Data Oriented Design

Data Oriented Design

Thinking about the data layout and its performance impacts is at the core of a new and growing programming approach called **data-oriented design**. The approach is to think about:

- **Data** rather than code
- Memory **bandwidth** rather than flops
- **Cache line** instead of individual data elements
- Operations prioritized on data already in cache

DOD *vs* OOP

In object-oriented programming data is grouped in structures and processed by methods. With are large numbers of objects of a class, OOP usually invokes a method for each object. We have:

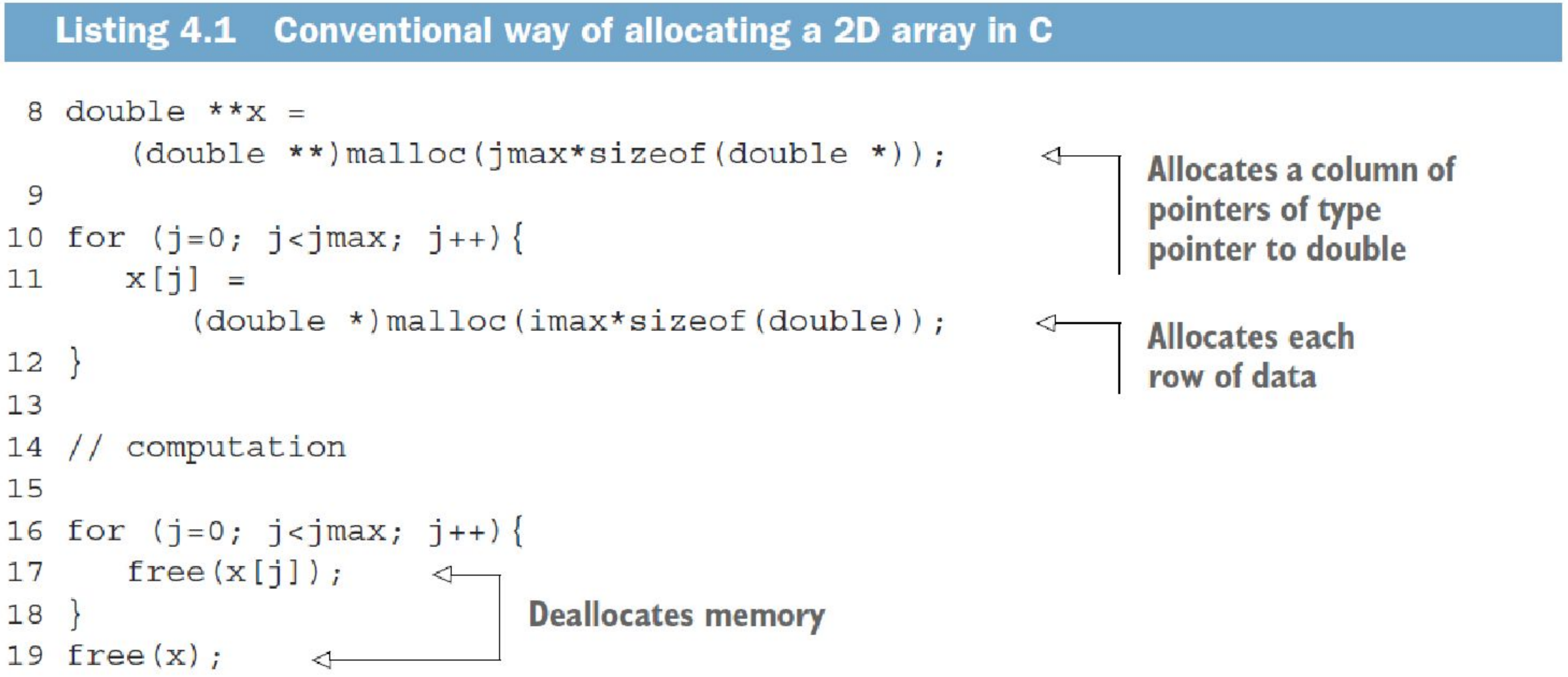
- deep call stack (no inline)
- Poor data cache utilization/ memory bandwidth
- Frequent **heap allocations**
- **Polymorphism** with virtual function tables (vtable), causes indirect function calls and branch mispredictions.

DOD

Data oriented design:

- Operates on **arrays**, not individual data items,
- Prefers **arrays** rather than structures for better cache usage
- Uses **contiguous array-based** linked lists to avoid the standard linked list implementations
- Inlines subroutines rather than transversing a deep call hierarchy
- Controls memory allocation, avoiding undirected reallocation behind the scenes

Multidimensional arrays



Listing 4.1 Conventional way of allocating a 2D array in C

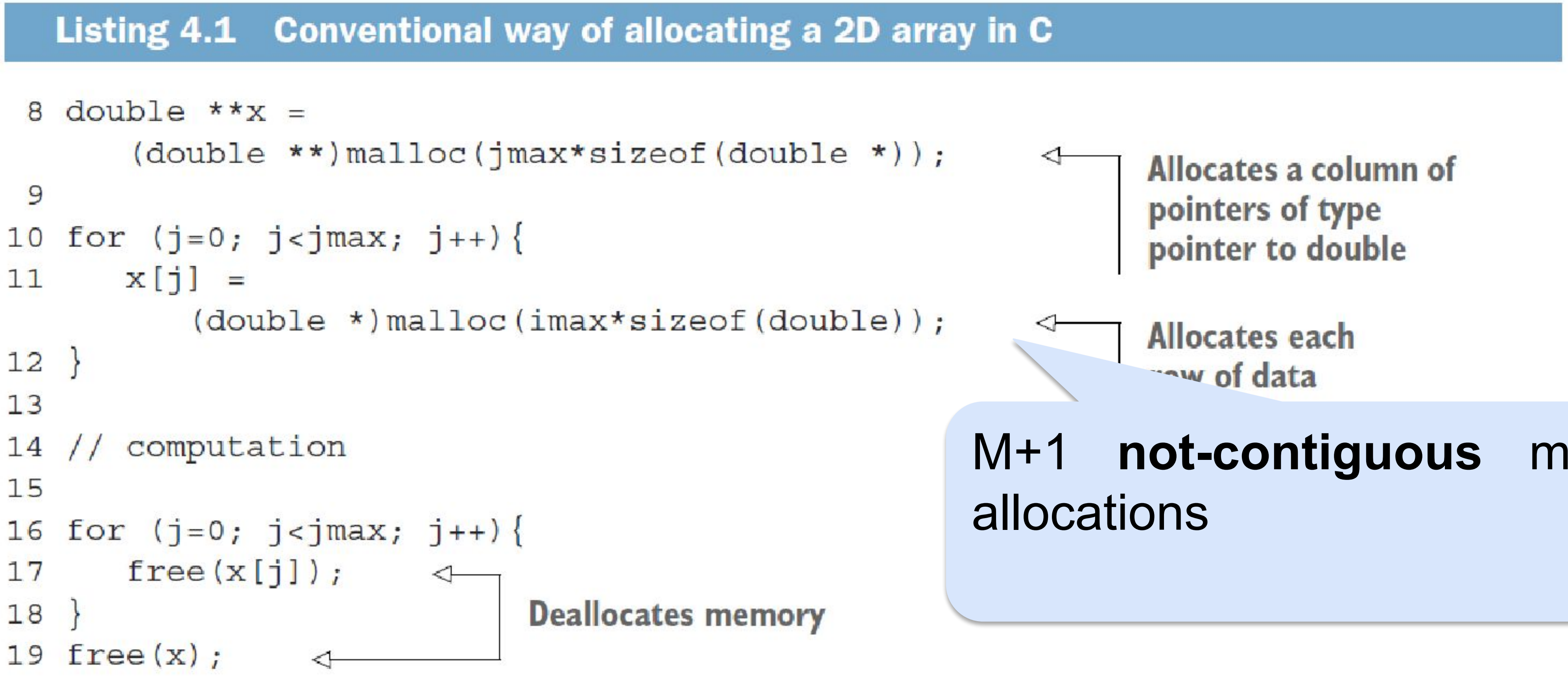
```c
double **x =
    (double **)malloc(jmax*sizeof(double *));

for (j=0; j<jmax; j++){
   x[j] =
       (double *)malloc(imax*sizeof(double));
}

// computation

for (j=0; j<jmax; j++){
   free(x[j]);
}
free(x);
```

Allocates a column of pointers of type pointer to double

Allocates each row of data

Deallocates memory

Multidimensional arrays



Listing 4.1 Conventional way of allocating a 2D array in C

```
double **x =
    (double **)malloc(jmax*sizeof(double *));

for (j=0; j<jmax; j++){
   x[j] =
       (double *)malloc(imax*sizeof(double));
}

// computation

for (j=0; j<jmax; j++){
   free(x[j]);
}
free(x);
```

