CS536
Prog. Language, Memory, Mutifile and AST

A Sahu

CSE, IIT Guwahati

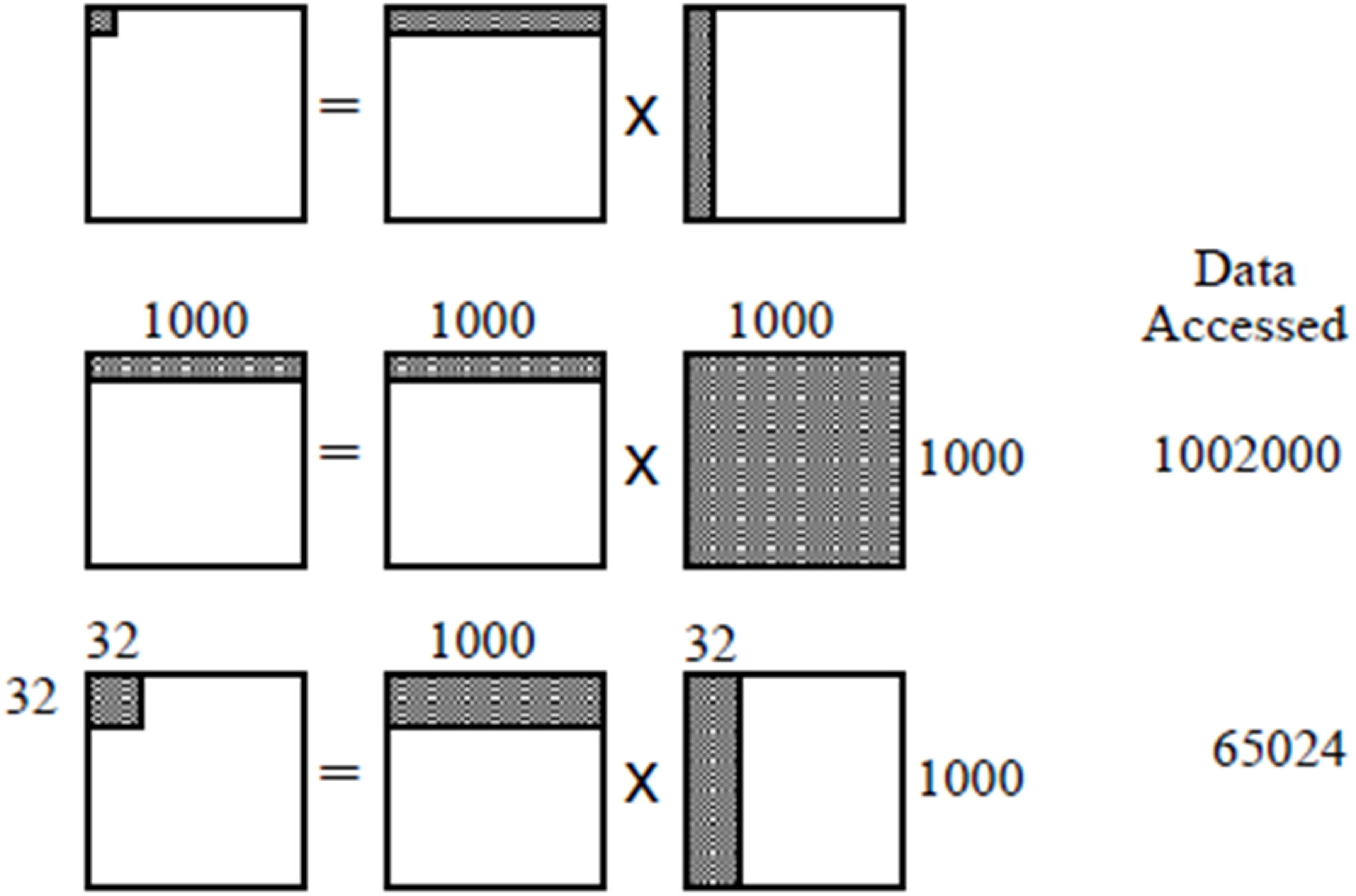
http://jatinga.iitg.ac.in/~asahu/cs536/

Outline

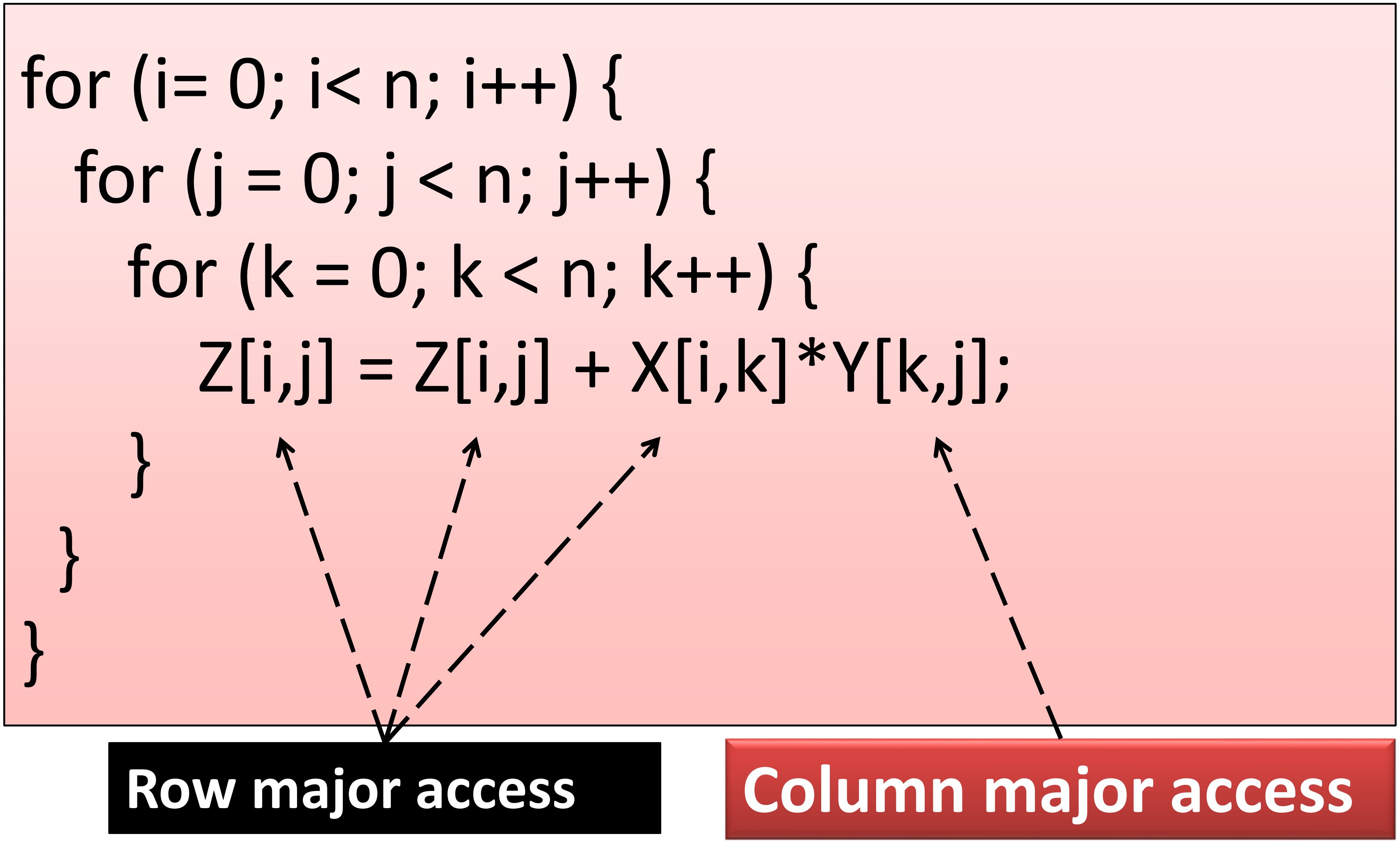
- Last Class : Blocking in Matrix Multiplication
- Programming Language Basic
- Memory Layout of a C Program
- Compiling Multiple C Files
- Basic of **Syntax Directed Translation**
- Intermediate Representation

http://jatinga.iitg.ac.in/~asahu/cs536/

