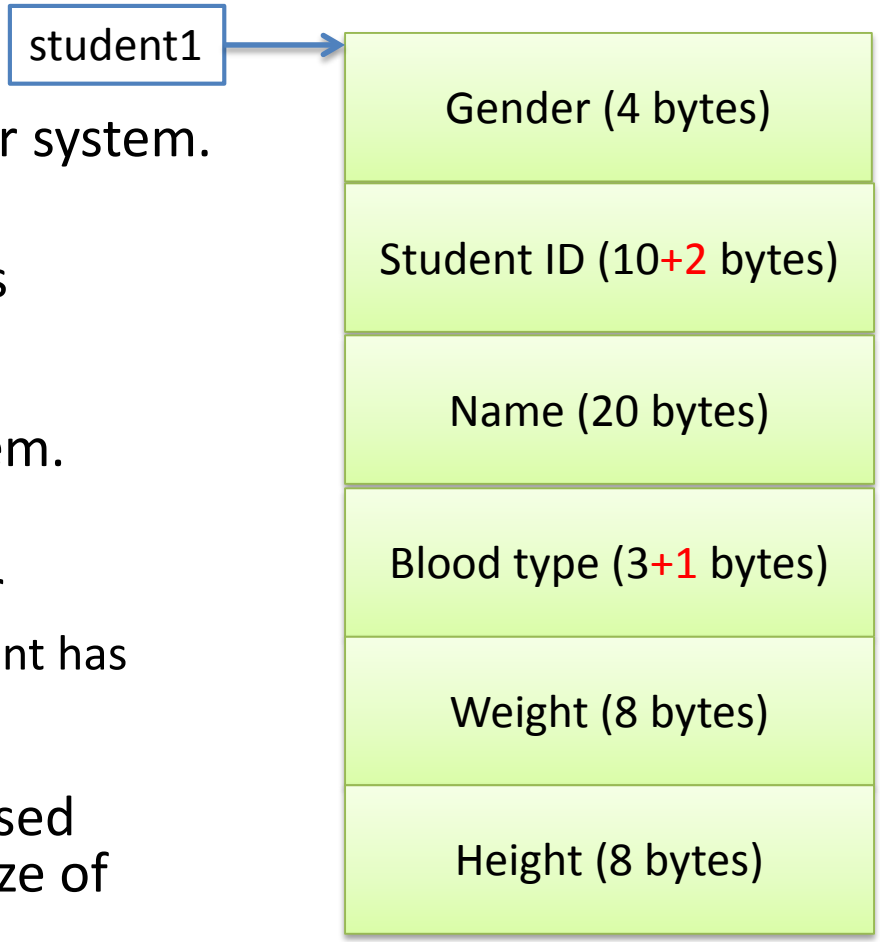
```
struct　構造体識別子　構造体変数名１，構造体変数名２，……，構造体変数名ｎ；
```

≪例≫

```
struct   Student   student1;          // student1という名前の構造体変数を定義
struct   Student   student2[20];      // student2という名前の構造体配列を定義
```

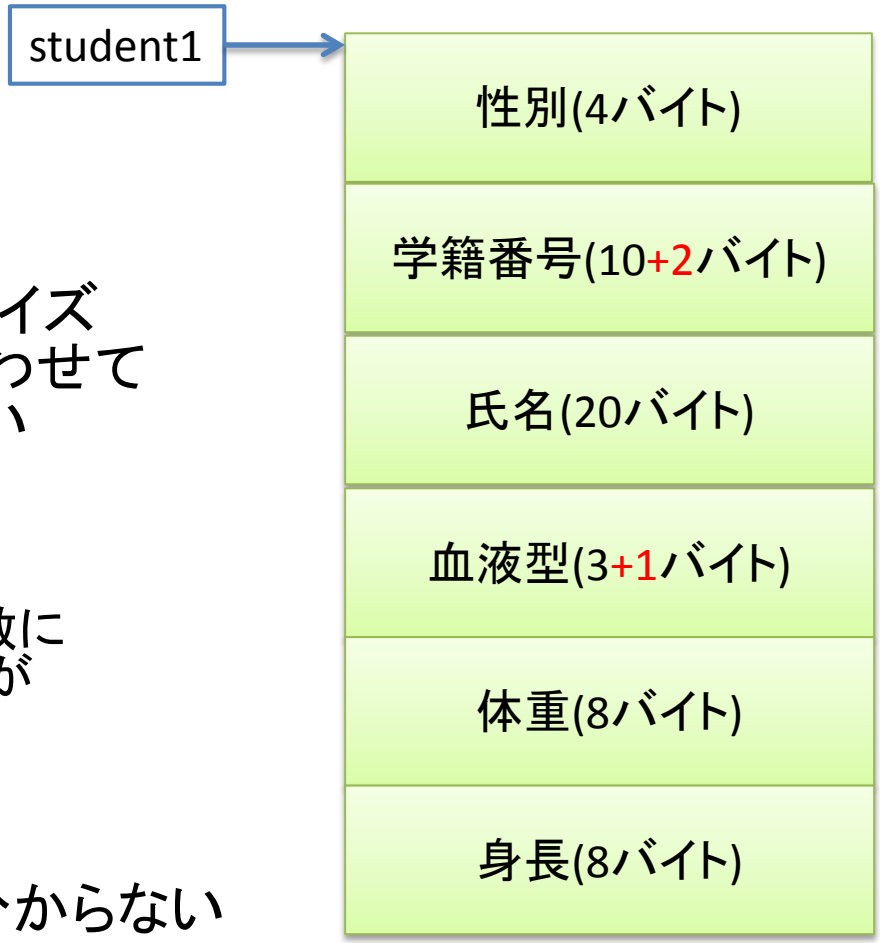
Structure and memory alignment (1)

- Actual memory allocation for a structure depends on a computer system.
- Usually, the memory allocation is done according to “word”, the byte size which is easy to be processed by the computer system.
- Figure: Example of 4-byte border
  - Use “pads” so that each component has multiple bytes of four.
- “sizeof(struct Student)” will be used to know the actual whole byte size of the structure including pads.



