

# アセンブリ言語

ABI, バイナリ形式, リンク

情報工学系  
権藤克彦



# ABI と API

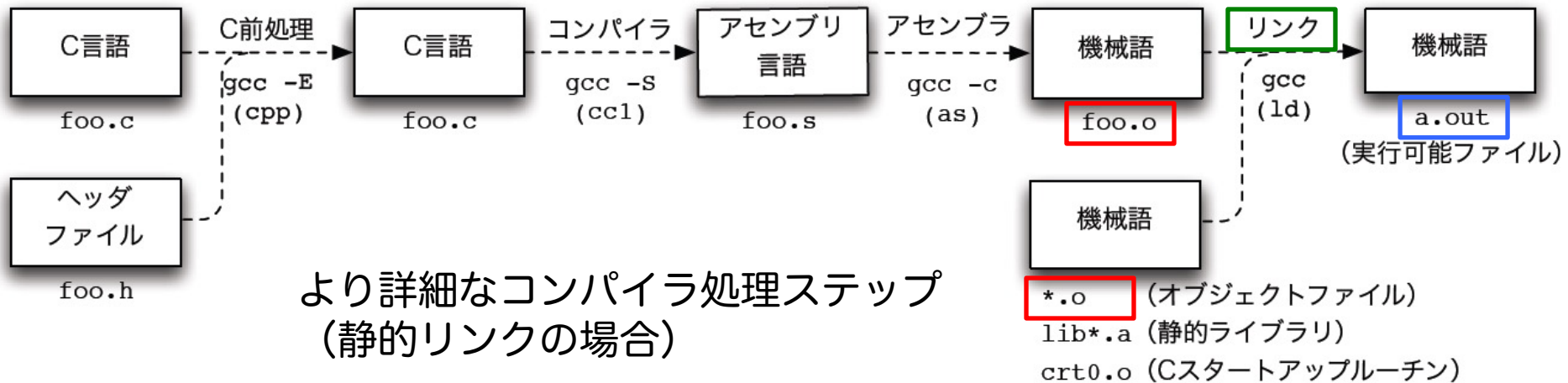
- **ABI** = application binary interface
  - バイナリコードのためのインタフェース規格.
  - 同じABIをサポートするシステム上では再コンパイル無しで同じバイナリコードを実行できる.
  - コーリングコンベンション, バイトオーダ, アラインメント, バイナリ形式などを定める.
- **API** = application programming interface
  - ソースコードのためのインタフェース規格.
  - 同じAPIをサポートするシステム上では再コンパイルすれば同じソースコードを実行できる.
  - 例: POSIX や SUSv3 は UNIX の API . システムコール, ライブラリ関数, マクロなどの形式や意味を定めている.

UNIXシステムコールは1Qの  
システムプログラミングで学ぶ.



# オブジェクトファイル, 実行可能ファイル

- オブジェクトファイル = 拡張子が `.o` のファイル.
- 実行可能ファイル = `a.out`
  - Windowsでは拡張子が `.exe` のファイル.
- 複数のオブジェクトファイルを1つの実行可能ファイルに合体させることをリンクという.





# ライブラリ関数とシステムコール（１）

- ライブラリ関数(library function)
  - 例：printf, fopen, malloc
  - （一般に）高レベル.
  - 自分で定義した関数と同様に呼び出せる.
  - （知識があれば）ライブラリ関数を自分でプログラムできる.
- システムコール(system call)
  - 例：write, open, mmap
  - OS（カーネル）が提供するサービスの関数インタフェース.
  - 自分で定義した関数と同様に呼び出せる.
    - ・ ライブラリ関数を経由せず，直接，システムコールを呼び出せる.
  - ライブラリ関数としてはプログラムできない.
    - ・ システムコールの実行にカーネルの特権（CPUの特権モード）が必要なので，システムコールはユーザ空間では実行できない.

printfは内部で  
writeを呼び出す.