Allocates a column of pointers of type pointer to double

Allocates each row of data

Deallocates memory

M+1 **not-contiguous** memory allocations

Multidimensional arrays

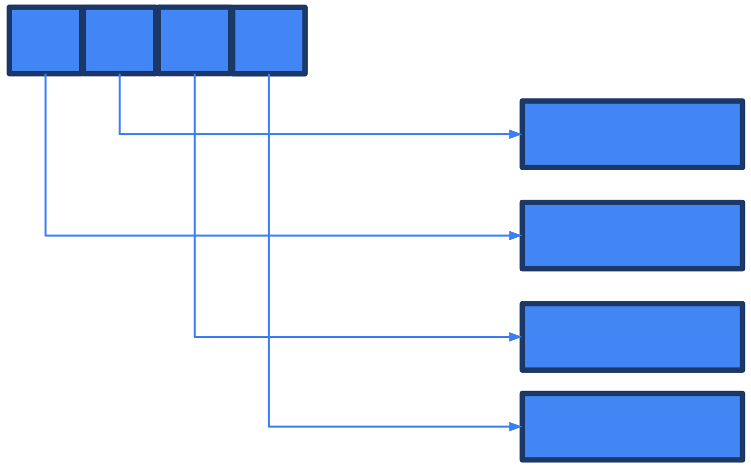
Listing 4.1 Conventional way of allocating a 2D array in C

```
double **x =
    (double **)malloc(jmax*sizeof(double *));

for (j=0; j<jmax; j++){
    x[j] =
        (double *)malloc(imax*sizeof(double));
}

// computation

for (j=0; j<jmax; j++){
    free(x[j]);
}
free(x);
```

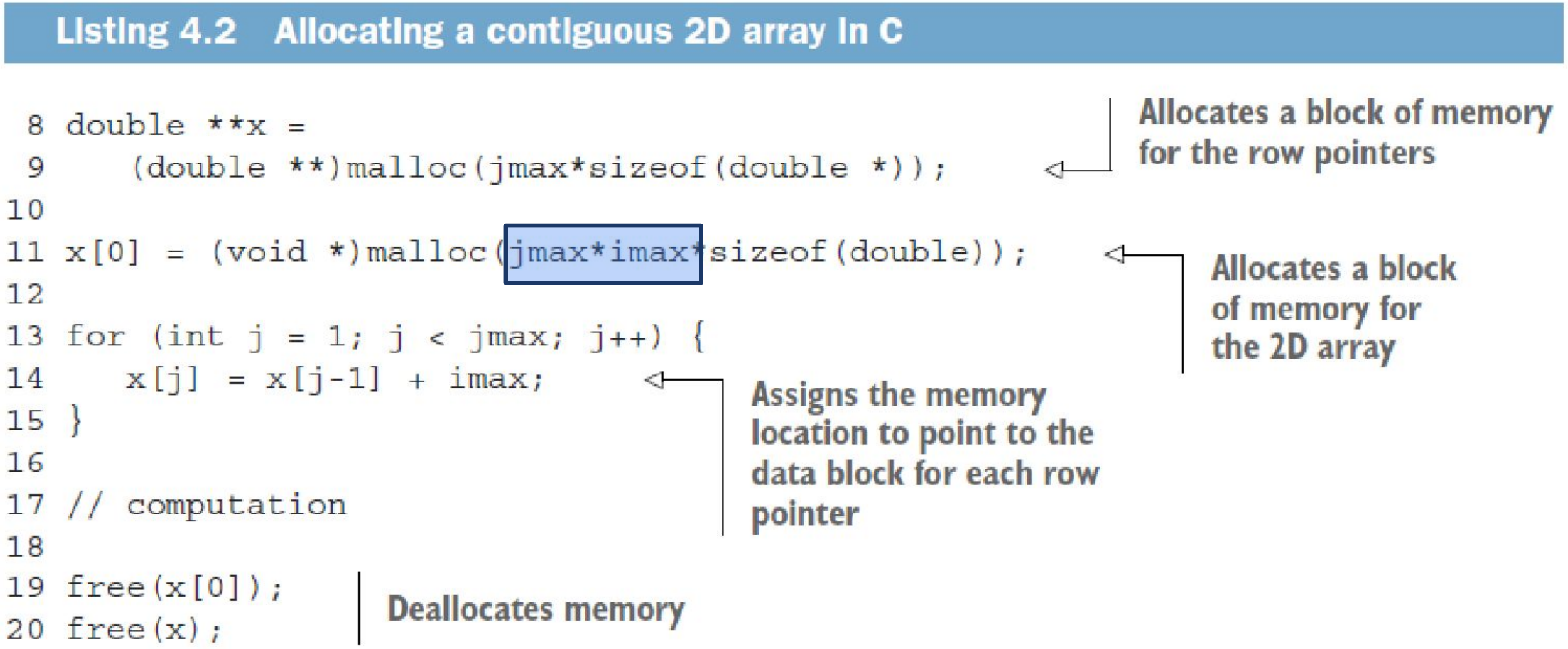
Allocates a column of pointers of type pointer to double

Allocates each row of data

Deallocates memory



Multidimensional arrays



Listing 4.2 Allocating a contiguous 2D array in C

```c
double **x =
   (double **)malloc(jmax*sizeof(double *));

x[0] = (void *)malloc(jmax*imax*sizeof(double));

for (int j = 1; j < jmax; j++) {
   x[j] = x[j-1] + imax;
}

// computation

free(x[0]);
free(x);
```

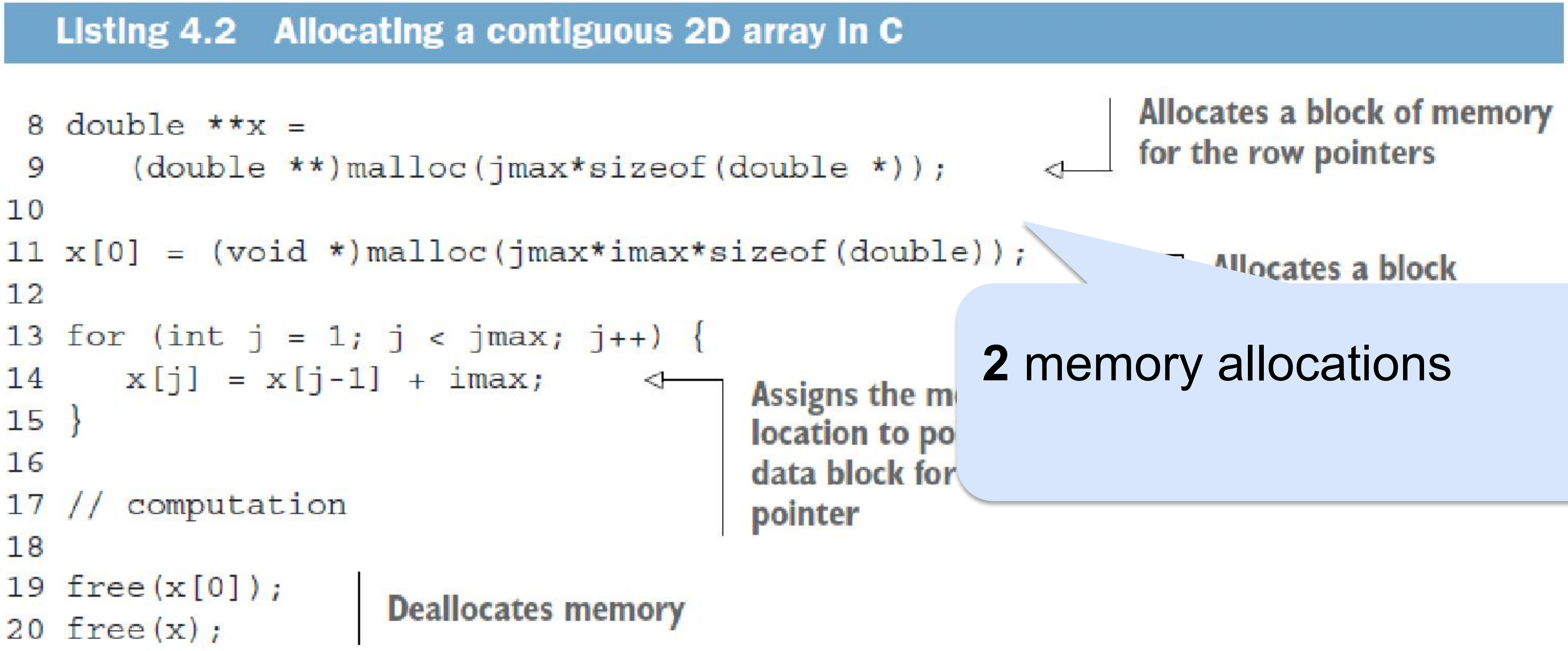
Allocates a block of memory for the row pointers

