

計算機構成論

Lecture 7

C言語から実行可能なプログラムへの変換

2023年度春学期

情報理工学部 Rクラス担当

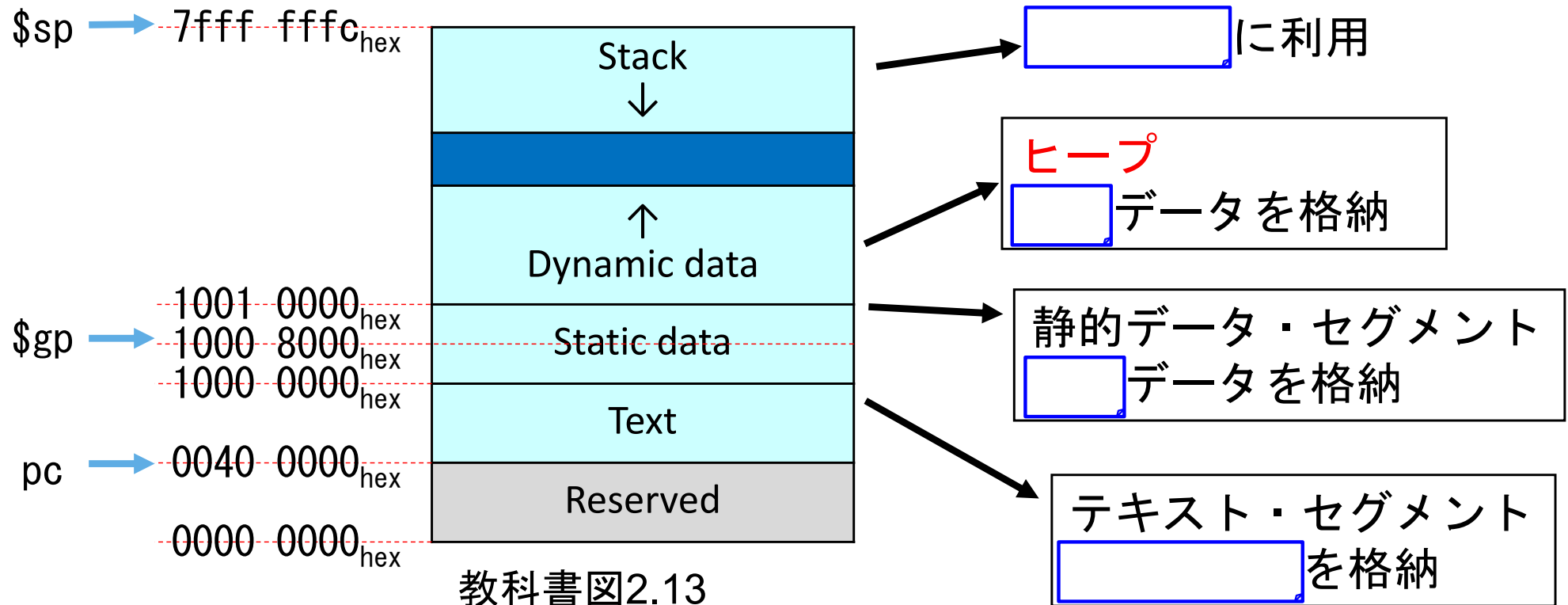
越智裕之

※ このレジュメの中で「教科書」とは、一昨年度まで教科書に指定されていた書籍のことです（Lecture 0 参照）。この書籍を入手できなくても支障なく学習できるよう、レジュメに加筆修正を行っています。

内容

- Cプログラムより実行可能な形式への変換の流れ
 - アセンブラ
 - リンカ
 - ローダ
 - Cプログラムの包括的な例題解説（演習問題④⑤）
-
- 教材：教科書2.12節(後半省略)、2.13節