Blocking for Matrix Multiplication



Blocking for Matmul: Original Code



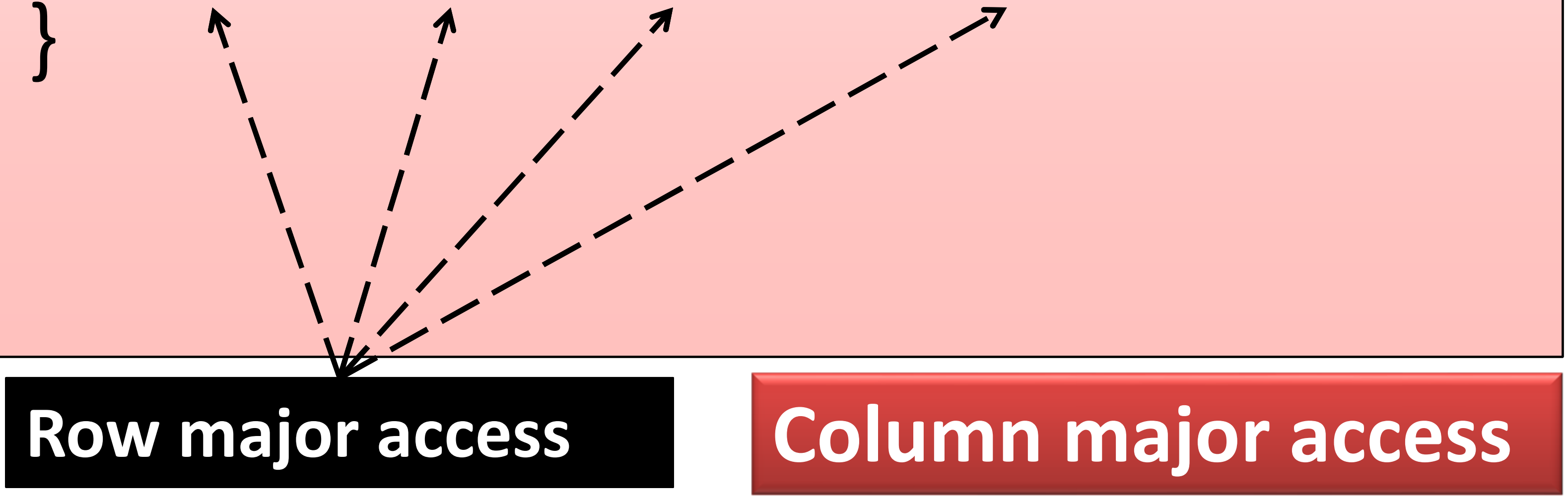
```
for (i= 0; i< n; i++) {
   for (j = 0; j < n; j++) {
      for (k = 0; k < n; k++) {
          Z[i,j] = Z[i,j] + X[i,k]*Y[k,j];
      }
   }
}
```

Row major access

Column major access

Blocking for Matmul: Original Code

```
for (i= 0; i< n; i++) {
   for (k = 0; k < n; k++) {
      for (j = 0; j < n; j++) {
         Z[i,j] = Z[i,j] + X[i,k]*Y[k,j];
      }
   }
}
```