Allocates a block of memory for the 2D array

Assigns the memory location to point to the data block for each row pointer

Deallocates memory

Multidimensional arrays



Listing 4.2 Allocating a contiguous 2D array in C

```c
double **x =
   (double **)malloc(jmax*sizeof(double *));

x[0] = (void *)malloc(jmax*imax*sizeof(double));

for (int j = 1; j < jmax; j++) {
   x[j] = x[j-1] + imax;
}

// computation

free(x[0]);
free(x);
```

Allocates a block of memory for the row pointers

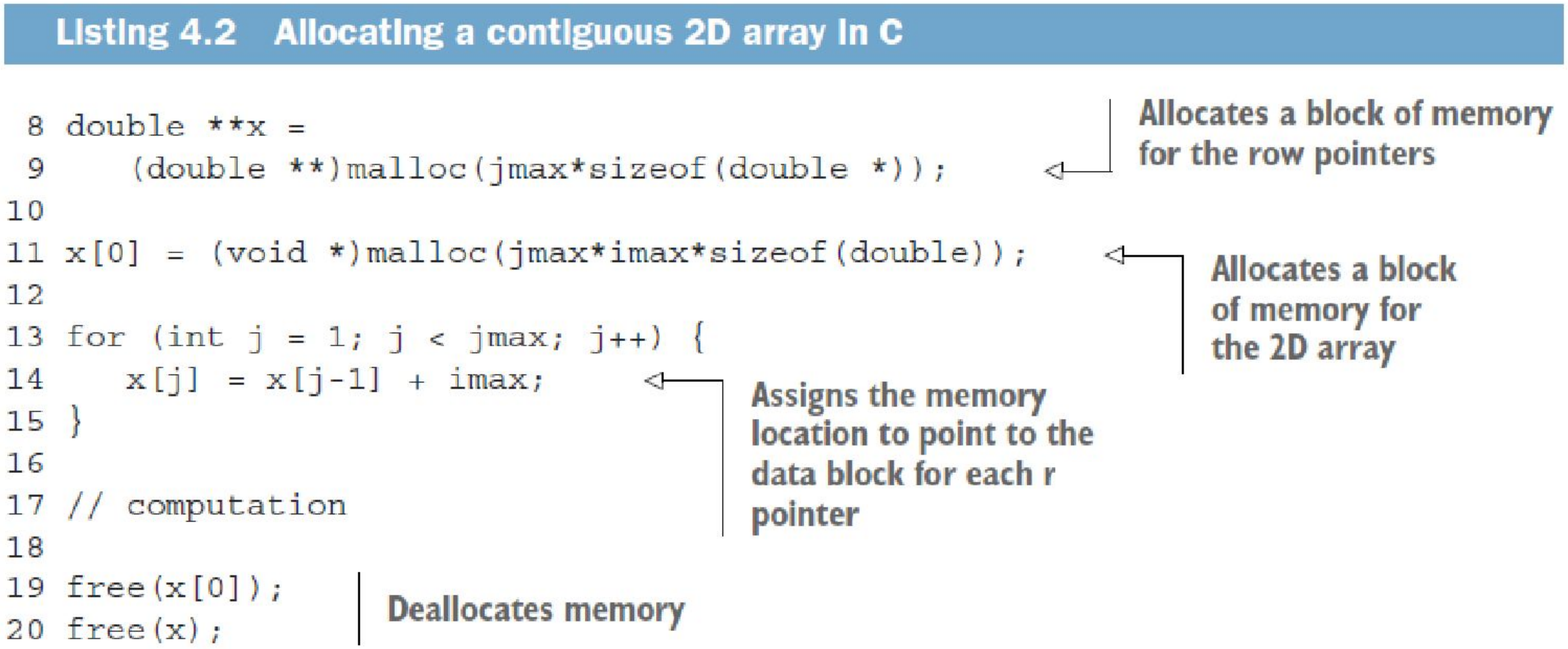
Allocates a block

Assigns the me
location to po
data block for
pointer

Deallocates memory

2 memory allocations

Multidimensional arrays



Listing 4.2 Allocating a contiguous 2D array in C

```c
double **x =
   (double **)malloc(jmax*sizeof(double *));

x[0] = (void *)malloc(jmax*imax*sizeof(double));

for (int j = 1; j < jmax; j++) {
   x[j] = x[j-1] + imax;
}

// computation

free(x[0]);
free(x);
```

Allocates a block of memory for the row pointers

Allocates a block of memory for the 2D array

Assigns the memory location to point to the data block for each r pointer

Deallocates memory

Multidimensional arrays

Listing 4.3 Single contiguous memory allocation for a 2D array

malloc2D.c

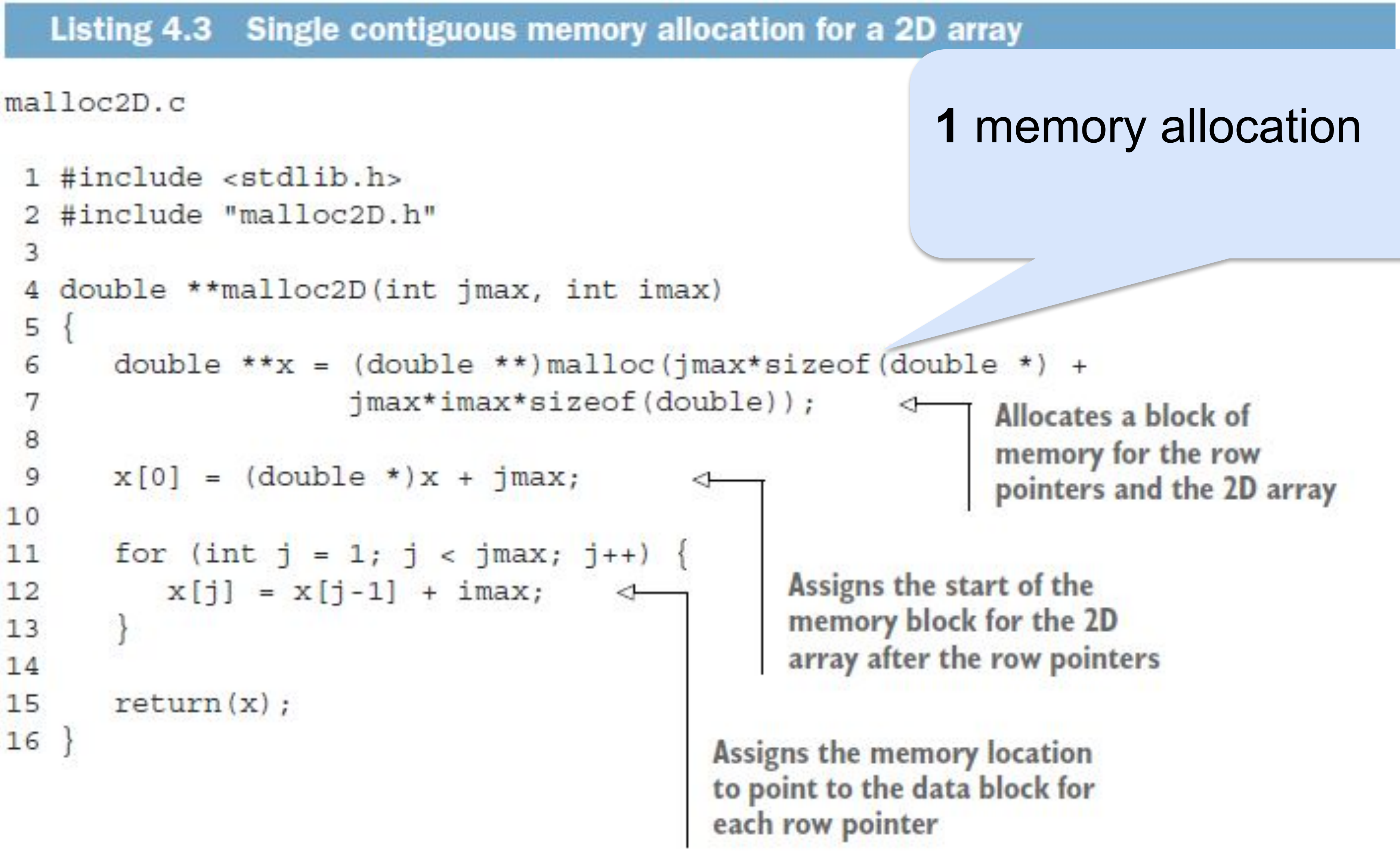
```c
#include <stdlib.h>
#include "malloc2D.h"

double **malloc2D(int jmax, int imax)
{
   double **x = (double **)malloc(jmax*sizeof(double *) +
                  jmax*imax*sizeof(double));

   x[0] = (double *)x + jmax;

   for (int j = 1; j < jmax; j++) {
      x[j] = x[j-1] + imax;
   }

   return(x);
}
```