メモリ上でのプログラムとデータの配置：Lec6の復習



教科書図2.13

以下のレジスタが保持しているアドレスは何か？

pc : 現在実行中の命令の格納アドレス

\$gp : 静的データのアクセスのための基準アドレス

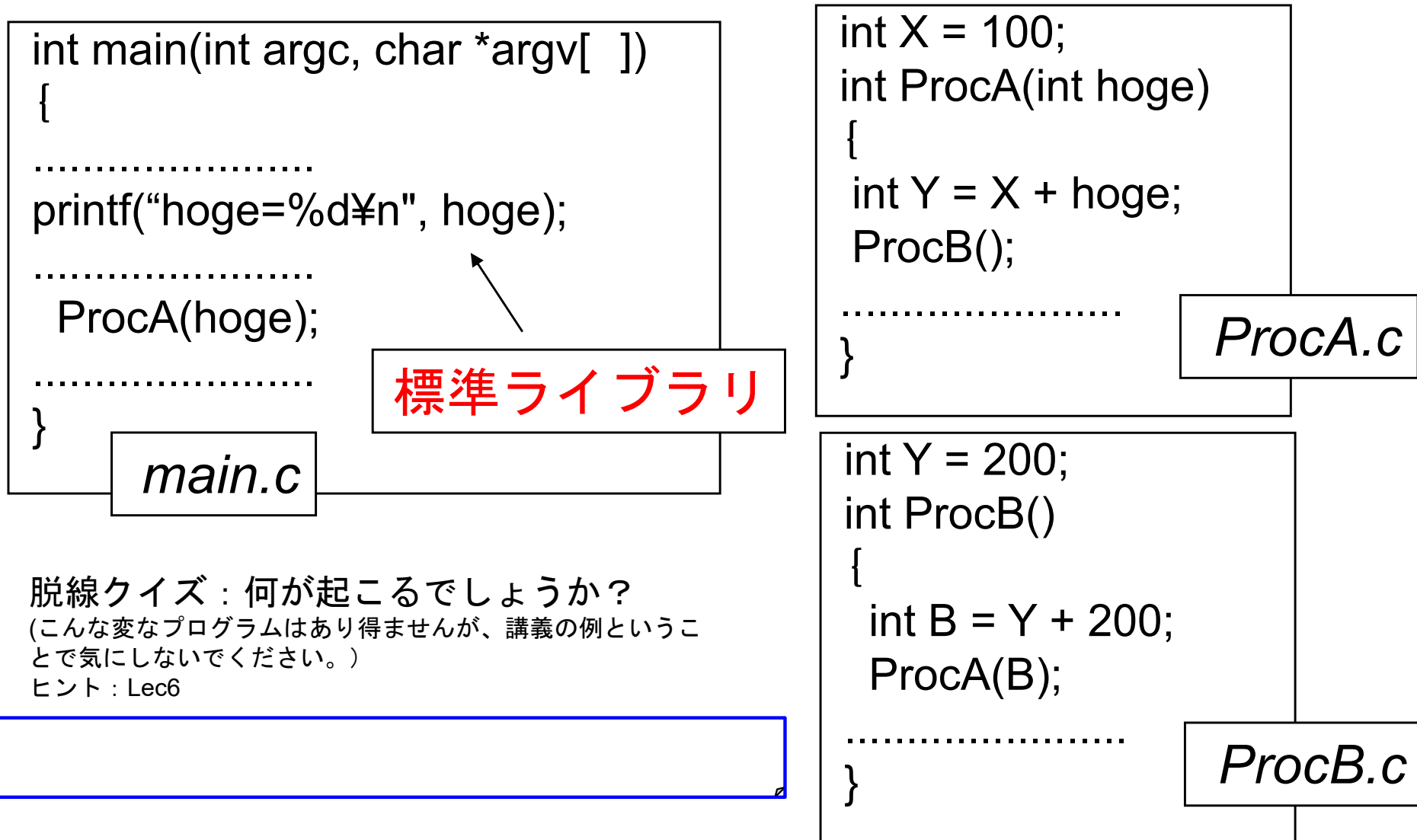
- 初期値が `1000 800016`
- 16ビットオフセットで `1000 000016` から `1000 ffff16` にアクセス可能