Row major access

Column major access

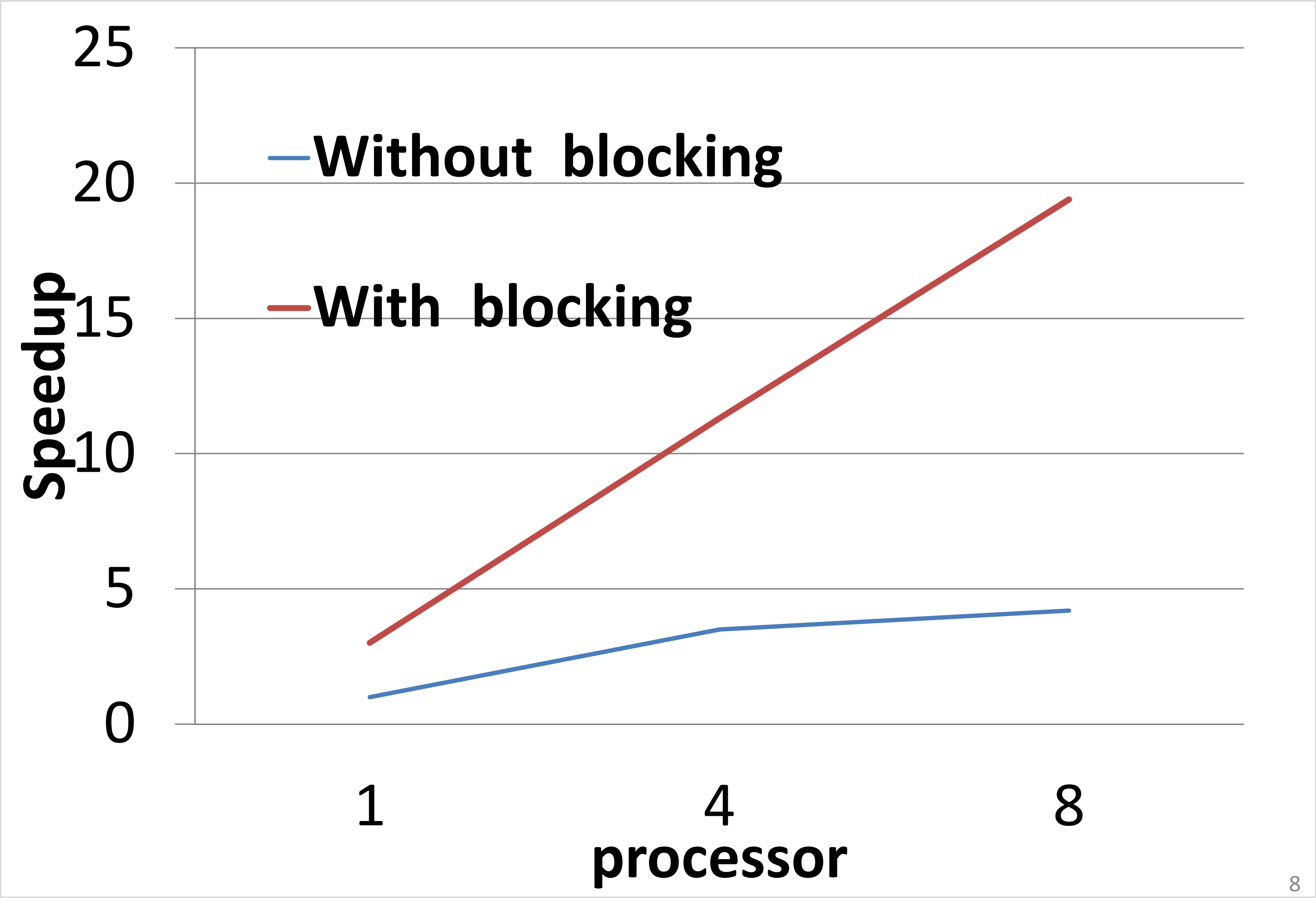
Blocking for Matmul: Stripmine 2 outerloop

```
for (ii = 0; ii < n; ii = ii+B) {
   for (i= ii; i< min(n,ii+B); i++) {
      for (jj= 0; jj< n; jj= jj+B) {
         for (j = jj; j < min(n,jj+B); j++) {
            for (k = 0; k < n; k++) {
               Z[i,j] = Z[i,j] + X[i,k]*Y[k,j];
}}}}}
```

Blocking for Matmul: permute

```
for (ii = 0; ii < n; ii = ii+B) {
   for (jj= 0; jj< n; jj= jj+B) {
      for (k = 0; k < n; k++) {
          for (i= ii; i< min(n,ii+B); i++) {
              for (j = jj; j < min(n,jj+B); j++) {
                  Z[i,j] = Z[i,j] + X[i,k]*Y[k,j];
}}}}}
```