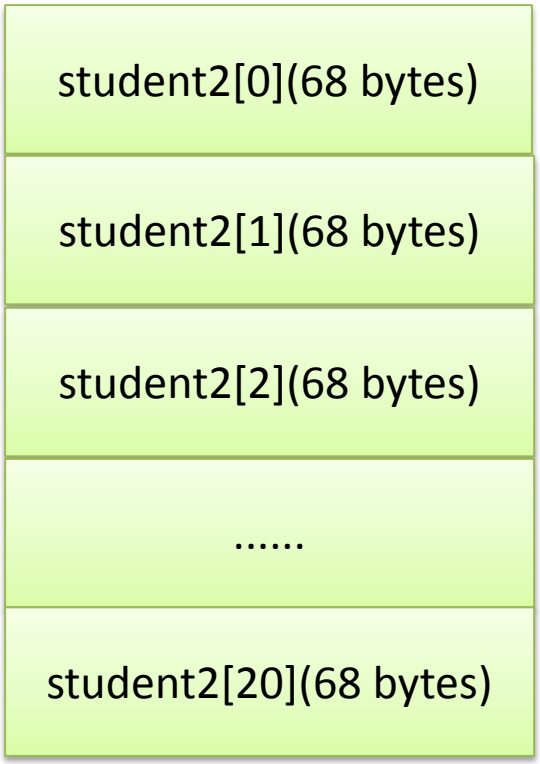
構造体とメモリ配置（1）

- メモリ上にどのように構造体が配置されるかは、処理系に依存する
- 計算機の処理しやすいバイトサイズ（ワード）があるため、それに合わせてメモリ配置が行われることが多い
- 右の図は4バイト境界の例
  - それぞれの要素が4バイトの倍数になるように、パディング（詰め物）が入れられている
- 実際の構造体全体のサイズは、sizeof(struct Student) でないと分からない



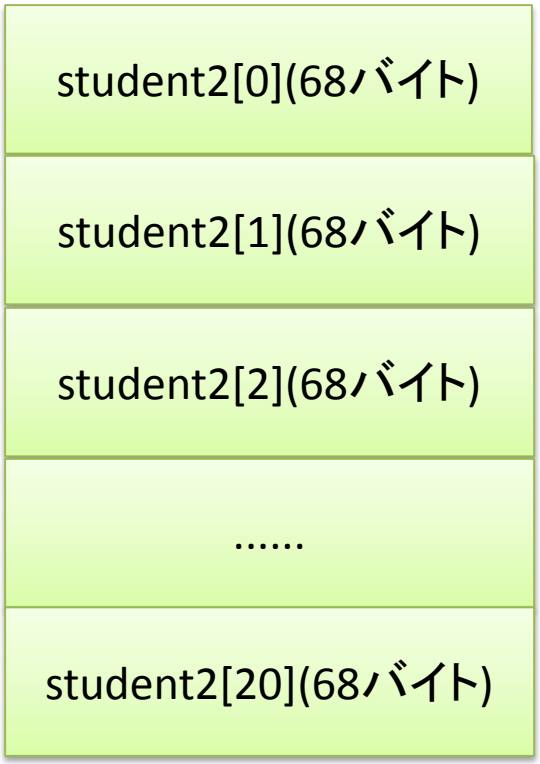
Structure and memory alignment (2)

- Former example
  - “sizeof(struct Student)” is 68 bytes.
- An array of structure has a form which aligns each structure in line on the memory including pads.



構造体とメモリ配置（2）

- 先ほどの例
  - sizeof(struct Student)は68バイト
- 構造体配列は、パディングされた構造体一つ一つがメモリ上に順番に配置される形となる



Initialization of structure

- Initialization of structure variables (members)

```
struct   Student   student1 = { 0, "B0001",“Taro Yamada", "A", 72.5, 170.0 };
```

- Initialization of an array of structures

```
struct   Student   student2[20]   =   {
    { 0, "B0003", “Jiro Tanaka", "AB", 55.9, 168.6 },
    { 1, "B0009", “Hanako Suzuki", "O", 45.0, 157.2 },
    { 0, “B0015”, “Saburo Sato”, “B”, 85.2, 180.2 },
};
```

  - Write as the same as the initialization of an array
  - Write members in order as the declaration

構造体の初期化

- 構造体変数の初期化

```
struct  Student  student1 = { 0, "B0001","山田太郎", "A", 72.5, 170.0 };
```

- 構造体配列の初期化

```
struct  Student  student2[20]  =  {
    { 0, "B0003", "田中二郎", "AB", 55.9, 168.6 },
    { 1, "B0009", "鈴木花子", "O", 45.0, 157.2 },
    { 0, “B0015”, “佐藤三郎”, “B”, 85.2, 180.2 },
};
```

  - 配列の初期化と同じような感じで
  - 宣言通りのメンバーの順序で書く

Referring a structure

- Use “.”（dot operator）
  - “（structure_name）.（member_name）”
- Referring a structure variable

```
printf("%d %s %s %s %3.1f %4.1f¥n¥n",
    student1.gender, student1.id, student1.name, student1.bloodType,
    student1.weight, student1.height);
```

- Referring an array of structures

```
for (i = 0; i < 3 ; i++ ) {
    printf("%d %s %s %s %3.1f %4.1f¥n",
        student2[i].gender, student2[i].id,
        student2[i].name, student2[i].bloodType,
        student2[i].weight, student2[i].height);
}
```

構造体の参照

- .（ドット演算子）を用いる
  - 「（構造体名）.（メンバー名）」
- 構造体変数の参照

```
printf("%d %s %s %s %3.1f %4.1f¥n¥n",
    student1.gender, student1.id, student1.name, student1.bloodType,
    student1.weight, student1.height);
```

- 構造体配列の参照

```
for (i = 0; i < 3 ; i++ ) {
    printf("%d %s %s %s %3.1f %4.1f¥n",
        student2[i].gender, student2[i].id,
        student2[i].name, student2[i].bloodType,
        student2[i].weight, student2[i].height);
}
```

Structure and function (1)

- Passing a structure to a function
  - function(student1);
    - “Call by Value” ⇒ Passing a copy of the structure to the function
    - How to refer a structure in the function: “.” (dot operator) (the same as the former example)
  - function(&student1);
    - “Call by Value” of an address ⇒ Passing an address of the structure to the function
    - How to refer a structure in the function: “->” (arrow operator)
      - “(pointer_to_structure)->(member_name)”
      - Refer a member without a pointer operator “*”.