# ライブラリ関数とシステムコール (2)

a.out

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

プログラム

```
int printf ()  
{  
    write (...);  
}
```

ライブラリ関数

```
int write ()  
{  
    ....  
}
```

システムコール  
の実体

カーネル

コール命令  
(call)

トラップ命令  
(int, syscall)

call命令の場合は  
ラップ関数の呼び出し

ユーザ空間  
カーネル空間



# ライブラリ関数とシステムコール（3）

	ライブラリ関数	システムコール
例	printf, fopen, malloc	write, open, mmap
実体の場所	ファイル（例： /usr/lib/libc.a）	OS（カーネル）内部
呼び出す方法	コール命令（例：call）	トラップ命令（例：int）
マニュアル(man)の章	（通常）3章	（通常）2章
呼び出しオーバーヘッド	小さい	大きい
実行時のメモリ空間	ユーザ空間	カーネル空間
実行時間の呼称	ユーザ時間	システム時間

トラップ命令は割り込みの一種.



# バイナリ形式 (binary format)

- オブジェクトファイルや実行可能ファイルの形式.
- いろいろある.

バイナリ形式	説明
Mach-O	Mac OS X用
a.out (assembler output)	古いUNIX用
ELF (executable and linking format)	Linuxなど用
PE (portable executable)	Windows用

- GNU Binutils中のツール（例：objdump）
  - 異なるバイナリ形式に対して使えて便利.
    - cf. readelfコマンドはELF形式のバイナリファイルしか扱えない.
  - ただし、異なるバイナリ形式への変換はできない.





# バイナリファイルの種類

- オブジェクトファイル (\*.o)
- 実行可能ファイル (a.out)
- ライブラリファイル
  - 静的ライブラリ (lib\*.a)
  - 動的ライブラリ（共有ライブラリ） (\*.so, \*.dll, \*.sa, ...)

この授業では  
動的ライブラリは  
扱いません。





# オブジェクトファイル(\*.o)

- 再配置可能オブジェクトファイル(relocatable object file)
- セクションからなるバイナリファイル
- 特徴
  - 外部シンボルの参照が未解決.
  - 再配置が可能.



# セクション

- セクションはバイナリコードの分割単位.
- 主なセクション 名前が異なる場合もあるが、役割は大きくこの4つ.
  - .text, .data, .rdata, .bss の4つ.
  - 他には記号表や再配置情報などのセクションがある.
- ヘッダ
  - 「どんなセクションがあるか」という情報を保持.

セクション {

ヘッダ
.text
.data
.rdata
.bss
記号表

機械語コード

初期化済みの静的変数

読み出し専用データ (例: "hello¥n")

未初期化の静的変数

ELF形式では、いくつかのセクションがまとまってセグメントという単位を構成.



# セクション：objdump -h (1)

読み方は  
次スライド参照.

- objdump -h はセクション情報を表示.

```
% objdump -h foo.o
foo.o:  file format mach-o-x86-64
Sections:
Idx Name          Size      VMA           LMA             File off  Algn
 0 .text          00000033  0000000000000000  0000000000000000  00000270  2**4
    CONTENTS, ALLOC, LOAD, RELOC, CODE
 1 .data          00000004  0000000000000034  0000000000000034  000002a4  2**2
    CONTENTS, ALLOC, LOAD, DATA
 2 .cstring       00000008  0000000000000038  0000000000000038  000002a8  2**0
    CONTENTS, ALLOC, LOAD, READONLY, DATA
 3 __LD.__compact_unwind 00000020  0000000000000040  0000000000000040  000002b0  2**3
    CONTENTS, RELOC, DEBUGGING
 4 .eh_frame      00000040  0000000000000060  0000000000000060  000002d0  2**3
    CONTENTS, ALLOC, LOAD, READONLY, DATA
```

.rdata は、macOSでは .cstring というセクション名に.

記号表の中身を見るには nm コマンドや objdump --symsを使う.  
文字列を見るには string を使う.



## セクション：objdump -h (2)

- セクション = 中身 + 様々な属性値.