Blocking for Matmul :Impact



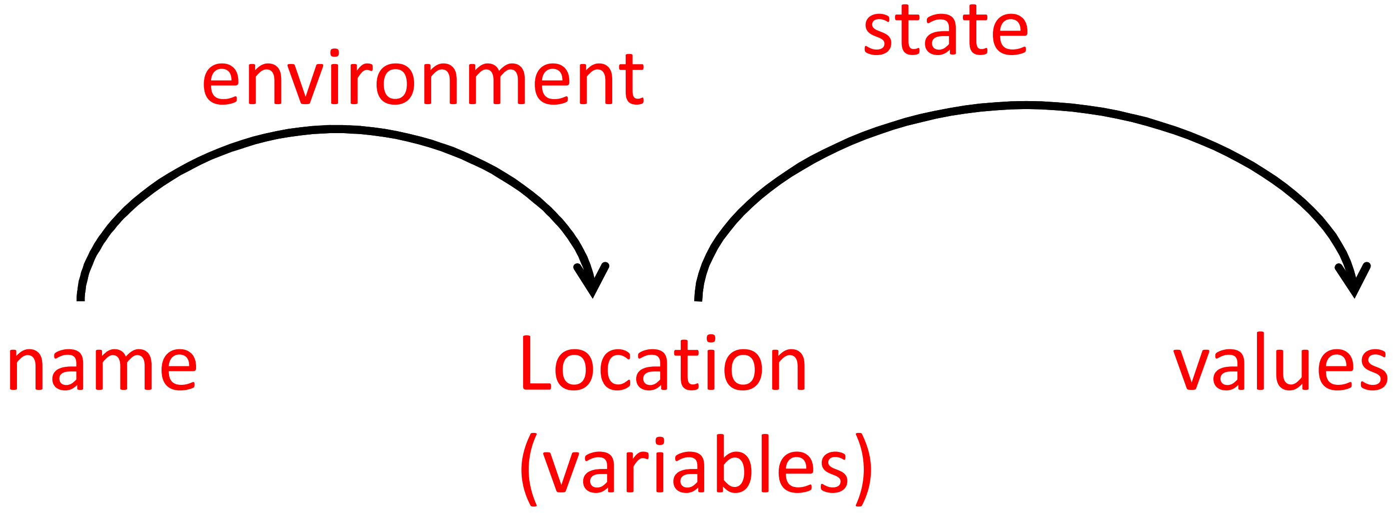
Programming Language Basic

Static Vs Dynamic Variable

- Static vs Dynamic Distinction
- Static Scope: Lexical/Compilation scope for a variable x
  - Global x,
  - static int x
- Dynamic Scope: as the program run same used of x   could refer to any several value of x

Environment and Scope

- Environment: mapping name to location in the memory (l-value of variable)
- States: mapping location to their values (r-value)



environment

state

name

Location (variables)

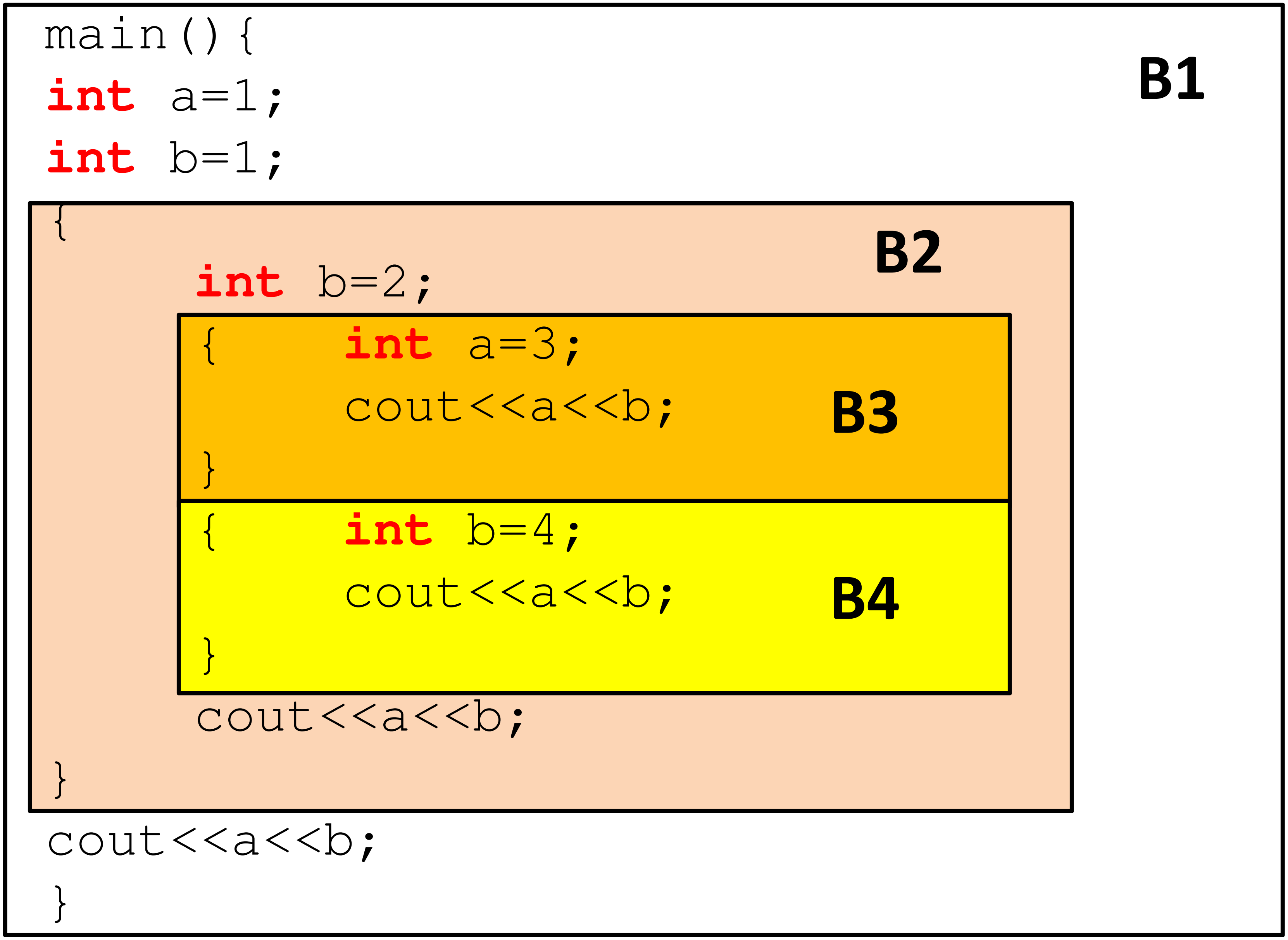
values

- int x=6; x have location &x, value =6

Scope and Block Structure

- Scope: Public, private, protected
- Static scope based on block: { ..}, begin..end
- C program consist of
  - top level declaration of variables
  - and functions
  - functions: may have variable declarations within them, scope is restricted with in that functions