構造体と関数（1）

- 構造体を関数に渡す
  - function(student1);
    - 値渡し ⇒ 構造体のコピーが関数に渡される
    - 関数内での構造体の参照は先ほどの例と同じ（ドット演算子）
  - function(&student1);
    - アドレス渡し ⇒ 構造体のアドレスが関数に渡される
    - 関数内での構造体の参照には、”->”（アロー演算子）を用いる
      - 「（構造体へのポインタ）->（メンバー名）」
      - ポインタ演算子（*）を使わないで書ける

Structure and function (2)

- How to use an arrow operator

```
int   main()
{
    struct   Student   student1;

    printf("%d %s %s %s %3.1f %4.1f¥n¥n",
        student1.gender, student1.id, student1.name, student1.bloodType,
        student1.weight, student1.height);

    function( &student1 );
}

void function( struct  Student  *p )
{
    printf("%d %s %s %s %3.1f %4.1f¥n¥n",
        p->gender, p->id, p->name, p->bloodType, p->weight, p->height);
}
```

構造体と関数（2）

- アロー演算子の使い方

```
int   main()
{
    struct   Student   student1;

    printf("%d %s %s %s %3.1f %4.1f¥n¥n",
        student1.gender, student1.id, student1.name, student1.bloodType,
        student1.weight, student1.height);

    function( &student1 );
}

void function( struct  Student  *p )
{
    printf("%d %s %s %s %3.1f %4.1f¥n¥n",
        p->gender, p->id, p->name, p->bloodType, p->weight, p->height);
}
```

typedef and structure

- Writing “struct Student” bothers us.
- Writing that the easy way by “typedef”

≪Example≫

```
typedef struct Student {
    int gender;             char id[10];
    char name[20];          char bloodType[3];
    double weight;          double height;
} STUDENT, *STUDENTP;

int main()
{
    STUDENT student1;  // the same as “struct Student student1;”
    STUDENT *p1;       // the same as “struct Student *p1;”
    STUDENTP p2;       // the same as “struct Student *p2;”

    p2 = p1 = &student1;
}
```

typedefと構造体

- いちいち、”struct Student” と書くのが面倒だ
- typedef を使って楽をする

≪例≫

```
typedef struct Student {
    int gender;          char id[10];
    char name[20];       char bloodType[3];
    double weight;       double height;
} STUDENT, *STUDENTP;

int main()
{
    STUDENT student1; // “struct Student student1;”と同義
    STUDENT *p1;      // “struct Student *p1;”と同義
    STUDENTP p2;      // “struct Student *p2;”と同義

    p2 = p1 = &student1;
}
```

Self-reference structure

- Structure which has a pointer to the own structure in the structure declaration
  - It's just a pointer to a structure variable, not a pointer to an own instance of the structure.

```
struct KeyValueList {
    int key;
    double value;
    struct KeyValueList *next; // self-reference
};
```

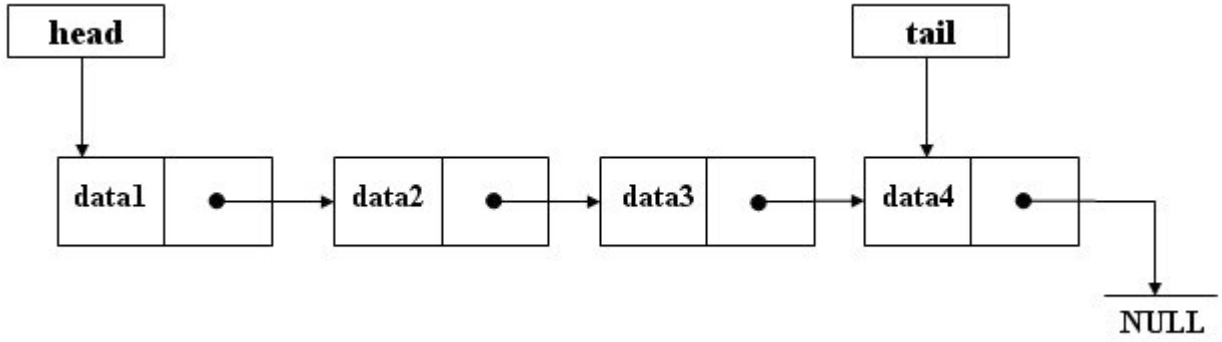
自己参照型構造体

- 構造体宣言の中で自らの構造体へのポインタを持つような構造体
  - 「構造」へのポインタであって、インスタンスとしての自分自身ではないので、混同しないこと

```
struct KeyValueList {
    int key;
    double value;
    struct KeyValueList *next; // ここが自己参照
};
```

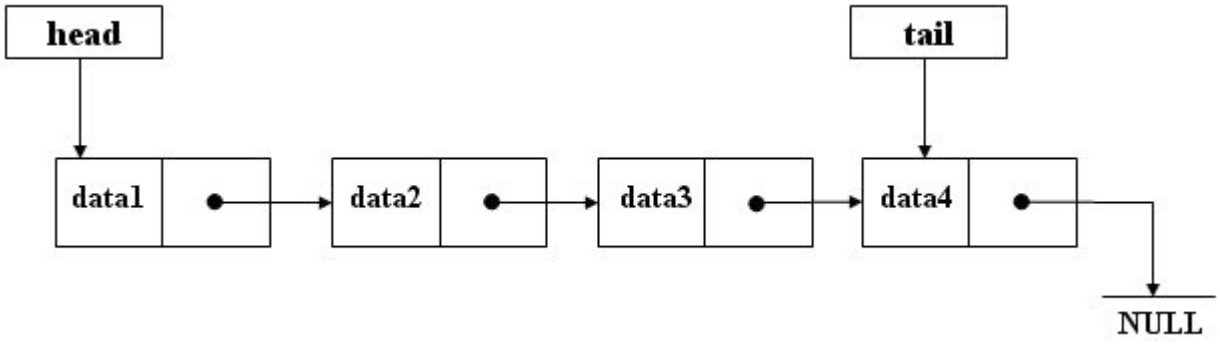
Review for Linear List

- The simplest linked list
  - unidirectional list (singly linked list), this time
  - some times, bidirectional list (doubly linked list) which has also a reverse link
- Each node has a link to the next node. The final node of the list has NULL pointer as a link.



