CS2141 – Software Development using C/C++

Libraries

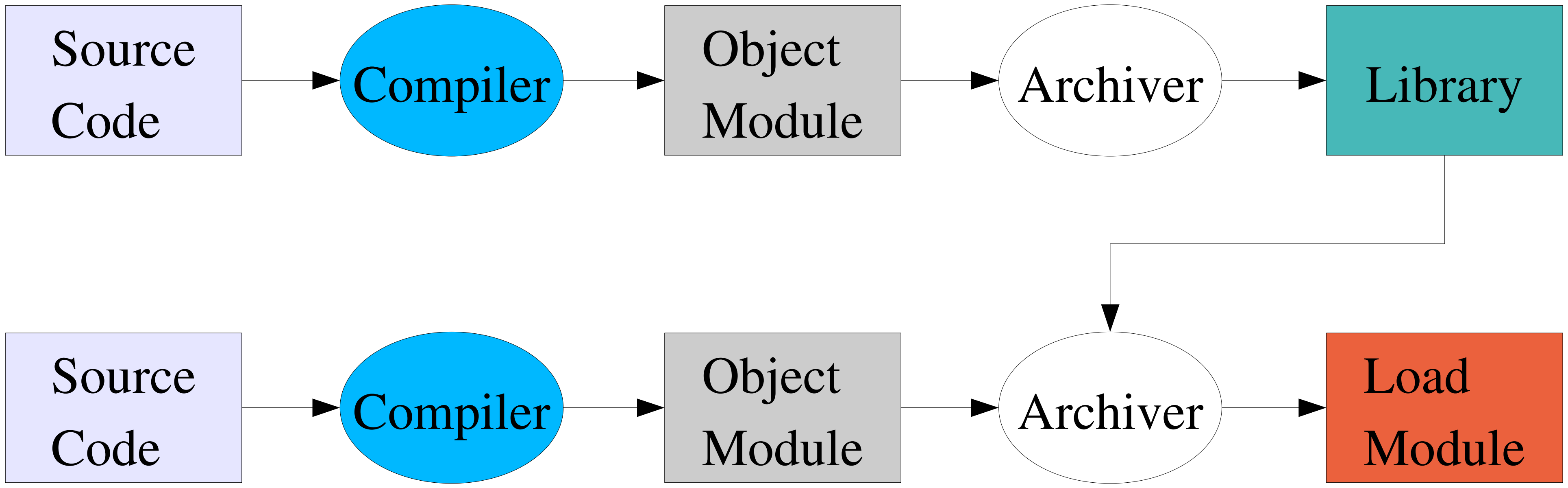
Compilation and linking

```
/* p1.c */
int x;
int z;
main()
{
   x=0; z=0;
   printf("f(3)=%d x=%d z=%d\n",f(3),x,z);
}
```

- Code for `int f(int)` not available yet, nor `printf()`
- `x` and `z` available to other object modules
- Compiled module must reflect these facts

Compilation sequence

- Compiler: Converts program from source file to machine language, produces an *object module* (which cannot be executed)
- Linker: Produces a *load module* which is ready to be executed
- Operating system will create a *process* from the load module



Symbol table

- Object file may contain unresolved global *symbols*
  - Defined: variables, functions defined within object file, can be referenced within other object files
  - Undefined: variables, functions used within this object file, defined elsewhere
- Linker combines object files and resolves symbols while creating executable
  - Object file contains *symbol table*
  - Symbol table will contain information needed to resolve symbols
  - Linker uses information from the symbol table
- Executable will contain no unresolved symbols

Symbol table cont.