項目	説明
Size	セクションのバイト数
VMA	Virtual Memory Addressの略. 実行時の先頭メモリアドレス. 再配置に使用.
LMA	Load Memory Addressの略. ロード時の先頭メモリアドレス. 通常はVMAと一致.
File off	ファイルオフセット. ファイル先頭からのバイト数.
Align	アラインメント制約. 例えば, $2^{**}2$ は $2^2$ を意味し, 4バイト境界に要配置.

セクションフラグ	説明
CONTENTS	このセクションには内容がある.
ALLOC	ロード時にメモリ割り当てが必要.
LOAD	ロード時にこのセクションをファイルからロードする必要がある.
RELOC	このセクションは再配置が必要である. 再配置情報を含む.
READONLY	このセクション中のデータは読み出し専用である.
CODE	このセクションは機械語コードを含む.
DATA	このセクションはデータ (静的変数) を含む.



# 外部シンボルの解決

```
% gcc -c main.c
% gcc -c sub.c
% gcc main.o sub.o
% ./a.out
999
%
```

←リンク

- 外部シンボル (external symbol)
  - 自分のファイル中で未定義の変数や関数. 例: 以下の変数x.
  - 記号ともいう (特に「記号表」(symbol table)という文脈で)
- 外部シンボルの解決
  - ファイルをまたいで, 変数や関数の対応関係を調べて, 変数名や関数名の参照にメモリアドレスを割り当てること.
  - つまり, 未定義の変数や関数を無くす.

main.c

```
#include <stdio.h>
extern int x;
int main (void)
{
    printf ("%d\n", x);
}
```

sub.c

```
int x = 999;
```

対応付けて, 未定義な状態を解決.



# 外部シンボルの解決

main.c

```
#include <stdio.h>
extern int x;
int main (void)
{
    printf ("%d¥n", x);
}
```

```
% gcc -c main.c
% nm main.o
00000000 T _main
                U _printf
                U _x
%
```

外部シンボル  
\_printf と \_x が  
未解決（未定義）

sub.c

```
int x = 999;
```

```
% gcc main.c sub.c
% nm a.out
00001fca T _main
                U _printf
00002014 D _x
%
```

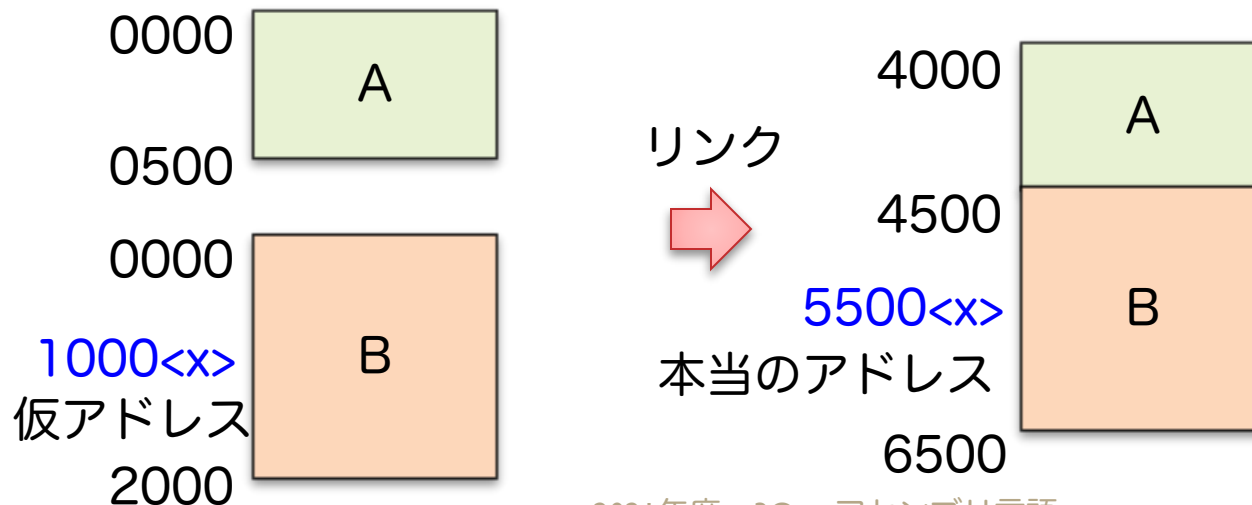
sub.oとリンクして  
\_xは解決された  
(実体と結びついた)。  
\_printf は未解決。