線形リストのおさらい

- 連結リストの中でも、最も単純なもの
  - 今回は片方向リスト
  - 後ろ向きのリンクも持たせて双方向とする場合もある
- ノード毎に1つのリンクを持ち、このリンクがリスト上の次のノードを指す。リストの最後尾ならNULL値を格納する。



Insertion into linear list (1)

- Memory allocation for a list component
  - Use malloc() to allocate a struct memory area.

```
typedef  struct  KeyValueList  {
    int  key;
    double  value;
    struct  KeyValueList  *next;
}  LIST,  *LISTP;

int  main()
{
    LISTP  p  =  ( LISTP )malloc( sizeof( LIST ) );
    p->key  =  1;
    p->value  =  10.0;
    p->next  =  NULL;
}
```

線形リストへの挿入（1）

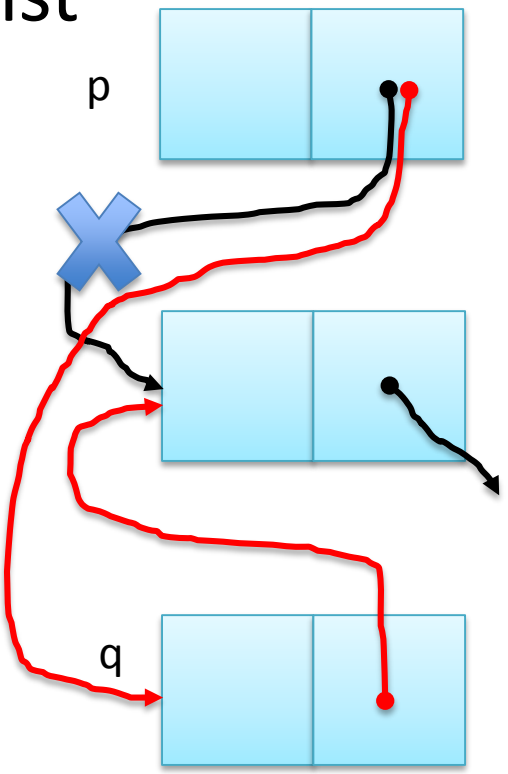
- リストの要素領域の確保
  - malloc()で構造体領域を確保

```
typedef  struct  KeyValueList  {
    int  key;
    double  value;
    struct  KeyValueList  *next;
}  LIST,  *LISTP;

int  main()
{
    LISTP  p  =  ( LISTP )malloc( sizeof( LIST ) );
    p->key  =  1;
    p->value  =  10.0;
    p->next  =  NULL;
}
```

Insertion into linear list (2)

- Insertion of a component into the list
  - Insert q into the place indicated by p

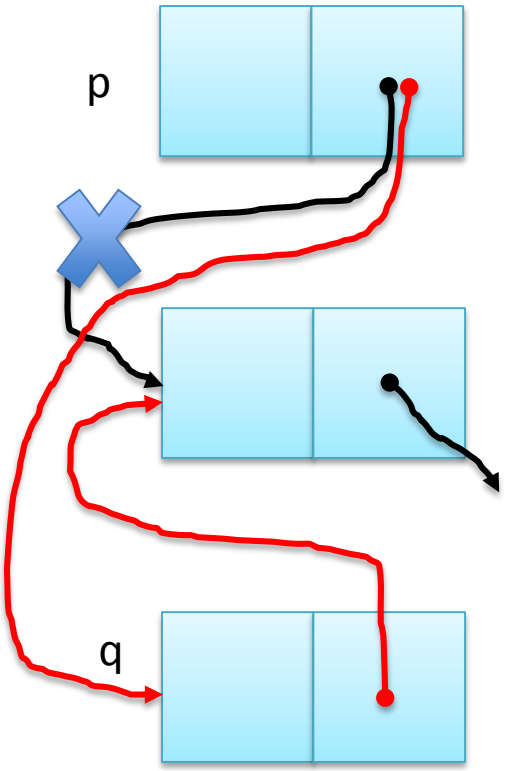
```
// Memory allocation
LISTP   q   =   ( LISTP )malloc( sizeof( LIST ) );

// Assigning values
q->key  =  1;
q->value  =  10.0;

// Change the link
q->next  =  p->next;
p->next  =  q;
```



線形リストへの挿入(2)

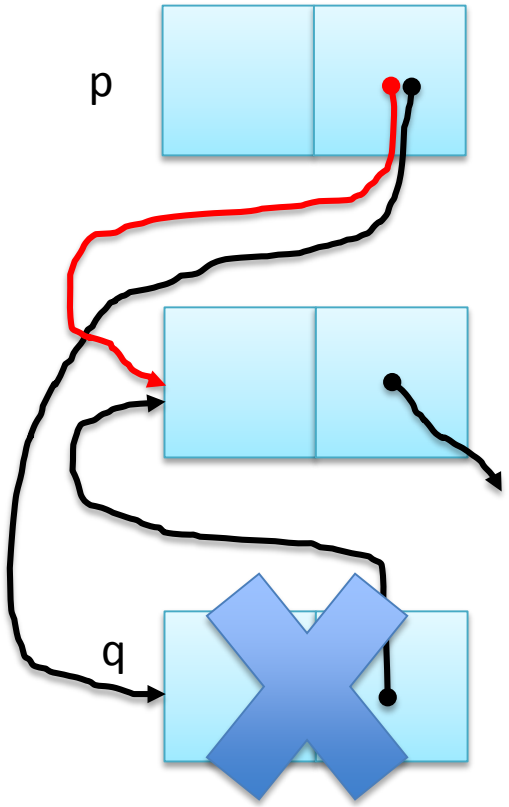
- リストへの要素の挿入
  - p が指す位置へ q を挿入する

```
// 領域確保
LISTP   q   =   ( LISTP )malloc( sizeof( LIST ) );

// 値を入れる
q->key  =  1;
q->value  =  10.0;

// リンク先の変更
q->next  =  p->next;
p->next  =  q;
```



Deletion from linear list (1)