- `nm` can be used to display symbol table
  - Uppercase is used for global symbols
  - Lowercase is used for local symbols
  - T code section
  - U is undefined
  - Look at `man nm` for other symbol types

Object Modules

- Many different formats (a.out, ELF, COFF, etc.)
- **Header Section** - Sizes required to parse object module and create program
- **Machine Code**- Generated machine code (also called text)
- **Initialized Data** - Initialized global and static data (doesn't go on stack)
- **Symbol Table** - External symbols
    - Undefined - Used in this module, defined elsewhere
    - Defined - Defined in this module, may be undefined in another module
- **Relocation Information** - Record of places where symbols must be relocated

```
#include  <iostream.h>
#include  <math.h>

float  arr[100];
int    size=100;

void  main(  int argc,  char *argv[ ] )  {

    int  i;
    float  sum = 0;

    for  ( i = 0;  i <  size;  ++i )  {

        cin  >>  arr[i];
        arr[i] = sqrt( arr[i] );
        sum  +=  arr[i];

    }

    cout << sum;

}
```

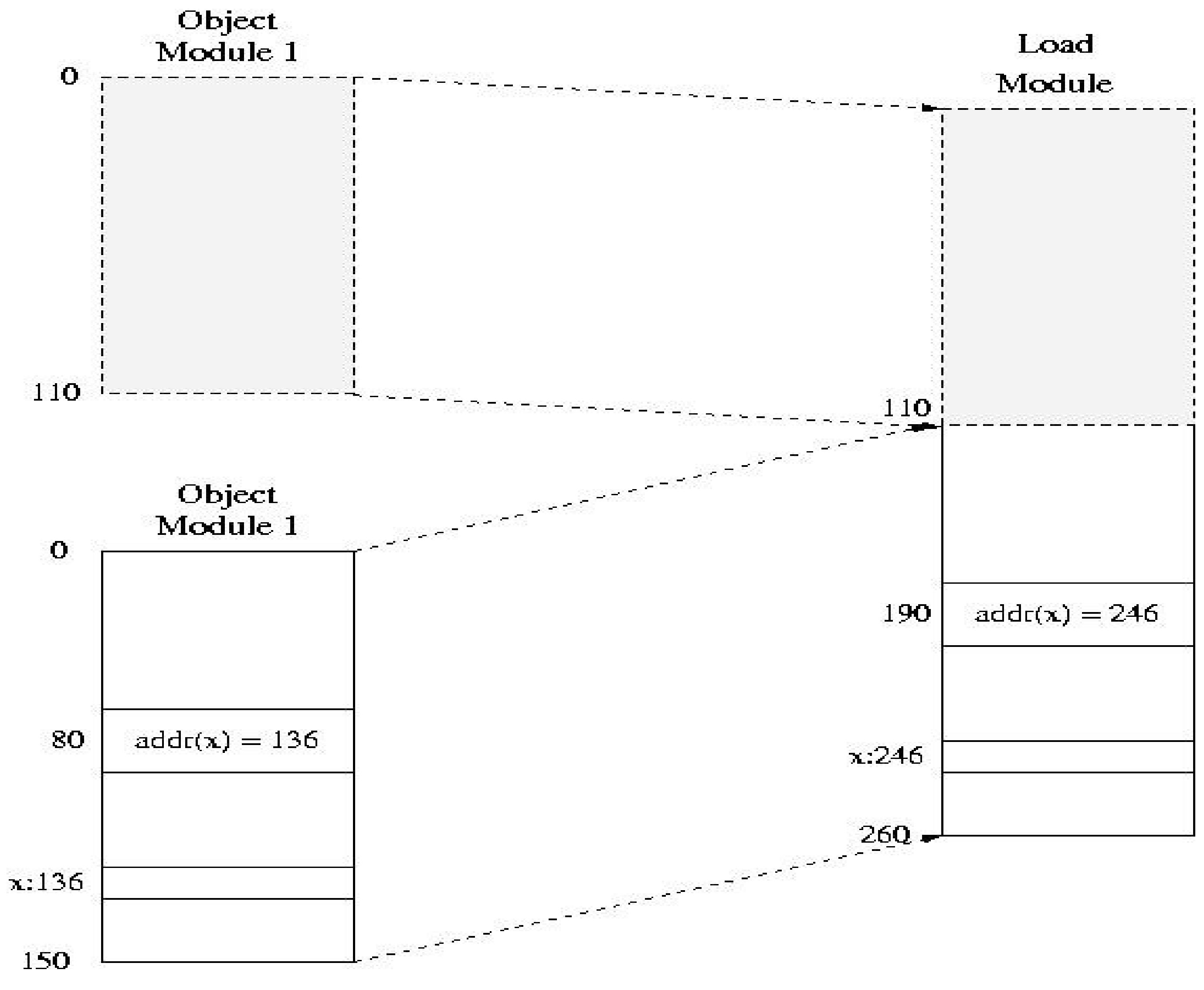
Offset	Contents	Comment
		Header Section
0	94	Number of bytes of machine code
4	4	Number of bytes of initialized data
8	400	Number of bytes of uninitialized data
12	72	Number of bytes of symbol table
16	?	Number of bytes of relocation information
		Machine Code Section (text)
20	XXXX	Code for top of `for` loop
50	XXXX	Code for `arr[i] << cin`
66	XXXX	Code for `arr[i]=sqrt(arr[i])`
86	XXXX	Code for `sum += arr[i]`
98	XXXX	Code for bottom of `for` loop
102	XXXX	Code for `cout << sum`
		Initialized Data Section
114	100	Location of `size`
		Symbol Table Section
118	?	"`size`" = 0 (in data section)
130	?	"`arr`" = 4 (in data section)
142	?	"`main`" = 0 (in code section)
154	?	"`>>`" = external , used at 42
166	?	"`sqrt`" = external, used at 62
178	?	"`<<`" = external, used at 90
		Relocation Information Section
190	?	Relocation Information