現在, macOS では -static オプションで  
静的リンクした a.out を生成できない。  
動的リンクの場合, \_printf の解決  
(つまりリンク) は実行時にされる。



# 再配置

- 再配置 (relocation)
  - 仮アドレスを本当のアドレスに修正すること.
- なぜ仮アドレス？
  - リンク時に変数や関数のアドレスは変化するから.
  - オブジェクトファイルでは仮アドレスとし，後で修正可能なようにしてある．（再配置情報として記憶しておく）





```
static int x = 0xAABBCCDD;
int main ()
{
    return x;
}
```

```
% gcc -static -c reloc.c
```

```
% objdump -Dr reloc.o
```

```
Disassembly of section .text:
```

```
0000000000000000 <_main>:
```

```
0: 55                push    %rbp
1: 48 89 e5          mov     %rsp,%rbp
4: c7 45 fc 00 00 00 00 movl    $0x0,-0x4(%rbp)
b: 8b 05 00 00 00 00 mov     0x0(%rip),%eax
```

再配置前(\*.o)

**d: DISP32 \_x**

再配置情報

仮アドレス

```
11: 5d                pop     %rbp
12: c3                retq
```

```
Disassembly of section .data:
```


```
0000000000000014 <_x>:
```

```
14: dd cc                (bad)
```

実行時ではないと決まらないアドレスもある (例: printf)  
→ そのアドレスを入れる場所を事前に決める

## 再配置後(a.out)

```
% gcc reloc.c
% objdump -Dr a.out
Disassembly of section .text:
00000000100000fa0 <_main>:
  100000fa0: 55                push    %rbp
  100000fa1: 48 89 e5          mov     %rsp,%rbp
  100000fa4: c7 45 fc 00 00 00 00 movl    $0x0,-0x4(%rbp)
  100000fab: 8b 05 4f 00 00 00  mov     0x4f(%rip),%eax
  100000fb1: 5d                pop     %rbp
  100000fb2: c3                retq
```



本当の(相対)アドレス

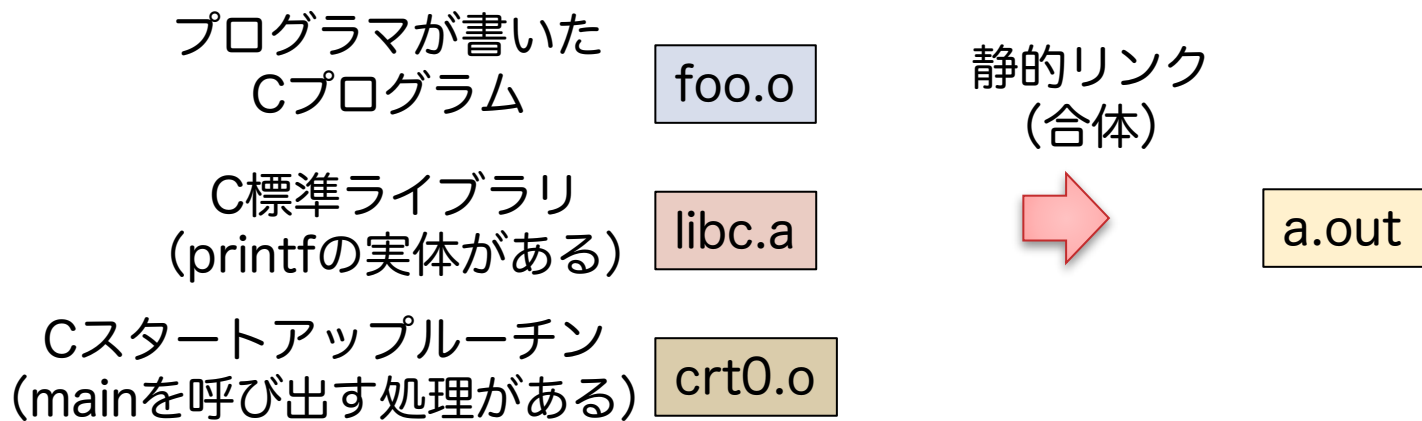
```
Disassembly of section .data:
00000000100001000 <_x>:
  100001000: dd cc            (bad)
```