Allocates a block of memory for the row pointers and the 2D array

Assigns the start of the memory block for the 2D array after the row pointers

Assigns the memory location to point to the data block for each row pointer

Multidimensional arrays



Listing 4.3 Single contiguous memory allocation for a 2D array

malloc2D.c

```c
#include <stdlib.h>
#include "malloc2D.h"

double **malloc2D(int jmax, int imax)
{
   double **x = (double **)malloc(jmax*sizeof(double *) +
                jmax*imax*sizeof(double));

   x[0] = (double *)x + jmax;

   for (int j = 1; j < jmax; j++) {
      x[j] = x[j-1] + imax;
   }

   return(x);
}
```

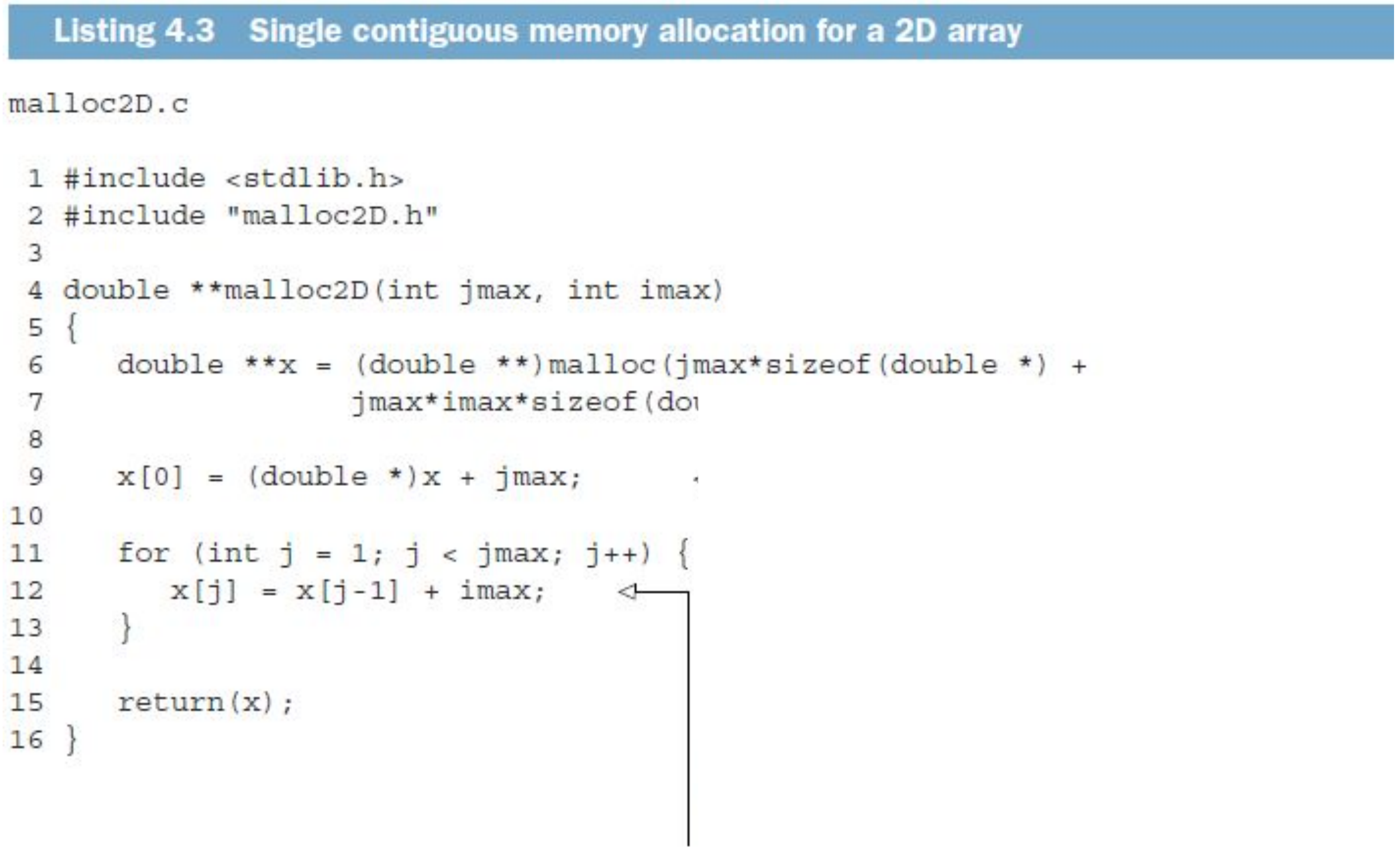
Allocates a block of memory for the row pointers and the 2D array

Assigns the start of the memory block for the 2D array after the row pointers

Assigns the memory location to point to the data block for each row pointer

1 memory allocation

Multidimensional arrays

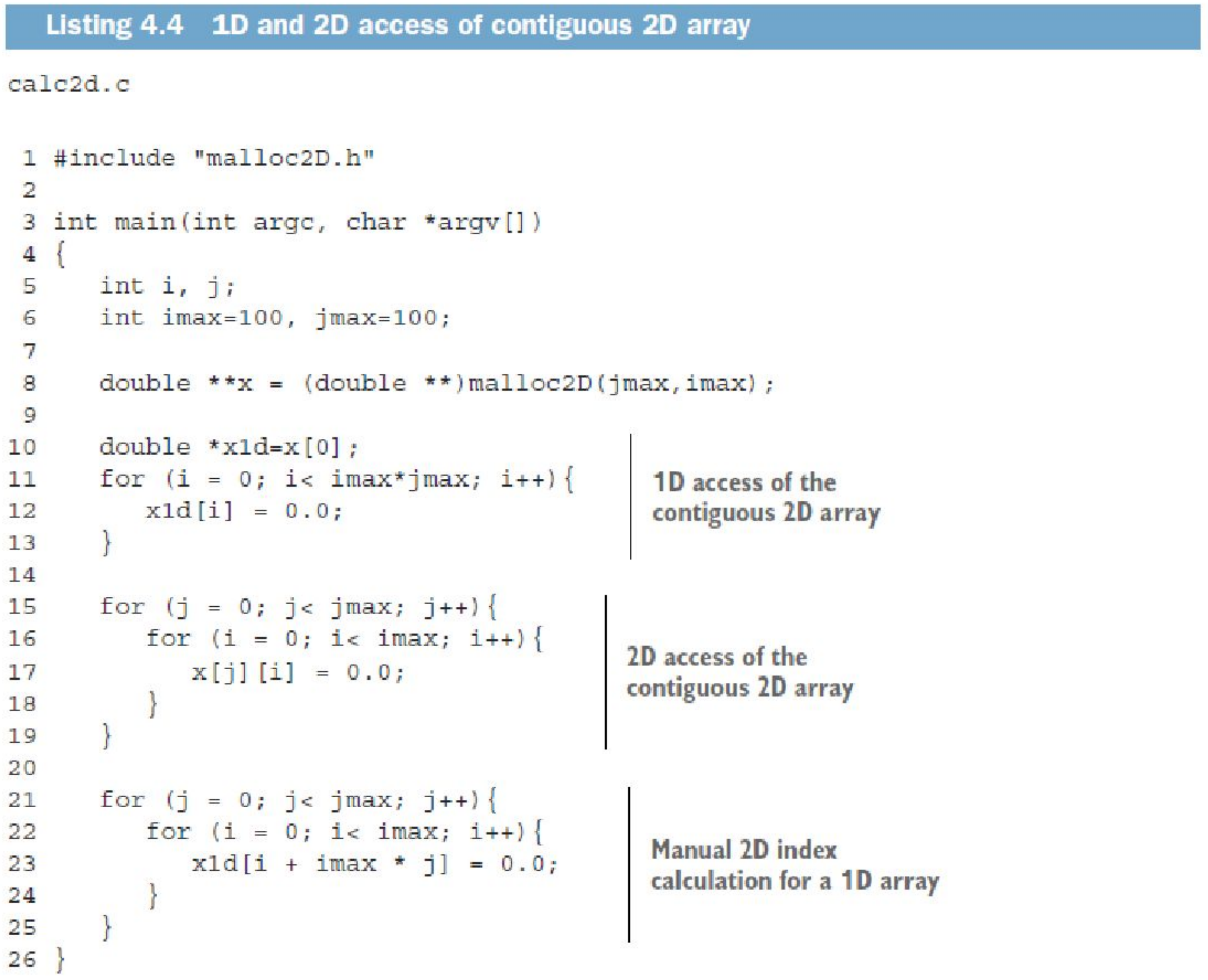


Listing 4.3 Single contiguous memory allocation for a 2D array

malloc2D.c

```
#include <stdlib.h>
#include "malloc2D.h"

double **malloc2D(int jmax, int imax)
{
   double **x = (double **)malloc(jmax*sizeof(double *) +
                     jmax*imax*sizeof(dou

   x[0] = (double *)x + jmax;

   for (int j = 1; j < jmax; j++) {
      x[j] = x[j-1] + imax;
   }

   return(x);
}
```