Libraries

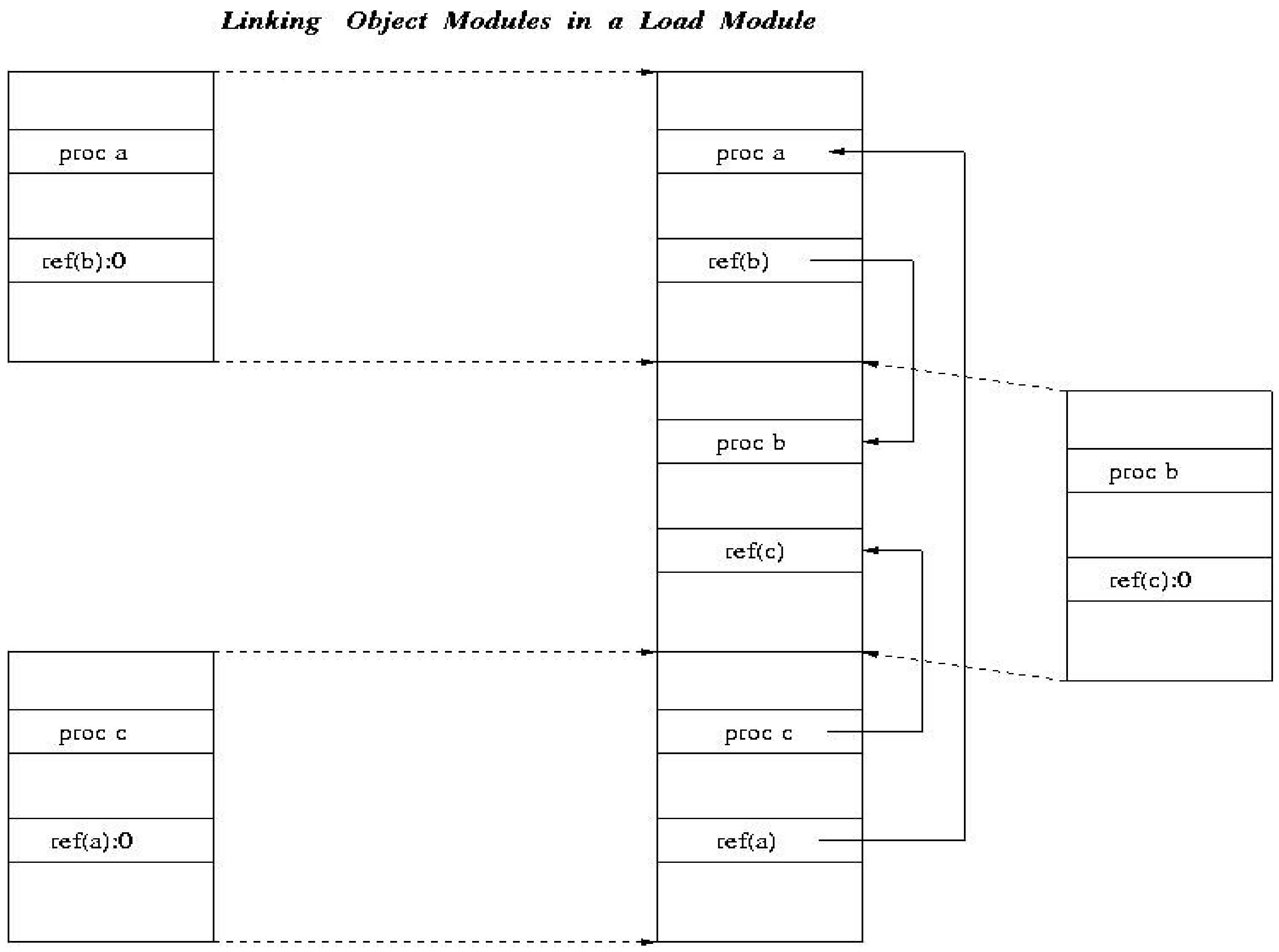
Linking

- Object module will (usually) assume starting address is zero
- Linker combines several object modules
  - Text sections combined, data sections combined, ...
- Combined modules cannot all start at zero
- Cannot have unresolved references in load module
- Two tasks then:
  - Relocate modules (account for starting address that results from combining modules)
  - Link modules (resolve undefined external references)

Relocation


Object Module 1
0
110
Object Module 1
0
80
addr(x) = 136
x:136
150
Load Module
110
190
addr(x) = 246
x:246
260

Resolve all symbols



Create a Load Module

1) Create empty load module and global symbol table

2) Get next object module or library name

3) Object module:

- Insert code and data, remember where
- Undefined external references:
  - Already defined in global symbol table, write value in just loaded object module
  - Not yet defined, note that links must be fixed when symbol defined
- Defined external references:
  - Fix up all previous references (to this symbol) noted in global symbol table