\$sp : 現在のスタックの一番下位のアドレス

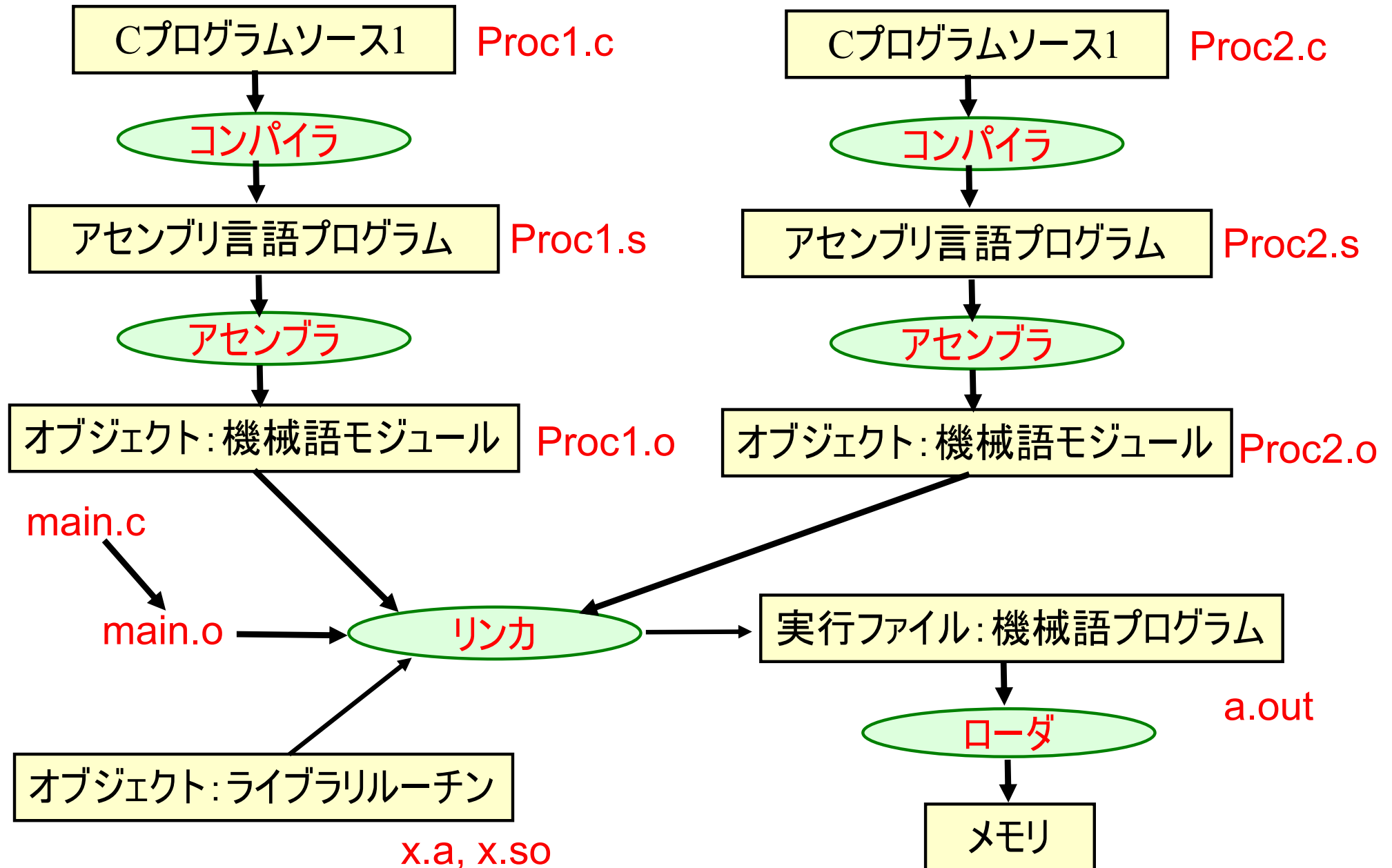
\$fp : 現在の手続きフレームの一番上位のアドレス

Cでのプログラム開発

開発のしやすさなどのために複数のファイルを使うのが普通



Cプログラムの起動の流れ



アセンブラの仕事

- **擬似命令**を実際の命令に変換

教科書(p122-p123)