Multidimensional arrays



Listing 4.4 1D and 2D access of contiguous 2D array

calc2d.c

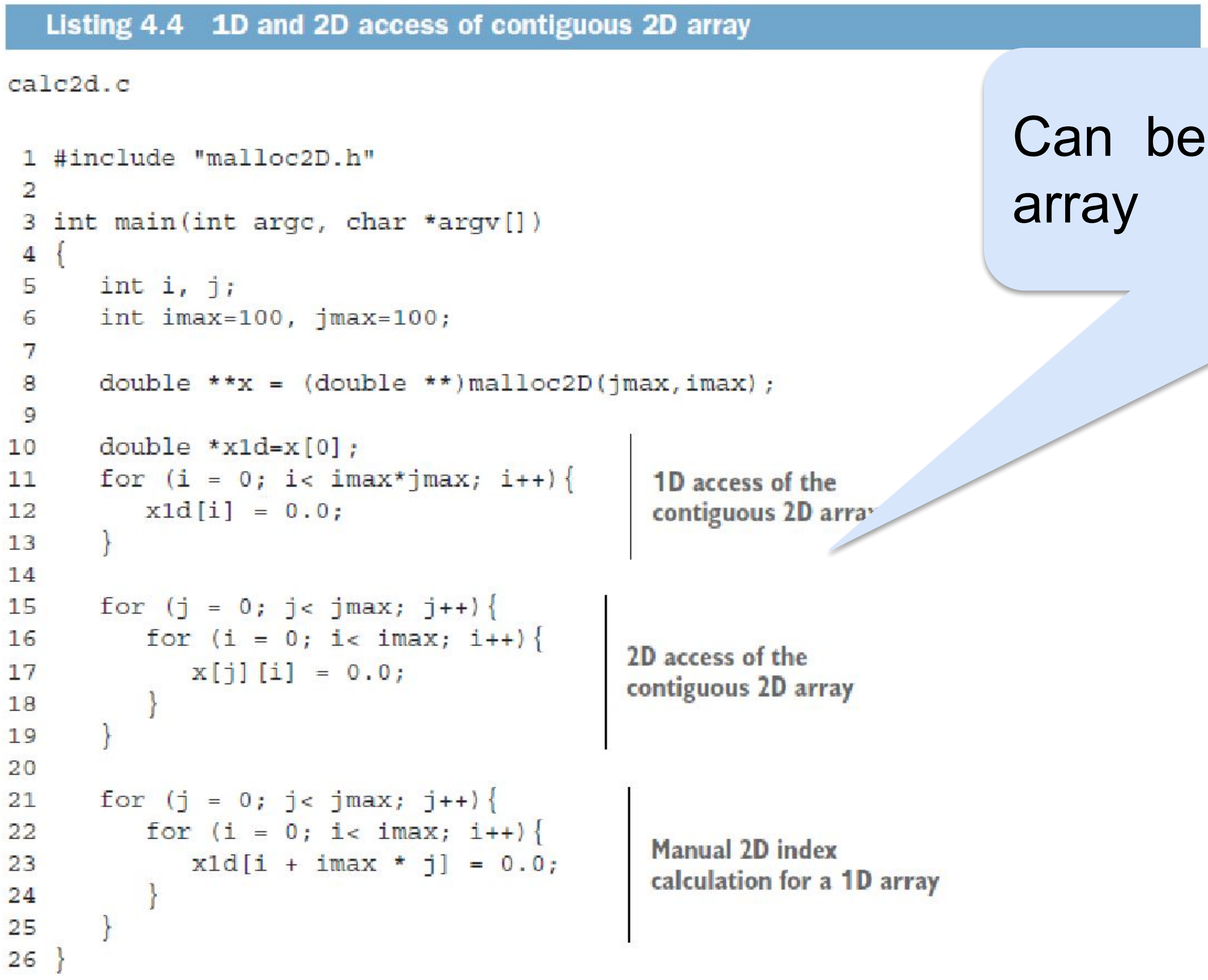
```c
#include "malloc2D.h"

int main(int argc, char *argv[])
{
   int i, j;
   int imax=100, jmax=100;

   double **x = (double **)malloc2D(jmax,imax);

   double *x1d=x[0];
   for (i = 0; i< imax*jmax; i++){
      x1d[i] = 0.0;
   }

   for (j = 0; j< jmax; j++){
      for (i = 0; i< imax; i++){
         x[j][i] = 0.0;
      }
   }

   for (j = 0; j< jmax; j++){
      for (i = 0; i< imax; i++){
         x1d[i + imax * j] = 0.0;
      }
   }
}
```

1D access of the contiguous 2D array

2D access of the contiguous 2D array

Manual 2D index calculation for a 1D array

Multidimensional arrays



Listing 4.4 1D and 2D access of contiguous 2D array

calc2d.c

```c
#include "malloc2D.h"

int main(int argc, char *argv[])
{
   int i, j;
   int imax=100, jmax=100;

   double **x = (double **)malloc2D(jmax,imax);

   double *x1d=x[0];
   for (i = 0; i< imax*jmax; i++){
      x1d[i] = 0.0;
   }

   for (j = 0; j< jmax; j++){
      for (i = 0; i< imax; i++){
         x[j][i] = 0.0;
      }
   }

   for (j = 0; j< jmax; j++){
      for (i = 0; i< imax; i++){
         x1d[i + imax * j] = 0.0;
      }
   }
}
```