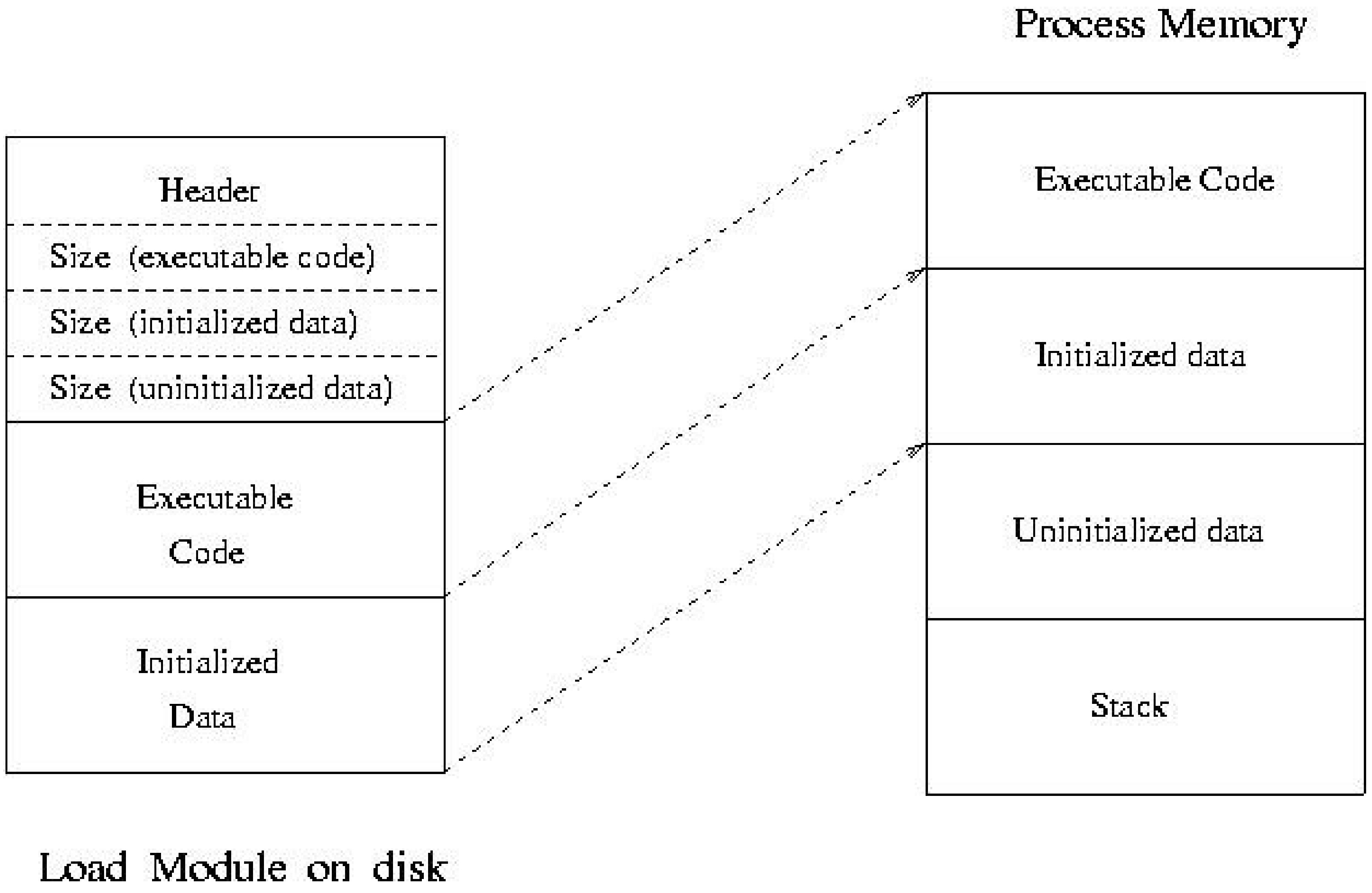
Create Load Module cont.

4) Library:

- Find each undefined external reference in global symbol table
- See if symbol defined in library
- If so, load it per step (3)

5) Back to step 2

Process Creation



Static Linking

- Library routines combined into binary program image
- Creates large load modules
- Same library may be contained in multiple images throughout file system
- Once load module is created, it is impervious to changes in referenced library
  - New versions require recompilation
  - Does not depend on existence of (specific version of) library on system
- gcc -static ...

Dynamic Linking

- *Stub* included in binary program image for each library-routine reference
- Stub is code to locate memory-resident routine or load it if library routine not present
- Stub replaces itself with address of routine and executes routine
- Will use most recent version of library routine
- Higher overhead during use; faster startup than statically linked
- Allows same code to be shared by multiple processes

Static libraries

- Static libraries created with `ar`. (See manual page.) Commonly used options:
  - `c` create a new library
  - `q` add the named file to the end of the archive
  - `r` replace a named archive/library member
  - `t` print a table of archive contents
- `ranlib` run on library to create index of each symbol defined by a relocatable library

Example: Working with static libraries

```
gcc -c libFunc.c

ar -cq libMyLib.a libFunc.o

ranlib libMyLib.a

gcc myProg -lmyLib

gcc -o myProg myProg.c -L. -lMyLib

ar -q libMyLib.a anotherFunc.o

ranlib libMyLib.a
```

Example: Working with dynamic libraries

- Must compile *position independent code*;
  - `gcc -fPIC -c myFunc.c`
- Use `ld` to create library;
  - `gcc -shared *.o -o libmyUtil.so` (ld via gcc)

- `ldd` returns shared libraries used by an object module

Useful tools

- file – gives information about file (executable, relocatable...)
- nm – list symbols from object file or library