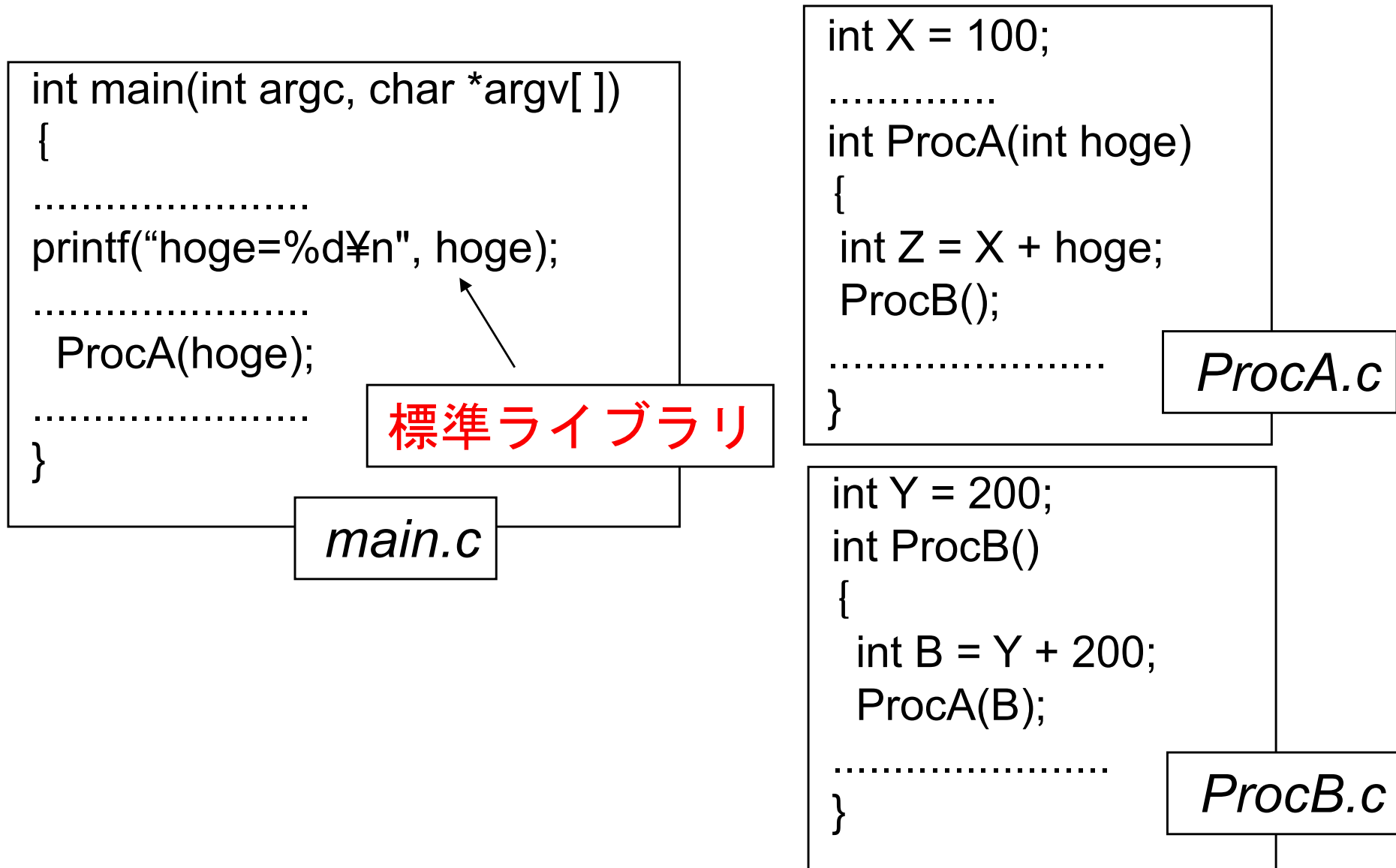
e.g., move \$t0, \$t1 → add \$t0, \$zero, \$t1
(\$t1 を\$t0に転送 →)

- (各Cのソースファイルをコンパイルしてできた)X.sを以下の情報を含むオブジェクトファイルX.oに変換

- ヘッダ(サイズの情報)
- テキスト・セグメント(機械語コード)
- 静的データ・セグメント(静的データ)
- リロケーション情報(絶対アドレスに依存する命令とデータの情報)
- シンボル表(未定義のラベルの情報)
- デバッグ情報

アセンブラのお仕事の例(1/2)

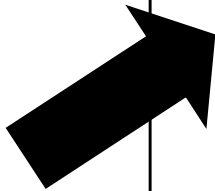
開発のしやすさなどのために複数のファイルを使うのが普通



アセンブラの仕事の例(2/2)

```
int X = 100;
int ProcA(int hoge)
{
    int Z = X + hoge;
    ProcB();
    .....
}
```

ProcA.c



ProcA.o

ヘッダ	名前	ProcA	
	テキスト・サイズ	100 ₁₆	
	データ・サイズ	20 ₁₆	
テキスト・セグメント	アドレス		
	0 ₁₆	lw \$t0, 0 (\$gp)	
	