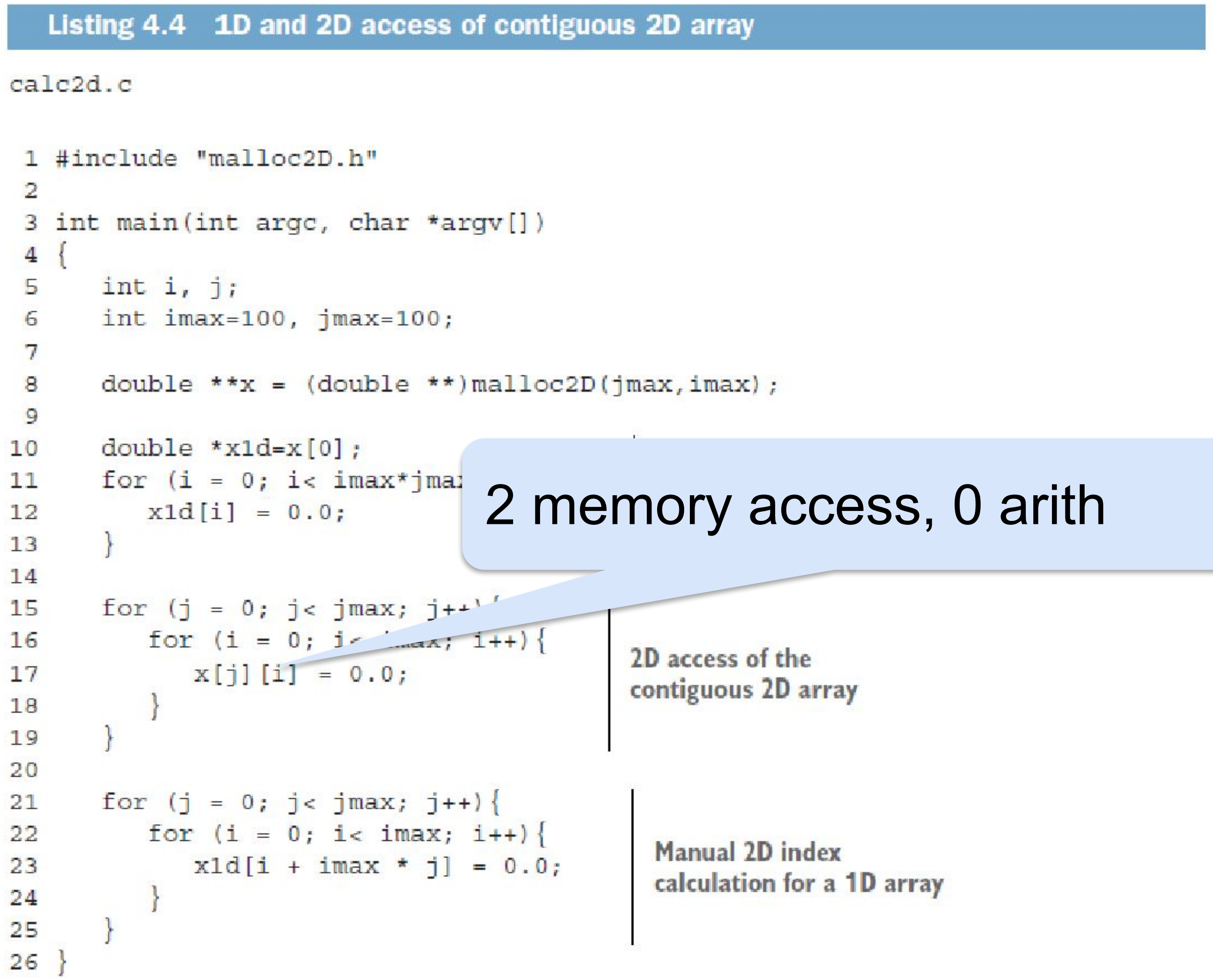
1D access of the contiguous 2D arra

2D access of the contiguous 2D array

Manual 2D index calculation for a 1D array

Can be accessed as 1D or 2D array

Multidimensional arrays



Listing 4.4 1D and 2D access of contiguous 2D array

calc2d.c

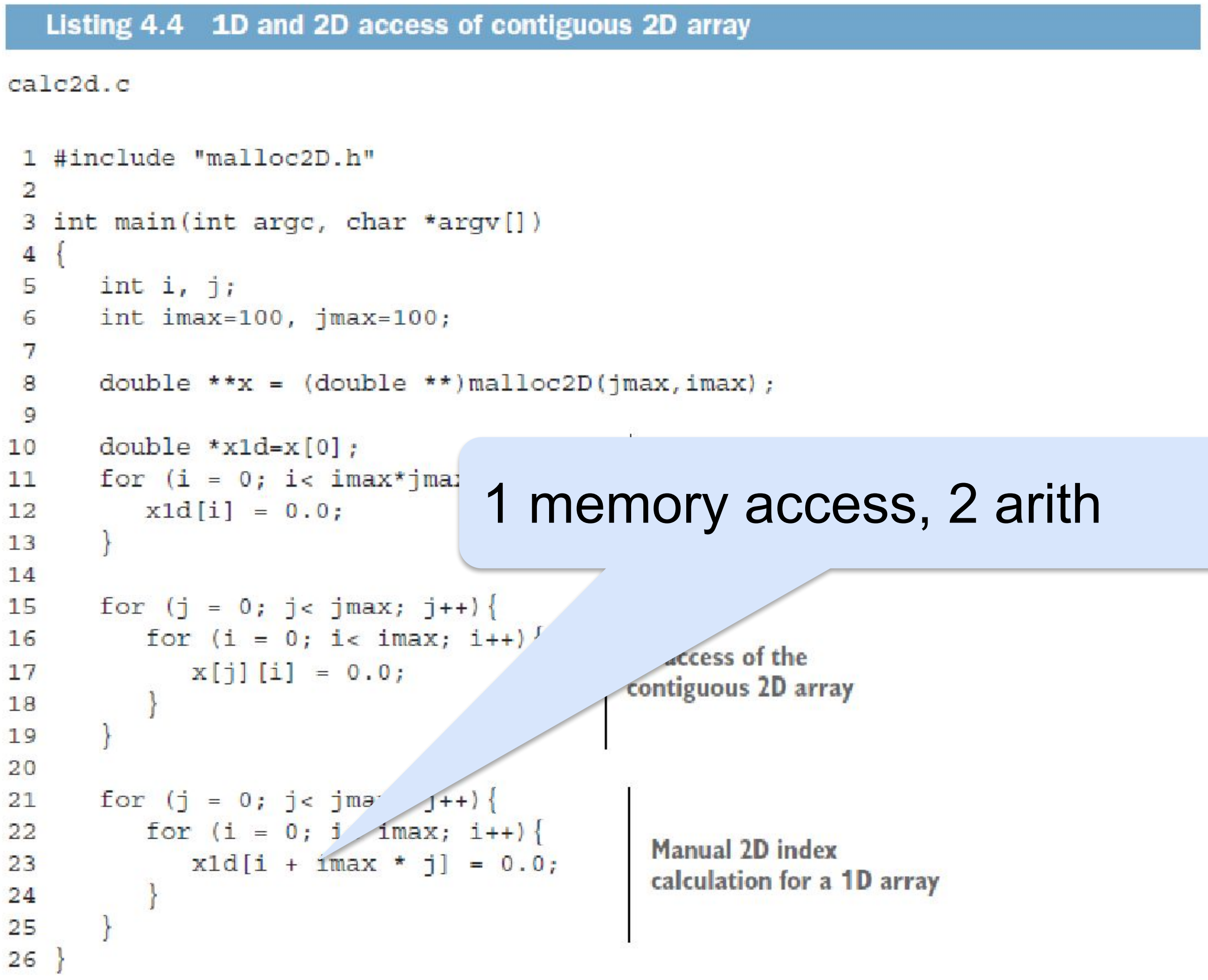
```
#include "malloc2D.h"

int main(int argc, char *argv[])
{
   int i, j;
   int imax=100, jmax=100;

   double **x = (double **)malloc2D(jmax,imax);

   double *x1d=x[0];
   for (i = 0; i< imax*jma
      x1d[i] = 0.0;
   }

   for (j = 0; j< jmax; j++){
      for (i = 0; i< imax; i++){
         x[j][i] = 0.0;
      }
   }

   for (j = 0; j< jmax; j++){
      for (i = 0; i< imax; i++){
         x1d[i + imax * j] = 0.0;
      }
   }
}
```

2D access of the contiguous 2D array

Manual 2D index calculation for a 1D array

2 memory access, 0 arith

Multidimensional arrays



Listing 4.4 1D and 2D access of contiguous 2D array

calc2d.c

```
#include "malloc2D.h"

int main(int argc, char *argv[])
{
   int i, j;
   int imax=100, jmax=100;

   double **x = (double **)malloc2D(jmax,imax);

   double *x1d=x[0];
   for (i = 0; i< imax*jma
      x1d[i] = 0.0;
   }

   for (j = 0; j< jmax; j++){
      for (i = 0; i< imax; i++){
         x[j][i] = 0.0;
      }
   }

   for (j = 0; j< jma   j++){
      for (i = 0; i  imax; i++){
         x1d[i + imax * j] = 0.0;
      }
   }
}
```

ccess of the contiguous 2D array

Manual 2D index calculation for a 1D array

1 memory access, 2 arith

AoS *vs* SoA

There are two different ways to organize related data into data collections.

- Array of Structures (**AoS**): the data is collected into a single unit at the lowest level and then an array is made of the structure

- Structure of Arrays (**SoA**): each data array is at the lowest level and then a structure is made of the arrays.

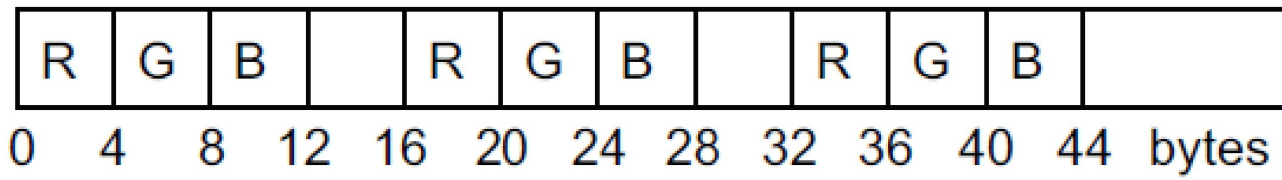
Array of Structures (AoS)

Listing 4.5 Array of Structures (AoS) in C

```
struct RGB {
    int R;
    int G;
    int B;
};
struct RGB polygon_color[1000];
```

Defines a scalar color value

Defines an Array of Structures (AoS)