Scope and Block Structure



```
main(){
int a=1;
int b=1;
{
       int b=2;
       {      int a=3;
              cout<<a<<b;
       }
       {      int b=4;
              cout<<a<<b;
       }
       cout<<a<<b;
}
cout<<a<<b;
}
```

Function Parameter Passing

- Actual parameter: used in the call of fun
- Formal parameter: used in fun definition
- Call by value: F(int A)
- Call by Reference: F(int *A), F(int &A)
- Call by name: F(int A)

- **Aliasing: can refer to same location**

```
F(int *A, int *B)//overlapped A & B
F(int * __restrict A, int *__restrict B)
```

Memory layout of C program

Dynamic memory allocation

- Reduce wastage of memory
- Useful when data size is unknown before hand
- Array Declaration

```
int A[100];
```

  - **Easy, Not to use pointer**, small size, known before
- Array Creation:
  - Not easy, use of pointer, typecast, **lager size, necessary size**

Memory management C: APIs

- Application program interfaces (APIs)
- Available function/APIs to manage memory
- Create/allocate/reserve space
  - malloc : memory allocation
  - calloc :   memory allocation + initialization to 0
- Move a reserved space to another location
  - realloc: move the space to another location
- Destroy/de-allocate/free space
    - free:

Memory Allocation

- Memory can be allocated
- Declaring a variable

```
int A[100];
```

- Explicitly requesting space

```
int *A;
A=(int*)malloc(sizeof(int)*100);
```

Example: Dynamic Array Allocation

- Given N persons (with their IQ level) in order
  - N may be dynamic, variable
- A person decide He/She is intelligent or dumb
- Decides locally:
  - If his/her IQ level is greater than equal to average of IQ level of both neighbors
  - Left neighbor and right neighbor

Example: Dynamic Array Allocation

```
main(){
    int *IQScore, *Intelligent, i, N;
    printf(“Input N:”); scanf(“%d”, &N);
    IQScore=(int*)malloc(N*sizeof(int));
    Intelligent =(int*)calloc(N*sizeof(int));
    for(i=0;i<N;i++)  scanf(“%d”,&IQScore[i]);
    for(i=1;i<N-1;i++){
        if(IQScore[i]>=(IQScore[i-1]+IQScore[i+1])/2)
          Intelligent[i]=1; else Intelligent[i]=0;
         printf(“ I am %d person  %s\n”, i,
              Intelligent[i]?”YES”:”NO”);
    }
    free(IQScore);   free(Intelligent);
    }
```

Memory layout of C program

- Program: Input, Output, Processing
- Code (Instruction), Data (Stack, Heap)
- To store: Require memory
  - Input data, output data, intermediate data
- Memory can be allocated
  - Declaring a variable