$$0x4F + 0xFB1 = 0x1000$$



# リンク (1)

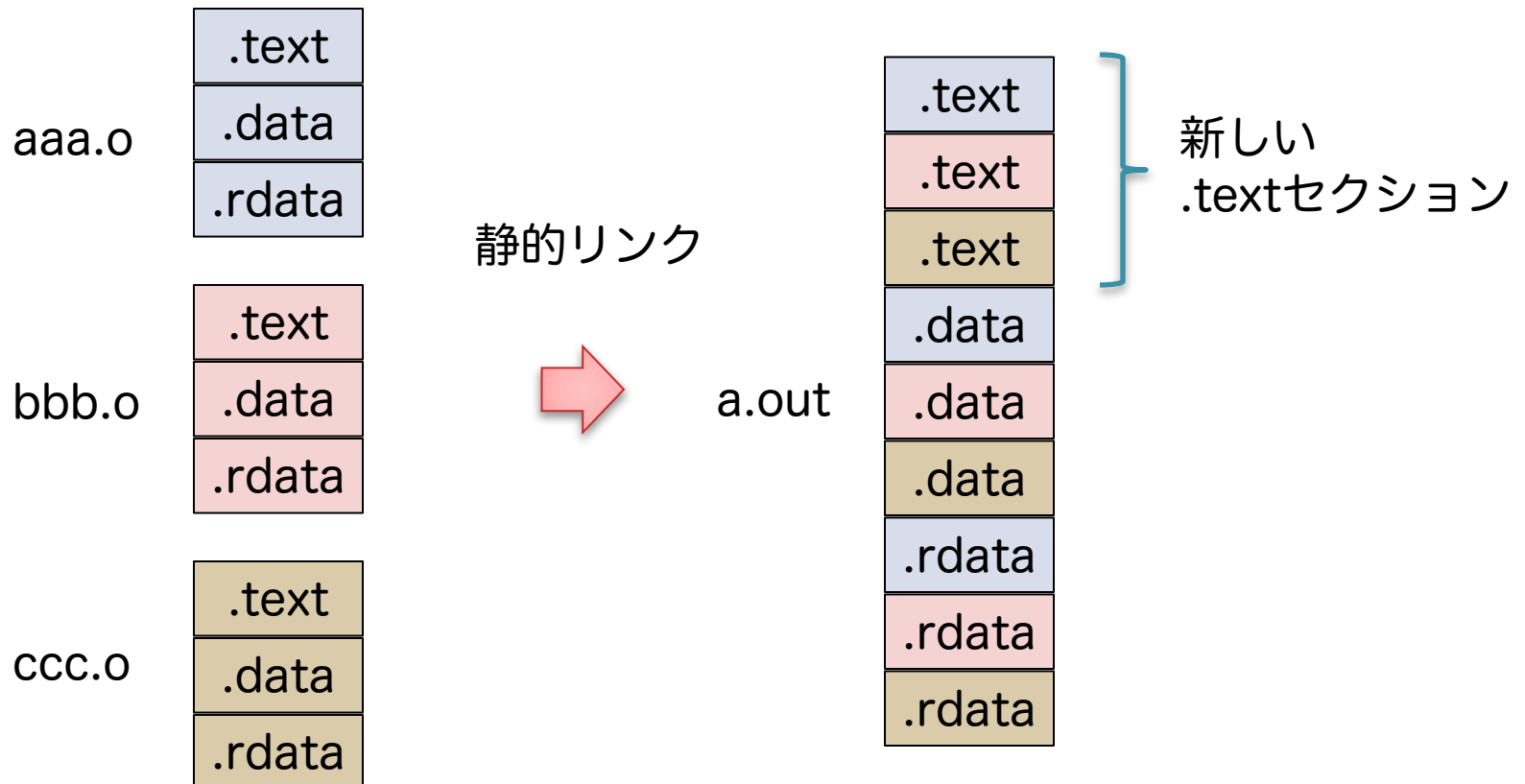
- リンク(link)
  - オブジェクトファイル(\*.o)やライブラリファイルを結合して、1つの実行可能ファイル(a.out)にすること.
  - リンカ (コンパイラの一部, ld) が実行処理.
- アドレスの調整が必要.
  - 外部シンボルの解決や再配置などを行って調整する.