- Delete a component from the list
  - Delete the component q next to p



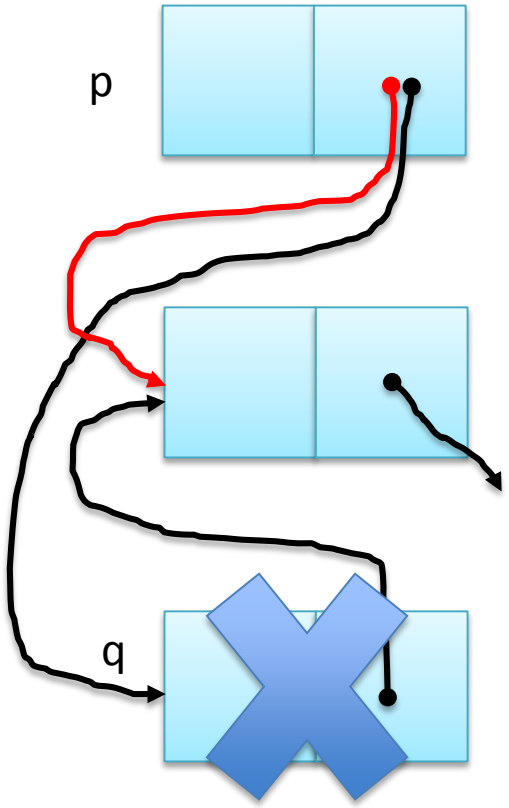

```
LISTP   q  =  p->next;

// Change the link
p->next  =  q->next;

// Free the memory
free(q);
```

  - "Segmentation Fault" if you free the memory in first.

線形リストからの削除（1）

- リストから要素を削除
  - p の次の要素 q を削除する

```
LISTP  q  =  p->next;

// リンク先の変更
p->next  =  q->next;

// メモリ解放
free(q);
```



  - 先にメモリを解放するとエラーになるので注意

Deletion from linear list (2)

- Delete a whole list

```
void   deleteMemory( LISTP p )
{
    if   ( p != NULL ) {
         deleteMemory( p->next );
         free( p );
    }
    return;
}
```

  - Delete the list by recursion
  - Call the function from the “head” node, but free() will be done from the tail node.

線形リストからの削除（2）

- リスト全体の削除

```
void   deleteMemory( LISTP p )
{
    if   ( p != NULL ) {
         deleteMemory( p->next );
         free( p );
    }
    return;
}
```

  - 再帰呼び出しにより削除を行う
  - 通常 head から呼ぶが、実際に free() が行われるのは末尾からとなる

Displaying linear list

- Follow the list from the “head” node
- The end of the list when NULL comes.
- You can write this by using recursion.

```
void  printNode( LISTP p )
{
   printf( “ %d: %f ”, p->key, p->value );
}

void printList( LISTP head )
{
   LISTP  p  =  head;
   int  i  =  1;
   while  ( p  !=  NULL )  {
      printf( “%d-th data is “,  i );
      printNode( p );
      printf( “.¥n”);
      p  =  p->next;  i++;
   }
}
```

線形リストの表示

- head から順番にリストを辿って行く
- NULLがきたらリストの末尾
- 再帰で書くこともできる

```
void  printNode( LISTP p )
{
   printf( " %d: %f ", p->key, p->value );
}

void printList( LISTP head )
{
   LISTP  p  =  head;
   int  i  =  1;
   while  ( p  !=  NULL )  {
      printf( "%d番目のデータは ",  i );
      printNode( p );
      printf( "です。¥n");
      p  =  p->next;  i++;
   }
}
```

Errors related to pointers

- “Bus Error”, “Segmentation Fault”
  - The address pointed by a pointer variable is :
    - “not existed”, “inaccessible word boundary”（Bus Error）
    - “forbidden the user’s access”（Segmentation Fault）
  - The pointer is not treated correctly either way.

  - After the definition of pointer variable,
    - the actual memory area pointed by the pointer does not exit.
    - the pointer variable is not still initialized. (e.g. top node of a list)

ポインタ周りのエラー

- “Bus Error”, “Segmentation Fault”
  - ポインタ変数の指すアドレスが
    - 「そんなアドレスはない」「アクセスが許されないword区切り」（Bus Error）
    - 「ユーザにアクセス許可がない」（Segmentation Fault）
  - どちらにしても、正しくポインタ変数が扱えてない
  - ポインタ変数を定義した後
    - ポインタ変数が指す実体領域が存在しない
    - 初期化していない（リストの先頭ノード）

Extra thing

- When compiling with gcc, ”-Wall” option can generate (many) detailed warnings.
  - Modify your code to stop the warnings !

- It is really wrong that you think “the code has warnings certainly, but the code can run !”
  - The compiler cannot find the bugs associated with pointers very often, then it causes a segmentation fault.
  - The situation can be improved frequently by stopping the warnings.

おまけ

- gccでコンパイルするときに、”-Wall” オプションを付けておくと、細かい warning をたくさん出してくれる
  - warningが出ないようにプログラムを修正すべし
- 「warning が出ていても問題なく動くからほっといても大丈夫」などと思っていたら大間違い！
  - ポインタ周りの不具合はコンパイラがエラーとして見てくれないケースも多く、Segmentation Faultになる
  - warningを潰すことで、状況が改善されることも多々ある

make (1)

- It is very hard to manage your code by only ONE file when the code size expands.
  - The code should be divided into some files necessarily.

- Compile the divided source files
  $ gcc –o execfile src1.c src2.c src3.c ...
  - Long command line
    - You can write a script for compiling.
  - Compiles all the files both changed ones and unchanged ones.
    - Waste of computational resources

- Compile each file by hand
  $ gcc –c src1.c
  - An object file “src1.o” will be generated.
  - By linking the object files, an execution file is generated.

make（1）

- プログラムが大きくなってきたら、1つのソースファイルでプログラムを管理するのは大変
  - 必然的にソースファイルを分割することになる

- 分割したソースファイルをコンパイルする
  $ gcc –o execfile src1.c src2.c src3.c ...
  - コマンドラインが長い
    - コンパイル用のscriptを書いておくという手はある
  - 変更したファイルもしてないファイルも一緒にコンパイルする
    - 計算機資源の無駄遣い

- 手動で分割コンパイルする
  $ gcc –c src1.c
  - オブジェクトファイル “src1.o” が生成される
  - オブジェクトファイルをリンクすることで、実行ファイルを生成する

make (2)

- We want to re-compile automatically only changed files.....
  - Here “make” !
  - Re-compile the source files which have newer time stamp than the object file.