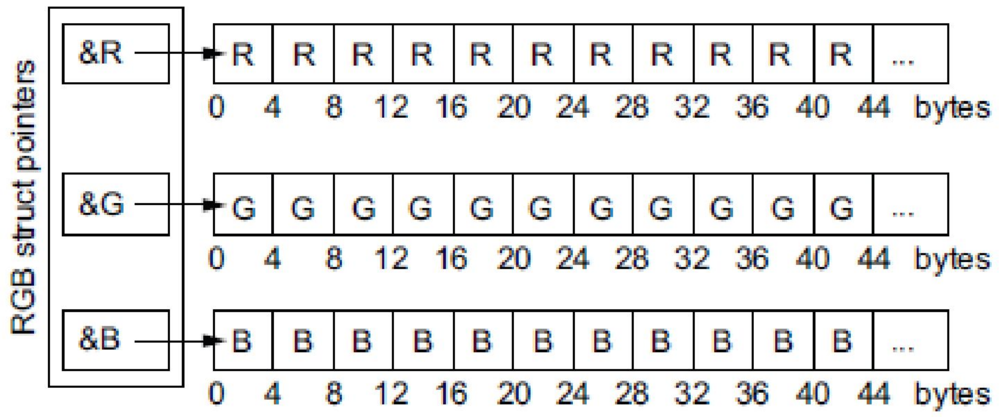
Structure of Arrays (SoA)

Listing 4.6 Structure of Arrays (SoA) in C

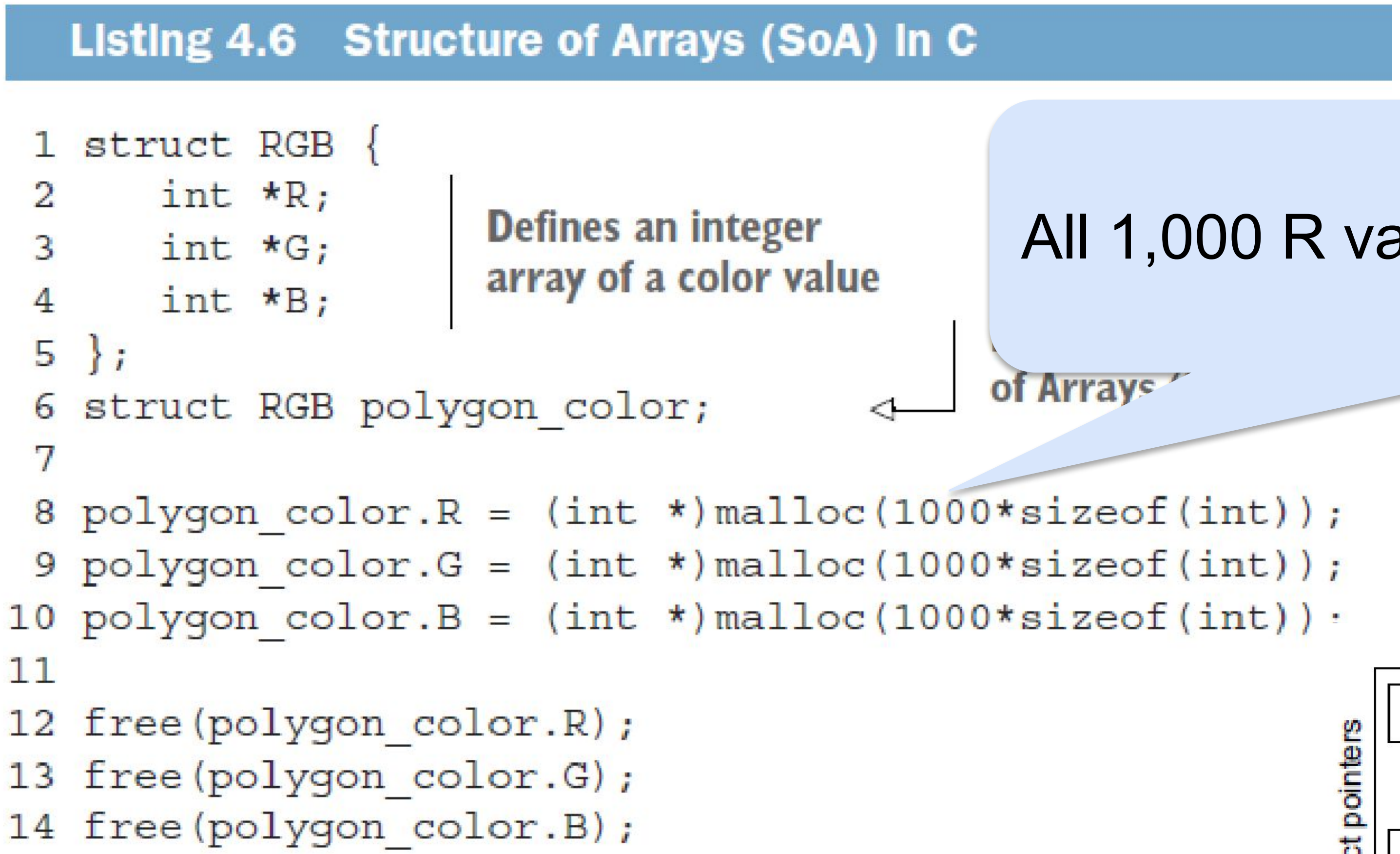
```c
struct RGB {
    int *R;
    int *G;
    int *B;
};
struct RGB polygon_color;

polygon_color.R = (int *)malloc(1000*sizeof(int));
polygon_color.G = (int *)malloc(1000*sizeof(int));
polygon_color.B = (int *)malloc(1000*sizeof(int));

free(polygon_color.R);
free(polygon_color.G);
free(polygon_color.B);
```

Defines an integer array of a color value

Defines a Structure of Arrays (SoA)


RGB struct pointers
&R
&G
&B
R R R R R R R R R R R ...
G G G G G G G G G G G ...
B B B B B B B B B B B ...
0 4 8 12 16 20 24 28 32 36 40 44 bytes

Structure of Arrays (SoA)



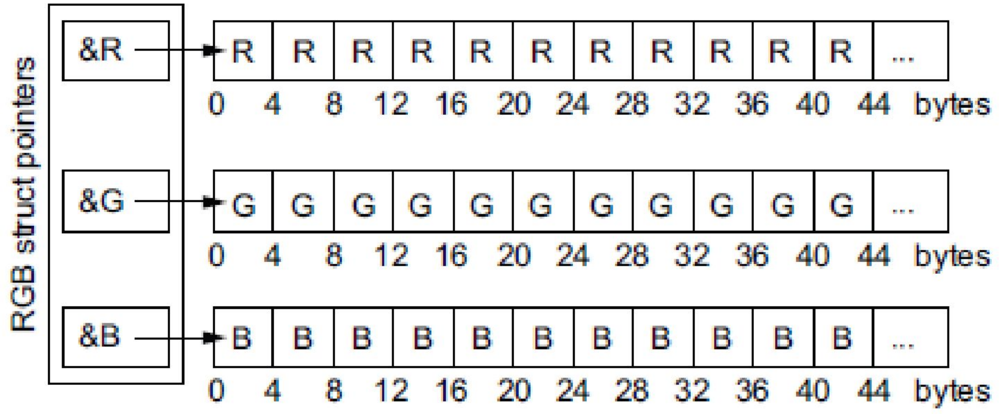
Listing 4.6 Structure of Arrays (SoA) in C

```c
struct RGB {
    int *R;
    int *G;
    int *B;
};
struct RGB polygon_color;

polygon_color.R = (int *)malloc(1000*sizeof(int));
polygon_color.G = (int *)malloc(1000*sizeof(int));
polygon_color.B = (int *)malloc(1000*sizeof(int))

free(polygon_color.R);
free(polygon_color.G);
free(polygon_color.B);
```

Defines an integer array of a color value

of Arrays

All 1,000 R values in contiguous memory.


RGB struct pointers
&R
&G
&B
R R R R R R R R R R R ...
G G G G G G G G G G G ...
B B B B B B B B B B B ...
0 4 8 12 16 20 24 28 32 36 40 44 bytes

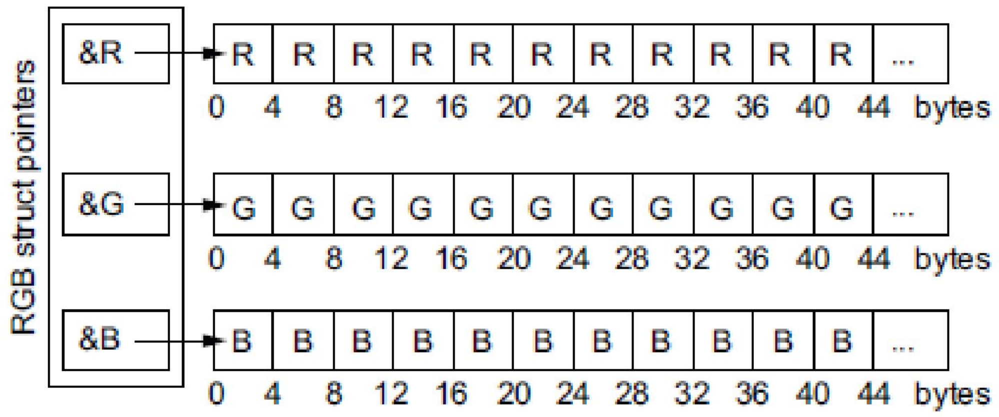
Structure of Arrays (SoA)

Listing 4.6 Structure of Arrays (SoA) in C

```
struct RGB {
    int *R;
    int *G;
    int *B;
};
struct RGB polygon_color;

polygon_color.R = (int *)malloc(1000*si[illegible]int));
polygon_color.G = (int *)malloc(1000*sizeof(int));
polygon_color.B = (int *)malloc(1000*sizeof(int));

free(polygon_color.R);
free(polygon_color.G);
free(polygon_color.B);
```

Defines an integer array of a color value

All 1,000 R values in contiguous memory. We could also use contiguous memory allocation.