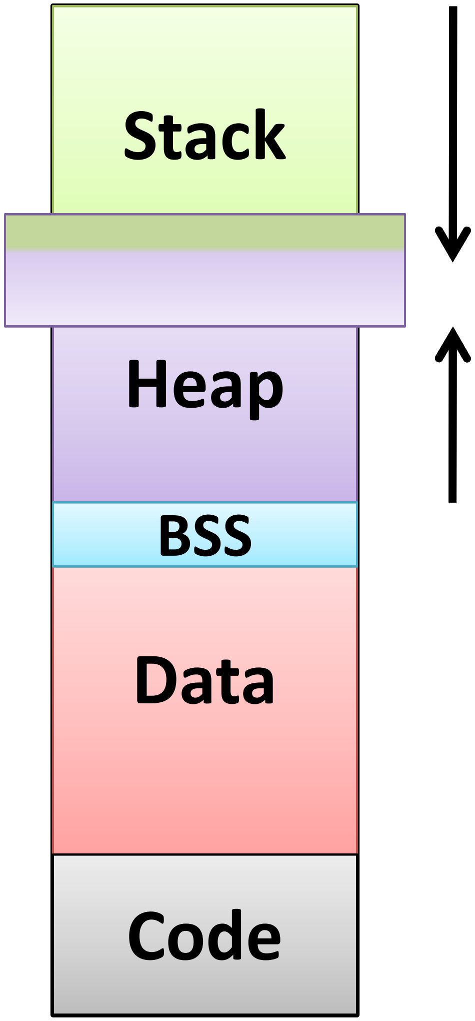
```
int A[100];
```

  - Explicitly requesting space

```
int *A;
A=(int*)malloc(sizeof(int)*100);
```

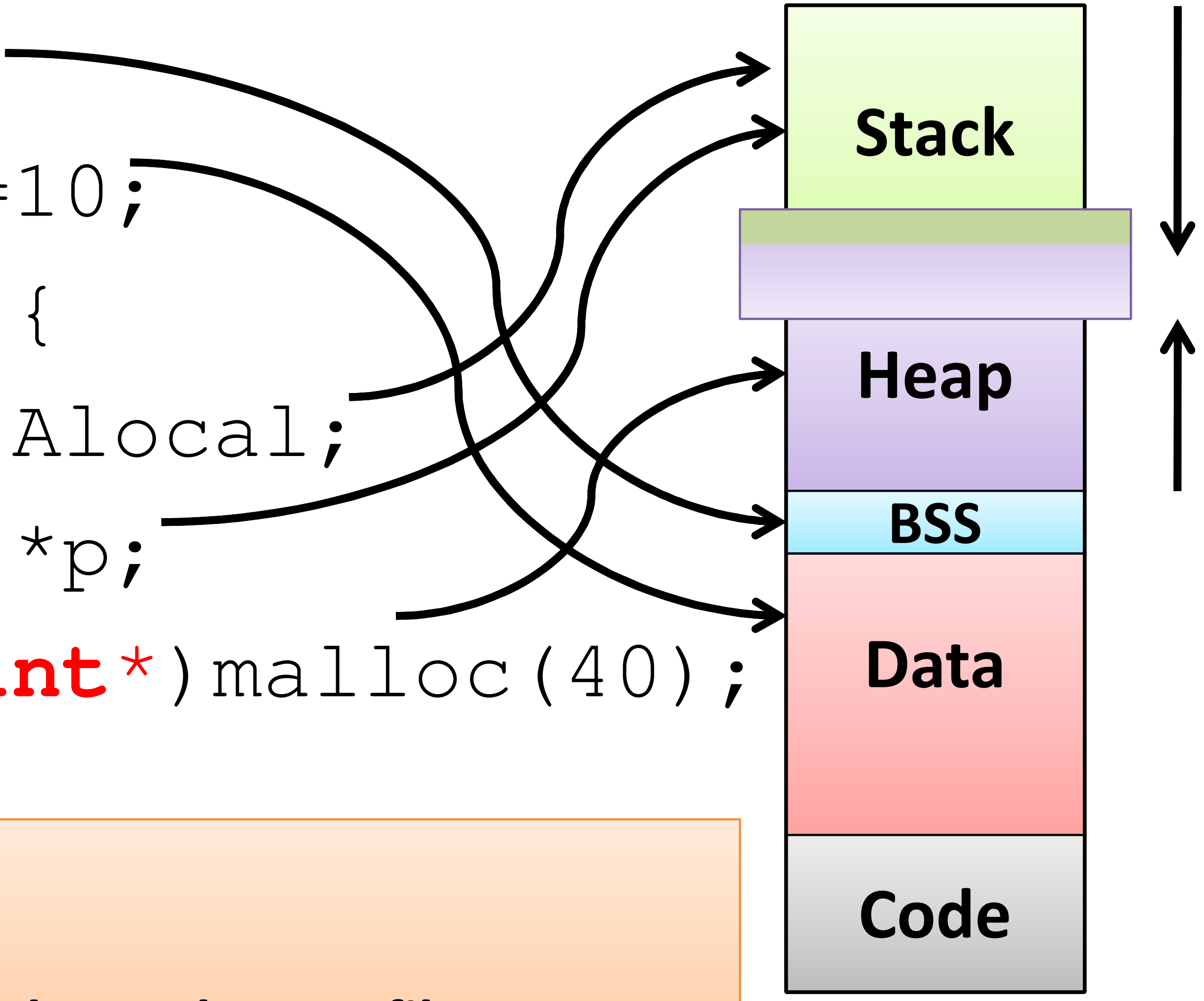
Memory layout of C program

- Stack
  - automatic (default), local
  - Initialized/uninitialized
- Data
  - Global, static, extern
  - BSS: Block Started by Symbol
  - BBS: Uninitialized Data Seg.
- Code : program instructions
- Heap
  - malloc, calloc



Memory layout of C program

```
int A;
int B=10;
main(){
   int Alocal;
   int *p;
   p=(int*)malloc(40);
}
```



```
$gcc  test.c
$size a.out
text    data    bss     dec     hex     filename
1200    544     8       1752    6d8     a.out
```

Compiling Multiple C Files

Compiling multiple Files

```c
//foo.c
int foo3x(int x){
        return 3*x;
 }
```

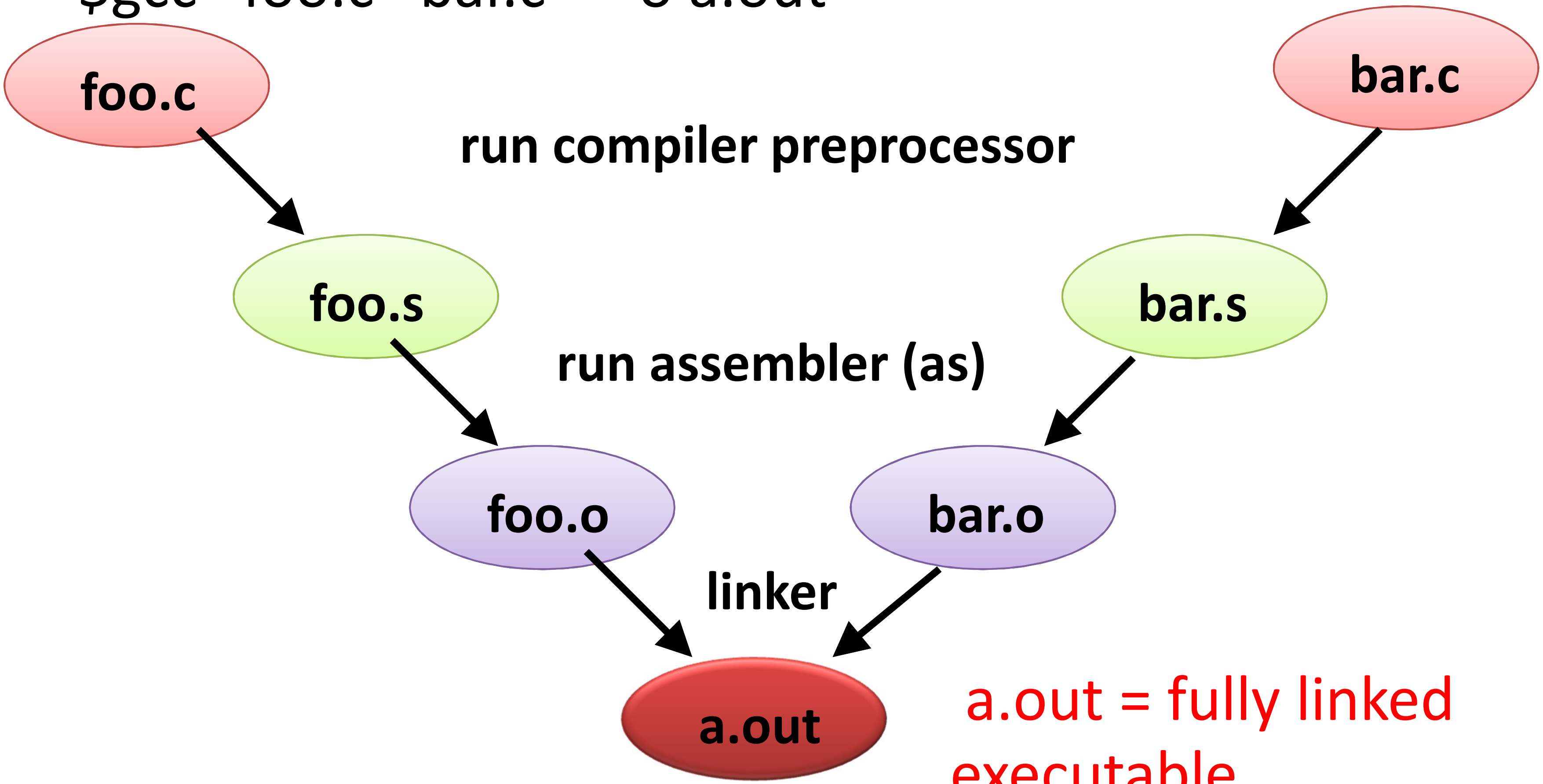
```c
//bar.c
int main(){
        int x;
        x=foo3x(10);
        printf(“%d”,x);
        return  0;
 }
```