- How to use
  - Write dependency between files and compiling rules on “Makefile” (or “makefile”)
  - In the directory which includes the Makefile, from command line
    $ make
    $ make –f (the_name_of_Makefile)

- Makefile contents are very important !

make（2）

- 変更したファイルだけ自動的にコンパイルしなおしてくれないかなぁ……
  - 「make」の登場！
  - オブジェクトファイルのタイムスタンプより新しいソースファイルだけコンパイルし直す

- 使い方
  - “Makefile”（または“makefile”）にコンパイル時に必要なファイルの依存関係とコンパイルのルールを記述
  - Makefileのあるディレクトリでコマンドラインから
    $ make
    $ make –f (Makefile名)

- Makefileの中身が重要！

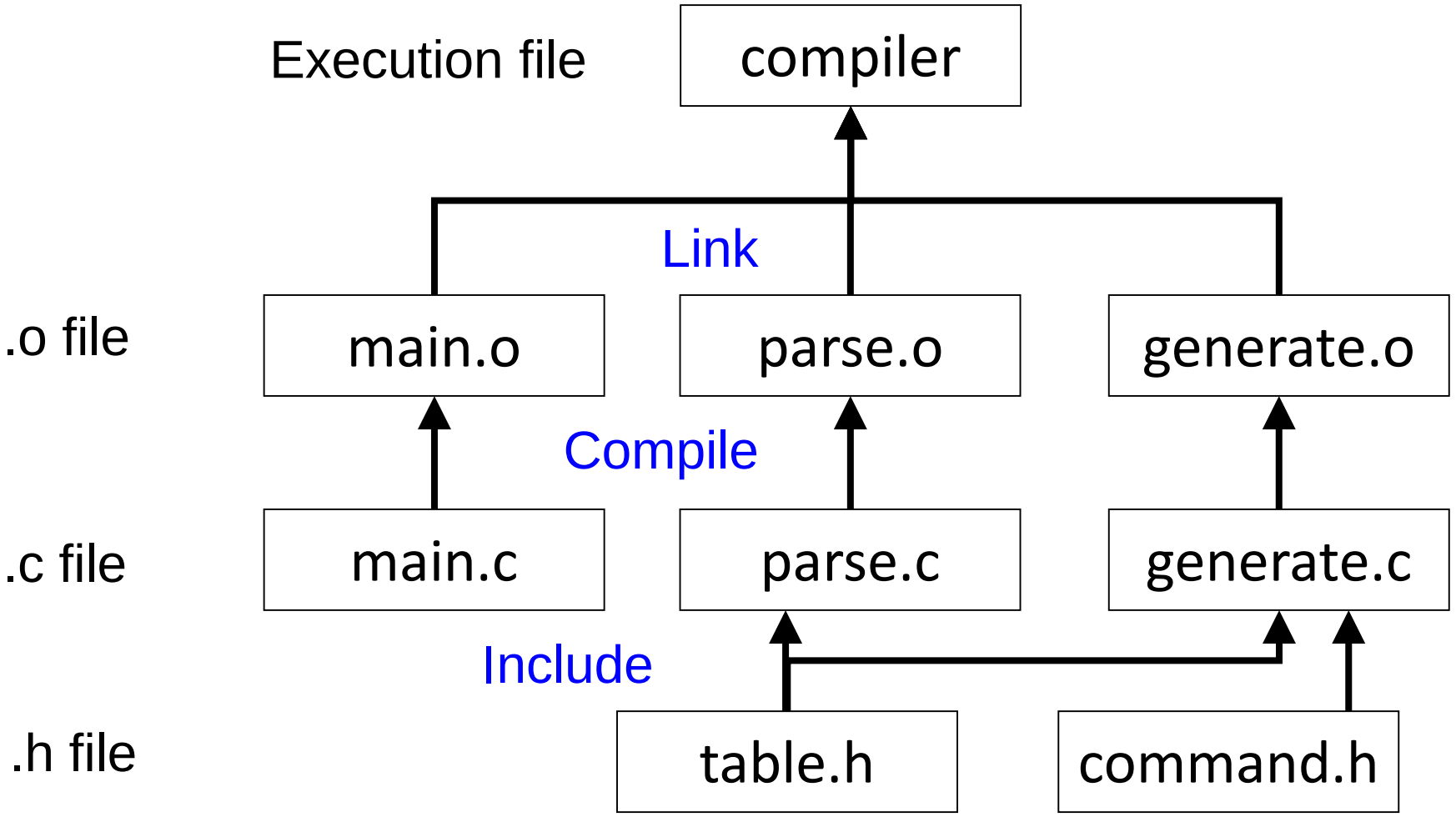
Example of Makefile (1)

```
CC = gcc
OPTS = -O2 –Wall
OBJS = main.o parse.o generate.o
TARGET = compiler

$(TARGET) : $(OBJS)
    $(CC) $(OBJS) -o $(TARGET) $(OPTS)

main.o : main.c
    $(CC) main.c -c $(OPTS)
```