	14 ₁₆	jal 0	
	
データ・セグメント	0 ₁₆	(X)	
	
リロケーション情報	アドレス	命令タイプ	依存関係
	0 ₁₆	lw	X
	14 ₁₆	jal	ProcB
シンボル表	ラベル	アドレス	
	X	-	
	ProcB	-	

* 数値, アドレスは
16進数とする

リンカの仕事

- 複数のオブジェクトファイルとライブラリファイルから1つの実行可能なオブジェクトファイルを作る
 - 内部および外部の参照を解決する
(データの格納されるアドレスと命令ラベルのアドレスの決定)
 - 命令コードとデータをメモリに配置する

ミニクイズ：符号付きとして、計算せよ。

$$\begin{array}{c} 1000\ 8000_{16} \\ (32\text{bit}) \end{array} + \begin{array}{c} 8000_{16} \\ (16\text{bit}) \end{array} = \boxed{}$$

演習問題 その①

以下の2つのCのファイルをコンパイルしたオブジェクトファイルを、次の2ページに示す。そしてそれらをリンクした結果をその次のページに示す。その空欄をうめよ。

(教科書の例題p124) の類題だがちょっと違う)

ただし、\$gp = 1000 8000₁₆ とする。

```
int X = 100;
int main( )
{
    int Z = X + 200;
    ProcX();
    .....
}
```

main.c

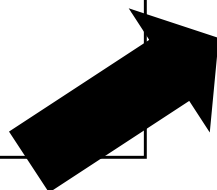
```
int Y = 200;
void ProcX()
{
    int B = Y + 200;
    .....
    return;
}
```

ProcX.c

main.o

```
int X = 100;
int main( )
{
  int Z = X + 200;
  ProcX();
  .....
}
```

main.c



main.o

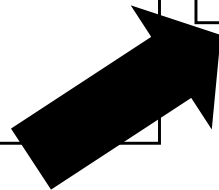
ヘッダ	名前	main	
	テキスト・サイズ	100 ₁₆	
	データ・サイズ	20 ₁₆	
テキスト・セグメント	アドレス		
	0 ₁₆	lw \$t0, 0(\$gp)	
	
<i>main.o</i>	10 ₁₆	jal 0	
	
データ・セグメント	0 ₁₆	(X)	
	
リロケーション情報	アドレス	命令タイプ	依存関係
	0 ₁₆	lw	X
	10 ₁₆	jal	ProcX
シンボル表	ラベル	アドレス	
	X	-	
	ProcX	-	

* 数値, アドレスは16進数とする

ProcX.o

```
int Y = 200;
void ProcX()
{
    int B = Y + 200;
    .....
    return;
}
```

ProcX.c



ProcX.o

ヘッダ	名前	ProcX	
	テキスト・サイズ	200 ₁₆	
	データ・サイズ	30 ₁₆	
テキスト・セグメント	アドレス		
	0 ₁₆	lw \$t1, 0(\$gp)	
	...		
データ・セグメント			
	0 ₁₆	(Y)	
	
リロケーション情報	アドレス	命令タイプ	依存関係
	0 ₁₆	lw	Y
	
シンボル表	ラベル	アドレス	
	Y	-	
		-	

演習問題 解答 (main.o と ProcX.o のリンク結果)

実行ファイル・ヘッダ		
	テキスト・サイズ	<input type="text"/>
	データ・サイズ	<input type="text"/>
テキスト・セグメント	アドレス	命令
	0040 0000 ₁₆	lw \$t0, <input type="text"/> (\$gp)

	<input type="text"/>	jal <input type="text"/>

	<input type="text"/>	lw \$t1, <input type="text"/> (\$gp)

	<input type="text"/>	jr \$ra
データ・セグメント	アドレス	
	1000 0000 ₁₆	(X)

	<input type="text"/>	(Y)

ローダの仕事

教科書p126に書いている通りで、基本的には以下の通り

- リンク後のオブジェクトファイルを主記憶に読み込む
- mainルーチンを実行する環境を用意（引数、スタック等）
- 最初の命令を呼び出す

（その他発展的なこと：p127からp128）

- 動的にリンクされるライブラリ

(DLL: dynamically linked library)

呼び出すためのダミールーチンのみを用意しておいて、必要になった時に必要な命令をテキスト領域に配置する

演習問題その②

here: **beq** \$s0, \$s2, **there**

...

there: add \$s0, \$s0, \$s0

が、アセンブラで問題が生じるのはどのような場合か？

その問題を解決するために、どのようなコード列に変換すればよいか？

演習問題その③

下記の命令のうちリンク・フェーズで編集が必要になるのはどれか.