- $ gcc –c foo.c
- $ gcc –c bar.c
- $ gcc foo.o bar.o
- $./a.out

Linker and Loader

- Compiler in Action…

  $gcc   foo.c   bar.c   –o a.out



a.out = fully linked executable

What is Linker ?

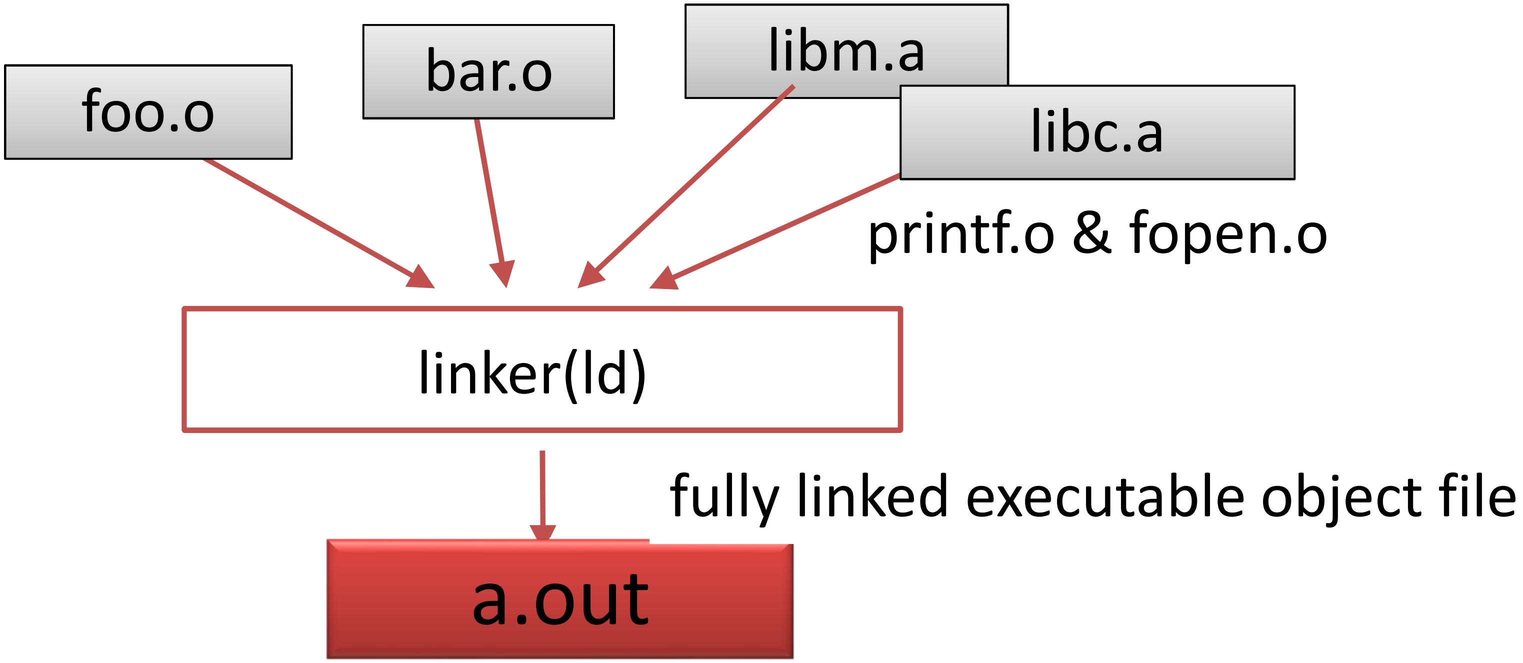
- Combines multiple relocatable object files
- Produces fully linked executable – directly loadable in memory
- How?
  - Symbol resolution – associating one symbol definition with each symbol reference
  - Relocation – relocating different sections of input relocatable files

Object files

- Types –
  - Relocatable : Requires linking to create executable
  - Executable : Loaded directly into memory for execution
  - Shared Objects : Linked dynamically, at run time or load time