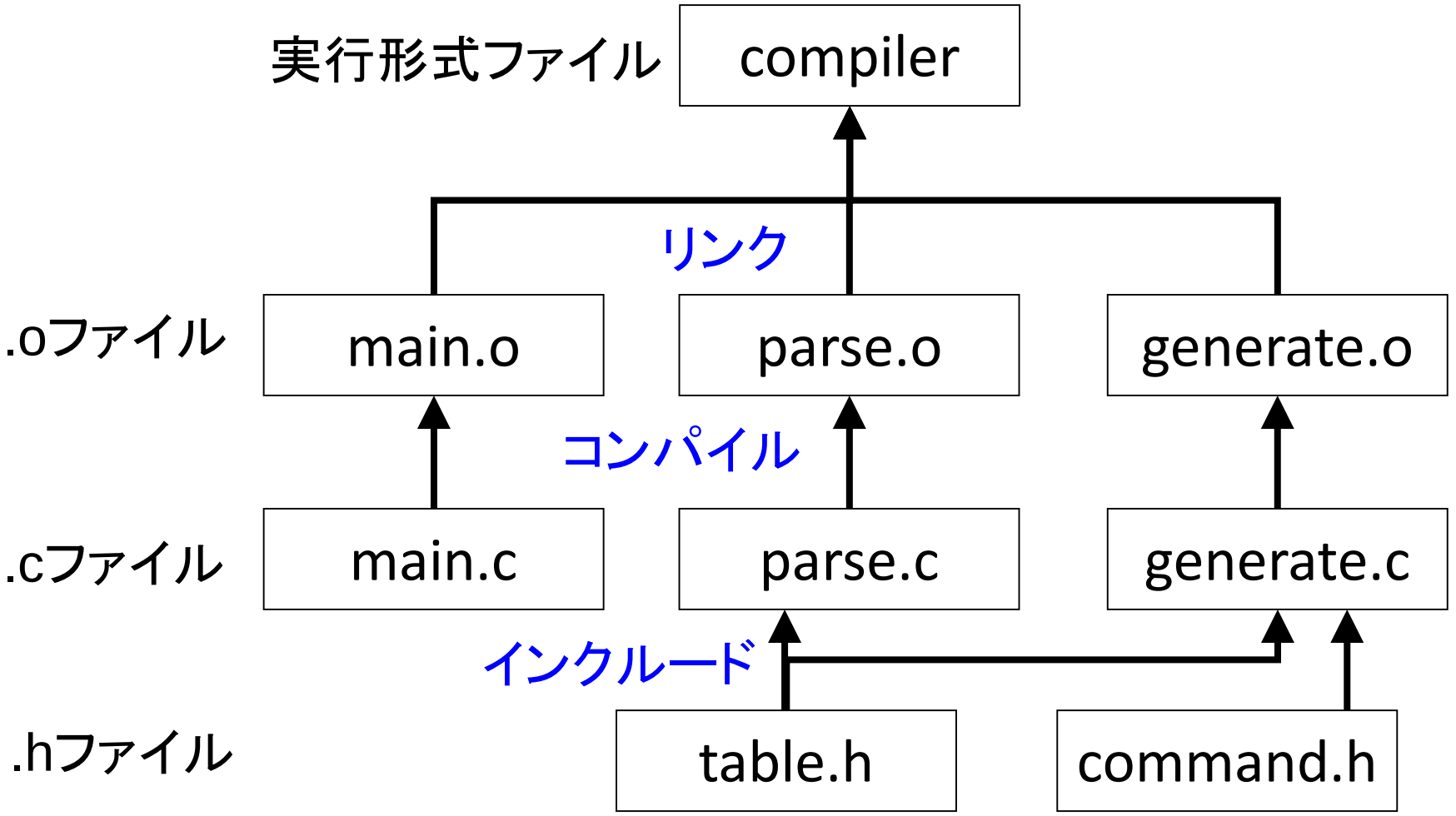
Makefileの記述例（1）

```
CC = gcc
OPTS = -O2 –Wall
OBJS = main.o parse.o generate.o
TARGET = compiler

$(TARGET) : $(OBJS)
    $(CC) $(OBJS) -o $(TARGET) $(OPTS)

main.o : main.c
    $(CC) main.c -c $(OPTS)
```

Example of Makefile (2)



Makefileの記述例（2）



Special characters in Makefile

- $(variable)
  - Refer the variable
- $@
  - Refer the current target placed on the left side of “ : ”
- $<
  - Replaced by a file which should be compiled in suffix rules.
  - If the target is “.c.o”, source files with newer time stamp will be compiled and the object files are generated.

Makefileの特殊記号

- $(変数)
  - 変数を参照する
- $@
  - “:”の左側にある、現在処理しているターゲットを参照する
- $<
  - サフィックスルール中でコンパイルすべきファイルの1つに置き換えられる
  - ターゲットが ”.c.o” であれば、タイムスタンプの新しいC言語ソースファイルをコンパイルし、オブジェクトファイルを生成する

Library (1)

- It is helpful to keep functions you use frequently as object files.
- However, the management of many object files independently bothers us.

- Manage by gathering ! ⇒ Library

- “static” library and “dynamic” library

ライブラリ（１）

- よく使う関数はオブジェクトファイルとして置いておくと便利
- しかし、バラバラで管理するのは面倒

- まとめて管理！⇒ライブラリ

- 静的（static）ライブラリと動的（dynamic）ライブラリがある

Library (2)

- Static library
  - A compiler links a library (object files) when compiling.
  - An execution file can run by only itself.
  - The execution file size will be rather large.

- Dynamic library
  - A computer system links a library when running the program.
  - If the library cannot be found when running the program, the program cannot run.
  - The execution file size will be rather small.

ライブラリ（２）

- 静的（static）ライブラリ
  - コンパイル時にライブラリ（オブジェクトファイル）のリンクを行う
  - 実行ファイルのみで実行可能
  - 実行ファイルのサイズは大きくなる

- 動的（dynamic）ライブラリ
  - 実行時にライブラリのリンクを行う
  - 実行時にライブラリが見つからないと、プログラムが実行できない
  - 実行ファイルのサイズは小さくなる

How to make a library

- Static library
  $ ar -ruv $(library_name)
  $(object_file_name) ……
  $ ranlib $(library_name)

- Dynamic library
  $ gcc -shared -o $(library_name)
  $(object_file_name) ……

- Please look “Makefile” of the this week’s Exercise
- Please check other options of “ar” by yourself.

ライブラリの作り方

- 静的ライブラリ
  $ ar -ruv $(ライブラリ名)
  $(オブジェクトファイル) .....
  $ ranlib $(ライブラリ名）

- 動的ライブラリ
  $ gcc -shared -o $(ライブラリ名)
  $(オブジェクトファイル) .....

- 今回の課題のMakefileを参照のこと
- arの他のオプションについては自分で調べましょう

Makefile of this week's Exercise

```
CC      = gcc
CFLAGS  = -Wall -O -I.

TARGET = libmat.a libmat.so
SRCS   = vector.c matrix.c
OBJS   = $(SRCS:.c=.o)

all: $(TARGET)

libmat.a: $(OBJS)
        ar -ruv $@ $(OBJS)
        -ranlib $@

libmat.so: $(OBJS)
        $(CC) -shared -o $@ $(OBJS)

.c.o:
        $(CC) $(CFLAGS) -c $<

clean:
        -rm $(TARGET) $(OBJS)

vector.o: vecmat.h
matrix.o: vecmat.h
```

You want to make "libmat.a" (static library) and "libmat.so" (dynamic library)

Source files are "vector.c" and "matrix.c"

Object files are "vector.o" and "matrix.o". These depend on a common header file "vecmat.h".