## リンク (2)

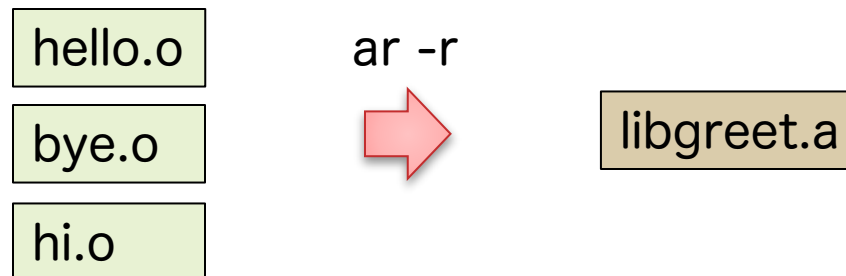
- リンクはセクションごとにマージする。
  - 当然, アドレス解決と再配置を行いながら.





# 静的ライブラリとアーカイブ (1)

- ライブラリ
  - オブジェクトファイル(\*.o)をアーカイブ形式で1つにまとめたファイル.
  - 静的ライブラリは \*.a. libc.a はC標準ライブラリ.
- アーカイブ処理
  - **arコマンド**で, 様々なアーカイブ処理を行う.





## 静的ライブラリとアーカイブ（２）

main.c

```
extern void hello (void);
extern void bye (void);
int main (void)
{
    hello (); bye ();
}
```

hello.c

```
#include <stdio.h>
void hello (void)
{    printf ("hello¥n"); }
```

bye.c

```
#include <stdio.h>
void bye (void)
{    printf ("bye¥n"); }
```

```
% gcc -c hello.c
% gcc -c bye.c
% ar -r libgreet.a hello.o bye.o
ar: creating archive libgreet.a
% ar -t libgreet.a
___.SYMDEF
hello.o
bye.o
% gcc main.c -lgreet -L.
% ./a.out
hello
bye
%
```

自分で作ったライブラリを  
呼び出せた！

-lオプションはリンクするライブラリ名を指定する。-lnameで libname.a とリンクする。  
-Lオプションはライブラリが存在するディレクトリを指定する。



# macOS のABIを少しだけ紹介

- Introduction to Mac OS X ABI Function Call Guide (英語)
- Mac OS X ABI Mach-O File Format Reference (英語)
- どちらも正式な URL を発見できず. 上は IA-32のみで, x86-64を含まず.





# 大きな構造体を関数から返す方法（１）

1つ目のマニュアルから転載

## Listing 4: Using a large structure --- source code

```
typedef struct {  
    float ary[8];  
} big_struct;  
big_struct callee(int a, float b) {  
    big_struct callee_struct;  
    ...  
    return callee_struct;  
}  
caller () {  
    big_struct caller_struct;  
    caller_struct = callee(3, 42.0);  
}
```

問題：大きな構造体はレジスタに入りきらない。  
どうやって、引数や戻り値として渡せばいいか？  
スタック上に大きなサイズの引数を積む？



# 大きな構造体を関数から返す方法（２）

1つ目のマニュアルから転載

Listing 5: Using a large structure --- compiler interpretation

```
typedef struct {  
    float ary[8];  
} big_struct;  
void callee(big_struct *p, int a, float b)  
{  
    big_struct callee_struct;  
    ...  
    *p = callee_struct;  
    return;  
}  
caller() {  
    big_struct caller_struct;  
    callee(&caller_struct, 3, 42.0);  
}
```

答え：隠し引数としてポインタを渡す（ようなアセンブリコードを生成する）。