Loop:

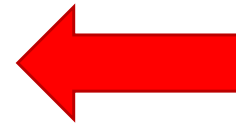
```
lui $at, 0xABCD      # a
```

```
ori $a0, $at, 0xFEDC  # b
```

```
jal add_link          # c
```

```
bne $a0, $v0, Loop    # d
```

cはサブルーチン呼び出し
なのでそのアドレスの解
決はリンク時に行われる



(dはPC相対アドレッシング
なので、リンクする前
に、その値は解決してい
ることに注意)

豆知識

- lui 命令と ori 命令で32bit定数をレジスタにセットできる旨は教科書pp.109～110に説明されている。
- \$at はレジスタ番号1のレジスタであり、MIPSでは疑似命令（教科書P.122、例えば li → lui + ori）の含まれたアセンブリコードをアセンブラが展開する際に利用する目的に予約されている。

内容

- Cプログラムより実行可能な形式への変換の流れ
 - アセンブラ
 - リンカ
 - ローダ
 - Cプログラムの包括的な例題解説（演習問題④⑤）
-
- 教材：教科書2.12節(後半省略)、2.13節

演習問題 その④

演習の④と⑤は、教科書2.13節「Cプログラムの包括的な例題解説」の内容です。変換したあとのコードの穴埋めなどをテストに出すかもしれません。

左下のswap関数をアセンブリ言語になおせ。

なお、前提条件は以下の通り

- \$a0と\$a1 に2つの引数の値(それぞれ、v[] の先頭アドレスおよび k の値)が格納されている。
- \$t0は変数 tempに使用する
- この関数からは他の関数が呼ばれないので、\$t0-\$t7以外を使用しないならば、特に退避・復帰の必要はない

```
void swap (int v[], int k)
{
    int temp;
    temp = v[k];
    v[k] = v[k+1];
    v[k+1] = temp;
}
```

演習問題 その④ の解答

```
void swap (int v[], int k)
{
    int temp;
    temp = v[k];
    v[k] = v[k+1];
    v[k+1] = temp;
}
```

- 引数は以下のレジスタで渡される
 - **\$a0** は配列 **v[]** の先頭アドレス
 - **\$a1** は変数 **k** の値
- **int**型は1語 = 4バイトであることに注意
- 解答は下の通り (教科書P.132より)

手続き本体

swap: sll	\$t1,	\$a1,	2	# k * 4 をレジスタ \$t1 に代入
add	\$t1,	\$a0,	\$t1	# v + (k * 4)をレジスタ \$t1 に代入
				# レジスタ \$t1 は v[k] のアドレスを表す
lw	\$t0,	0(\$t1)		# レジスタ \$t0 (temp) = v[k]
lw	\$t2,	4(\$t1)		# レジスタ \$t2 = v[k + 1]
				# v の次の要素を読み出し
sw	\$t2,	0(\$t1)		# v[k] = レジスタ \$t2
sw	\$t0,	4(\$t1)		# v[k+1] = レジスタ \$t0 (temp)

戻り

jr	\$ra	# 呼出し元のルーチンに戻る
----	------	----------------

演習問題 その⑤

- Convert the following sort function to an assembly code.
- Register allocation: arguments v and n use \$a0 and \$a1, i and j use \$s0 and \$s1

```
void sort (int v[], int n)
{
    int i, j;
    for (i=0; i<n; i+=1) {
        for (j=i-1; j>=0 && v[j] > v[j+1]; j-=1) {
            swap (v, j);
        }
    }
}
```

演習問題その⑤の関数sortに関して

これは、何ソートと呼ばれるか？

$v[0] = 104$, $v[1] = 102$, $v[2] = 30$, $v[3] = 64$ の時の、前ページの動作を説明せよ

V[0]	V[1]	V[2]	V[3]
104	102	30	64

iとjの値の変化と共に配列vの変化がどのように変わるかを確認してください。

102	104	30	64
-----	-----	----	----

102	30	104	64
-----	----	-----	----

30	102	104	64
----	-----	-----	----

30	102	64	104
----	-----	----	-----

30	64	102	104
----	----	-----	-----

```
void sort (int v[], int n)
{
    int i, j;
    for (i=0; i<n; i+=1) {
        for (j=i-1; j>=0 && v[j] > v[j+1]; j-=1) {
            swap (v, j);
        }
    }
}
```