Linking with Static Libraries

- Collection of concatenated object files – stored on disk in a particular format – archive
- An input to Linker
  - Referenced object files copied to executable



Creating Static Library

```
//foo.c
int foo3x(int x){
        return 3*x;
 }
```

```
int main(){//bar.c
        int x;
        x=foo3x(10);
        printf(“%d”,x);
        return  0;
 }
```

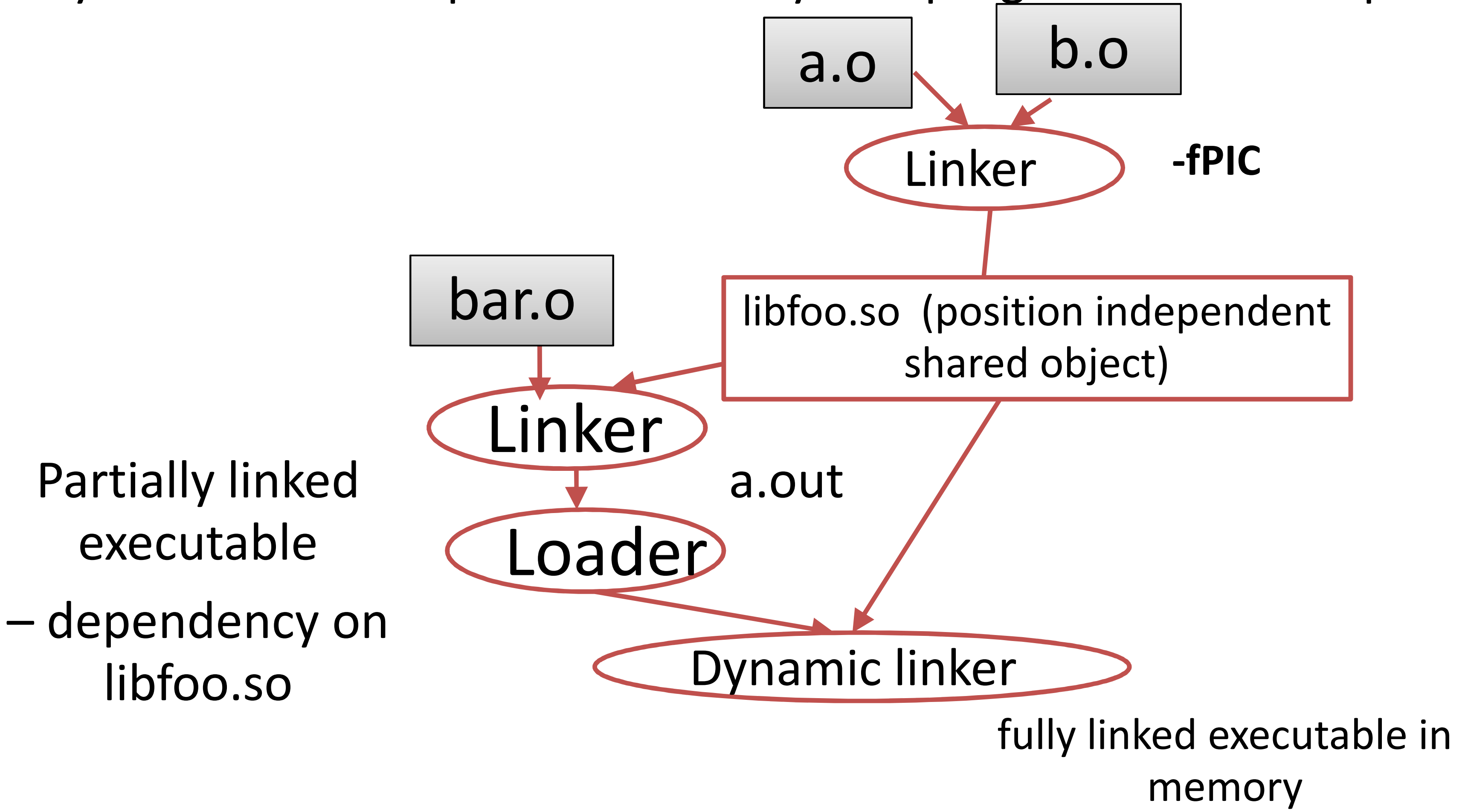
- `$ gcc -c foo.c`
- `$ ar rcs libfoo.a  foo.o`   //it create libfoo.a
- `$ gcc  bar.c -L. -lfoo`
- `$ ./a.out`

Dynamic Linking – Shared Libraries

- Addresses disadvantages of static libraries
  - Ensures one copy of text & data in memory
  - Change in shared library does not require executable to be built again
  - Loaded at run-time by dynamic linker, at arbitrary memory address, linked with programs in memory
  - On loading, dynamic linker relocates text & data of shared object

Shared Libraries ..(Cntd)

- Linker creates libfoo.so (PIC) from a.o b.o
- a.out – partially executable – dependency on libfoo.so
- Dynamic linker maps shared library into program's address space



a.o

b.o

Linker

-fPIC

bar.o

libfoo.so (position independent shared object)

Linker

a.out

Partially linked executable

– dependency on libfoo.so

Loader

Dynamic linker

fully linked executable in memory

Creating Dynamic Library

```
//foo.c
int foo3x(int x){
        return 3*x;
 }
```

```
int main(){//bar.c
        int x;
        x=foo3x(10);
        printf(“%d”,x);
        return  0;
 }
```

- $gcc -c -fPIC foo.c
- $gcc -shared -Wl,-soname,libfoo.so.1 -o libfoo.so.1 foo.o
- $ gcc bar.c -L. –lfoo
- $ export LD_LIBRARY_PATH=.
- $ ./a.out

Thanks