今回の課題のMakefile

```
CC      = gcc
CFLAGS  = -Wall -O -I.

TARGET = libmat.a libmat.so
SRCS   = vector.c matrix.c
OBJS   = $(SRCS:.c=.o)

all: $(TARGET)

libmat.a: $(OBJS)
        ar -ruv $@ $(OBJS)
        -ranlib $@

libmat.so: $(OBJS)
        $(CC) -shared -o $@ $(OBJS)

.c.o:
        $(CC) $(CFLAGS) -c $<

clean:
        -rm $(TARGET) $(OBJS)

vector.o: vecmat.h
matrix.o: vecmat.h
```

作りたいライブラリは、
libmat.a (静的ライブラリ) と
libmat.so (動的ライブラリ)

ソースファイルは、
vector.c と matrix.c

オブジェクトファイルは、
vector.o と matrix.o
これらは、共通のヘッダーファイル
vecmat.h に依存している

How to use libraries

- Linking will be done with "-l"（lower-case of L） option by gcc
  - Linking libraries which do not link as standard
    - Standard libraries are linked without special treatments.
  - the same for both static and dynamic library

- Linking your own writing library
  - -L(library_directory) -l(library_name)
  - -L. -lmat

ライブラリの使い方

- gccでは"-l"(小文字のL)オプションを使ってリンク
  - 標準でリンクされないライブラリのリンク
    - 標準的なライブラリは何もしなくてもリンクされる
  - 静的ライブラリも動的ライブラリも同じ

- 自作ライブラリのリンク
  - -L(ライブラリの場所) -l(ライブラリの名前)
  - -L. -lmat

How to check dynamic library

- Which dynamic library dose the execution file need ?

- Linux, etc.
  - “ldd” command
    $ ldd execution_file_name
- MacOS X
  - “otool” command
    $ otool -L execution_file_name

- You can find that standard libraries are linked dynamically to the execution file.

動的ライブラリの調べ方

- 動的ライブラリを必要とする実行ファイルが、どの動的ライブラリを必要としているのか？

- Linux等
  - ldd コマンド
    $ ldd 実行ファイル名
- MacOS X
  - otool コマンド
    $ otool -L 実行ファイル名

- システム標準ライブラリは動的にリンクされていることが分かる

Header file (1)

- A file to gather and use common declarations

- #include <stdio.h>
  - “stdio.h” is a header file.
    - #include is not only used for header files.
  - Actually, the file is at /usr/include/stdio.h
  - By surrounding with <>, the notation is construed as “a file in the default directory which the compiler specifies”.
    - You will have an error if the file was not found in the directory.

ヘッダファイル（１）

- 共通で使う宣言をまとめておいて、使い回すためのファイル

- #include <stdio.h>
  - stdio.hがヘッダファイル
    - #include で読めるのはヘッダファイルだけではない
  - 実体は、/usr/include/stdio.h
  - 「<」「>」で囲むことで、「コンパイラが指定するデフォルトディレクトリにあるファイル」と解釈する
    - そこでファイルが見つからなければエラー

Header file (2)

- #include ”myheader.h”
  - By double-quoting, search the header file in the order of the current directory → the default include directory.
  - For including a own writing header file.

- Specify the place of a header file (an including file) when compiling
  - -I option
  - (Place of a library : -L option)

ヘッダファイル（２）

- #include ”myheader.h”
  - 「”」「”」で囲むことで、カレントディレクトリ → デフォルトディレクトリの順番でファイルを探す
  - 自作ヘッダファイルの読み込み

- コンパイル時のヘッダファイル（includeするファイル）の場所指定
  - -Iオプション
  - （ライブラリの場所：-Lオプション）

Knowledge for this week's exercise (1)

- About "Postscript"
  - A "page description language" developed by Adobe
    - Do drawing processes, etc. by consturing a notation in a text file.
    - "Reverse Polish Notation" is used to write commands.
      - Notation that a command follows values.
  - Display the file graphically by Ghostscript, etc.
    - On MacOS X, when a ".ps" file is opened, the file will be transformed into pdf file automatically and displayed.
      - It's a chance to use "open" command !

本日の課題用知識（1）

- Postscriptについて
  - Adobeが開発したページ記述言語
    - テキストファイルで書かれた記述を解釈することで、描画等の処理を行う
    - 逆ポーランド記法で命令が記述される
      - 値の後ろに命令がある記法
  - Ghostscript等を用いて表示することができる
    - MacOS Xでは、”.ps”のファイルを開くと、自動的にpdfファイルに変換されて、表示される
      - openコマンドの出番だ！

Postscript commands which used today

- setrgbcolor
  - Three values which are weights for “R”, “G”, “B” ($0 \leqq x \leqq 1$) are set as a color for drawing.
- newpath
  - After this command, a new drawing set will run.
- moveto
  - Two values are x- and y-coordinates. Move to the point (the origin is on the bottom-left corner).
- lineto
  - Two values are x- and y-coordinates. Draw a line to the point from a current point with the set color.
- closepath
  - Draw a line to the point of “moveto” from a current point with the set color.
- stroke
  - Here, the drawing set runs actually.
- showpage
  - Draw whole page.

今日使うPostscriptの命令

- setrgbcolor
  - 3つの値を“R”, “G”, “B”の重み（0≦値≦1）として、その後の描画に用いる色をセットする
- newpath
  - この後、新しい描画セットを行うこと示す
- moveto
  - 2つの値をx座標、y座標（原点は用紙の左下）として、その点に移動する
- lineto
  - 2つの値をx座標、y座標として、セットされている色を用いてその点まで線を引く
- closepath
  - movetoの点までセットされている色を用いて線を引く
- stroke
  - ここで実際に描画セットを実行
- showpage
  - ページの全体の描画を行う

Knowledge for this week's exercise (2)

- Simple random number generation
  - There are some random number generation functions, but the simplest one is the system standard "rand()".
    - rand() generates a random number from 0 to RAND_MAX (defined in <stdlib.h>) uniformly.
    - To get a number from 0 to 1, do like "(double)rand() / (double)RAND_MAX"
    - "srand(seed)" can change the random number sequence.
      - The time is used as "seed".

本日の課題用知識（２）

- 手軽なランダム数生成
  - ランダムに数を生成する関数はいくつかあるが、システム標準で簡単な関数は “rand()”
    - rand() は、０～RAND_MAX（<stdlib.h>内で定義）の間の整数値をランダム（一様）に生成する
    - “(double)rand() / (double)RAND_MAX” とすることで、０～１の間の実数を生成できる
    - “srand(seed)” によって乱数系列を変えることができる
      - seed には時刻を使うことが多い