\$a0	v
\$a1	n
\$s0	i
\$s1	j

外側のループ

move \$s0, \$zero = add \$s0, \$zero, \$zero

外側のループは以下の通り(疑似命令を使っている)

```

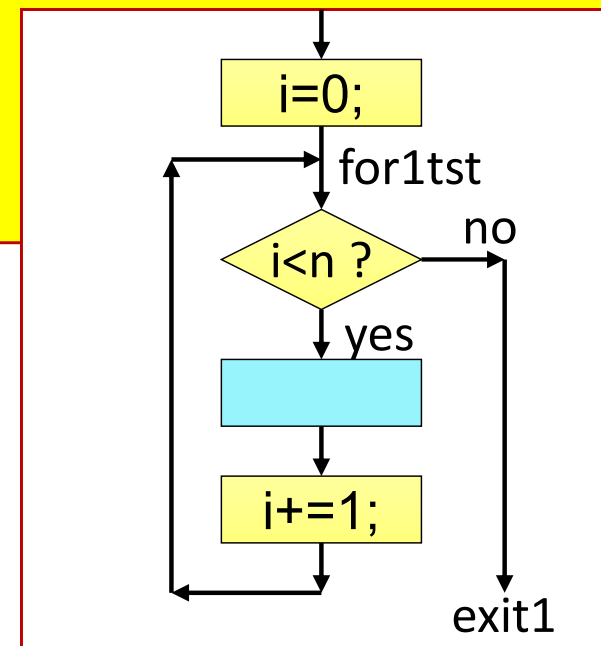
        move    $s0, $zero           # ループの初期化 (i=0)
for1tst: slt     $t0, $s0, $a1        # $t0=0 if $s0 >= $a1 (i>=n)
        beq     $t0, $zero, exit1    # go to exit1 if $s0 >= $a1 (i>=n)
        ... body of the loop ...
        addi    $s0, $s0, 1          # i=i+1
        j       for1tst
exit1:

```

```

for (i=0; i<n; i+=1) {
    for (j=i-1; j>=0 && v[j] > v[j+1]; j-=1) {
        swap (v,j);
    }
}

```



\$a0	v
\$a1	n
\$s0	i
\$s1	j

内側のループ

```

addi    $s1, $s0, -1           # ループの初期化  j=i-1
for2tst: slti    $t0, $s1, 0     # $t0=1 if $s1<0 (j<0)
        bne     $t0, $zero, exit2 # go to exit2 if $s1<0 (j<0)
        sll     $t1, $s1, 2
        add     $t2, $a0, $t1
        lw      $t3, 0($t2)      # $t3=v[j]
        lw      $t4, 4($t2)      # $t4=v[j+1]
        slt     $t0, $t4, $t3    # $t0=0 if $t4 >= $t3
        beq     $t0, $zero, exit2 # go to exit2 if $t4>=$t3
        ... body of the loop ...
        addi    $s1, $s1, -1
        j       for2tst
exit2:

```

```

for (i=0; i<n; i+=1) {
    for (j=i-1; j>=0 && v[j] > v[j+1]; j-=1) {
        swap (v,j);
    }
}

```


レジスタの退避と復元

- swapを\$a0と\$a1を引数として何度も呼ぶ
 - \$a0と\$a1の退避の必要あり
 - swapの最初に引数\$a0と\$a1を\$s2と\$s3にコピーして使うことにする
 - 前の2つのスライドのsortの中で使われている\$a0と\$a1を\$s2と\$s3に変更
- swapを呼ぶために\$raの値が変わるので, sortの最初に\$ra, \$s0-\$s3のスタックへの退避, 最後にスタックから復元する

以上を考慮して, レジスタの退避・復元・コピーと, 内側と外側のループを併合すると, 全体の回答は次の2ページとなる.

(教科書p136)

全体のコードの解答

一部、穴埋めが試験に出ても
できるだけ理解しておくこと

```
sort: addi    $sp, $sp, -20
      sw      $ra, 16($sp)
      sw      $s3, 12($sp)
      sw      $s2, 8($sp)
      sw      $s1, 4($sp)
      sw      $s0, 0($sp)
      move    $s2, $a0
      move    $s3, $a1
```

レジスタの退避

引数のコピー(退避)

外側のループの前半

(但し、P.23 の \$a1 は \$s3 に置き換える)

内側のループの前半

(但し、P.24 の \$a0 は \$s2 に置き換える)

```
move    $a0, $s2
move    $a1, $s1
jal      swap
```

引数の引き渡しと呼び出し

内側のループの後半

外側のループの後半

(次頁に続く)

全体のコードの続き

```
exit1:  lw    $s0, 0($sp)
        lw    $s1, 4($sp)
        lw    $s2, 8($sp)
        lw    $s3, 12($sp)
        lw    $ra, 16($sp)
        addi  $sp, $sp, 20
        jr    $ra
```

レジスタの復元

9 lines of C code → 35 lines of assembly

全体のコード (教科書P.136)

レジスタの退避				
sort:	addi	\$sp, \$sp, -20	#	スタック上にレジスタ 5 つ分の領域を確保
	sw	\$ra, 16 (\$sp)	#	スタック上に \$ra を退避
	sw	\$s3, 12 (\$sp)	#	スタック上に \$s3 を退避
	sw	\$s2, 8 (\$sp)	#	スタック上に \$s2 を退避
	sw	\$s1, 4 (\$sp)	#	スタック上に \$s1 を退避
	sw	\$s0, 0 (\$sp)	#	スタック上に \$s0 を退避
手続き本体				
パラメータの コピー	move	\$s2, \$a0	#	\$a0 を \$s2 にコピー (\$a0 を退避)
	move	\$s3, \$a1	#	\$a1 を \$s3 にコピー (\$a1 を退避)
外側のループ	move	\$s0, \$zero	#	i = 0
	for1tst:	slt	\$t0, \$s0, \$s3	# \$s0 ≥ \$s3 (i ≥ n) なら \$t0 = 0
	beq	\$t0, \$zero, exit1	#	\$s0 ≥ \$s3 (i ≥ n) なら exit1 へ
内側のループ	addi	\$s1, \$s0, -1	#	j = i - 1
	for2tst:	slti	\$t0, \$s1, 0	# \$s1 < 0 (j < 0) なら \$t0 = 1
	bne	\$t0, \$zero, exit2	#	\$s1 < 0 (j < 0) なら exit2 へ
	sll	\$t1, \$s1, 2	#	\$t1 = j * 4
	add	\$t2, \$s2, \$t1	#	\$t2 = v + (j * 4)
	lw	\$t3, 0(\$t2)	#	\$t3 = v[j]
	lw	\$t4, 4(\$t2)	#	\$t4 = v[j+1]
	slt	\$t0, \$t4, \$t3	#	\$t4 ≥ \$t3 なら \$t0 = 0
	beq	\$t0, \$zero, exit2	#	\$t4 ≥ \$t3 なら exit2 へ
パラメータの引渡し と呼出し	move	\$a0, \$s2	#	swap 用の第 1 パラメータ v を渡す
	move	\$a1, \$s1	#	swap 用の第 2 パラメータ j を渡す
	jal	swap	#	図 2.25 参照
内側のループ	addi	\$s1, \$s1, -1	#	j -= 1
	j	for2tst	#	内側のループの条件判定に戻る
外側のループ	exit2:	addi	\$s0, \$s0, 1	# i += 1
	j	for1tst	#	外側のループの頭に戻る
レジスタの復元				
exit1:	lw	\$s0, 0 (\$sp)	#	スタックから \$s0 を復元
	lw	\$s1, 4 (\$sp)	#	スタックから \$s1 を復元
	lw	\$s2, 8 (\$sp)	#	スタックから \$s2 を復元
	lw	\$s3, 12 (\$sp)	#	スタックから \$s3 を復元
	lw	\$ra, 16 (\$sp)	#	スタックから \$ra を復元
	addi	\$sp, \$sp, 20	#	スタック・ポインタを復元
呼出し元への戻り				
	j	\$ra	#	呼出し元のルーチンへ戻る

図 2.27 図 2.26 の sort 手続きの MIPS アセンブリ・コード.

Lec. 7での要チェック用語集

アセンブラ
リンカ
ローダ
標準ライブラリ
テキスト・セグメント
静的データ
動的データ
リロケーション情報
シンボル表
ヒープ
オブジェクトファイル
ライブラリファイル
DLL