RGB struct pointers
&R
&G
&B
R R R R R R R R R R R ...
G G G G G G G G G G G ...
B B B B B B B B B B B ...
0 4 8 12 16 20 24 28 32 36 40 44 bytes

AoS Performance Assessment

- If the compiler adds padding, it increases the number of memory loads.
- If only one of the RGB values is accessed in a loop, the cache usage would be poor
- When this access pattern is **vectorized** by the compiler, it would need to use a less efficient gather/scatter operation.
- Compilers try to optimize the code to better exploit vectorization (e.g. exchanging loops)

AoS in action

1. Standard compilation/execution:

```
$ gcc aos.c -o aos ; ./aos
./aos Simulation completed in 2.563 seconds.
```

2. Optimized **scalar** version (check with objdump)

```
$ gcc -fopt-info -fno-tree-vectorize -march=native -O3  aos.c -o aos-noavx ; ./aos-noavx
./aos-noavx Simulation completed in 1.276 seconds.
$ objdump -d -M intel aos-noavx | grep mul
```

AoS in action

3. Optimized **avx** version   (check with objdump)

```
$ gcc -fopt-info -mavx -O3  aos.c -o aos-avx ; ./aos-avx
[…]
aos.c:22:23: optimized: loop vectorized using 32 byte vectors
[…]
./aos-avx Simulation completed in 1.193 seconds.
$ objdump -d -M intel aos-avx | grep mul
```

AoS in action

3. Optimized **avx** version   (check with objdump)

```
$ gcc -fopt-info -mavx -O3  aos.c -o aos-avx ; ./aos-avx
[…]
aos.c:22:23: optimized: loop vectorized
[…]
./aos-avx Simulation completed in 1.193 seconds.
$ objdump -d -M intel aos-avx | grep mul
```

Is not compute bound

AoS in action

4. Optimized **avx** version with loop exchange (different code)

```
$ gcc  -fopt-info -march=native -O3  aos-le.c -o aos-le ; ./aos-le
[…]
aos-le.c:63:29: optimized: loops interchanged in loop nest
./aos-le Simulation completed in 0.614 seconds.
```

```
$ perf stat -e cache-misses ./aos-le
 4,589,102      cache-misses
$ perf stat -e cache-misses ./aos
 396,080,776      cache-misses
```

AoS in action

4. Optimized **avx** version with loop exchange (different code)

```
$ gcc  -fopt-info -march=native -O3  aos-le.
[…]
aos-le.c:63:29: optimized: loops interchanged in loop nest
./aos-le Simulation completed in 0.614 seconds.
```

gcc automatically interchange code

```
$ perf stat -e cache-misses ./aos-le
 4,589,102      cache-misses
$ perf stat -e cache-misses ./aos
 396,080,776      cache-misses
```

AoS without loop exchange

The nested loops in the original code cannot be exchanged

```
for (int step = 1; step <= STEPS; step++) {
     //the clock_gettime prevents loops interchange
        clock_gettime(CLOCK_MONOTONIC, &start); // Start timing
        updateBodies(bodies, 1.2, N);
        clock_gettime(CLOCK_MONOTONIC, &end); // End timing
        elapsed += (end.tv_sec - start.tv_sec) + (end.tv_nsec -
                  start.tv_nsec) / 1e9;
    }
```

AoS with loop exchange

The nested loops in the now can be exchanged

```
//the clock_gettime prevents loops interchange
clock_gettime(CLOCK_MONOTONIC, &start); // Start timing
for (int step = 1; step <= STEPS; step++) {
     updateBodies(bodies, 1.2, N);
}
clock_gettime(CLOCK_MONOTONIC, &end); // End timing
elapsed=(end.tv_sec - start.tv_sec)+(end.tv_nsec - start.tv_nsec)/1e9;
```

SoA in action

1. Standard compilation/execution:

```
$ gcc soa.c -o soa ; ./soa
./soa Simulation completed in 2.797 seconds.
```

2. Optimized **scalar** version   (check with objdump)

```
$ gcc -fopt-info -fno-tree-vectorize -O3  soa.c -o soa-noavx ; ./soa-noavx
./soa-noavx Simulation completed in 0.802 seconds.
$ objdump -d -M intel aos-noavx | grep mul
```

SoA in action

3. Optimized **avx** version

```
$ gcc -fopt-info -DSIMD_WIDTH=8 -O3  soa.c -o soa-avx ; ./soa-avx
soa.c:36:23: optimized: loop vectorized using 16 byte vectors
./soa-avx Simulation completed in 0.7011 seconds.
```

4. Optimized **avx2** version

```
$ gcc -fopt-info  -mavx2 -DSIMD_WIDTH=8 -O3  soa.c -o soa-avx2 ; ./soa-avx2
soa.c:36:23: optimized: loop vectorized using 32 byte vectors
soa.c:36:23: optimized:  loop versioned for vectorization because of possible aliasing
./soa-avx2 Simulation completed in 0.679 seconds.
```

SoA in action

What is this?

3. Optimized **avx** version

```
$ gcc -fopt-info -DSIMD_WIDTH=8 -O3  soa.c -o soa-avx ; ./soa-avx
soa.c:36:23: optimized: loop vectorized using 16 byte vectors
./soa-avx Simulation completed in 0.7011 seconds.
```

4. Optimized **avx2** version

```
$ gcc -fopt-info  -mavx2 -DSIMD_WIDTH=8 -O3  soa.c -o soa-avx2 ; ./soa-avx2
soa.c:36:23: optimized: loop vectorized using 32 byte vectors
soa.c:36:23: optimized:  loop versioned for vectorization because of possible aliasing
./soa-avx2 Simulation completed in 0.679 seconds.
```

SoA in action

3. Optimized **avx** version

```
$ gcc -fopt-info -DSIMD_WIDTH=8 -O3  soa.c -o soa-avx ; ./soa-avx
soa.c:36:23: optimized: loop vectorized using 16 byte vectors
./soa-avx Simulation completed in 0.7011 seconds.
```

4. Optimized **avx2** version

and this?

```
$ gcc -fopt-info  -mavx2 -DSIMD_WIDTH=8 -O3  soa
soa.c:36:23: optimized: loop vectorized using 32 byte vectors
soa.c:36:23: optimized:  loop versioned for vectorization because of possible aliasing
./soa-avx2 Simulation completed in 0.679 seconds.
```

SoA with AVX/AVX2

5. The 2 nested loops are split in 3 loops, one of them with WIDTH=SIMD_WIDTH

```
for (int step = 1; step <= STEPS; step++) {
   for (int g = 0; g < N; g +=SIMD_WIDTH) {
      for (int i = 0; i < SIMD_WIDTH; i++) {
               // Update positions
               bodies.x[g+i] = bodies.x[g+i] * 1.2;
               bodies.y[g+i] = bodies.y[g+i] * 1.2;
            }
   }
}
```

SoA in action

5. Optimized **avx2** version without aliasing (code change)

```
typedef struct {
    double* __restrict__ x, * __restrict__ y;     // Positions
    double* __restrict__ mass;                 // Masses
} Bodies;
[…]
void updateBodies(Bodies* __restrict__ b, double scaling, int n)
```

Roofline model for AoS/SoA

1. Compute theoretical values

```
ProcessorFrequency=3.8
MaxProcessorFrequency=4.9
ProcessorCores=8
Hyperthreads=2
VectorWidth=256
WordSizeBits=64
DataTransferRate=2933
MemoryChannels=2
BytesTransferredPerAccess=8

TheoreticalMemoryBandwidth=DataTransferRate*MemoryChannels*BytesTransferredPerAccess/1000
= 25.6 GiB/s


Scalar Single Core Theoretical Maximum Flops = 9.8 GFLOPs/s

Vector Single Core Theoretical Maximum Flops = 39.2 GFLOPs/s
```

Roofline model for AoS/SoA

2. Check with likwid-bench

```
$  likwid-bench -t copy -w S0:100MB
MByte/s:            24975.10

$ likwid-bench -t peakflops -W N:1MB:1
MFlops/s:           9927.61

$  likwid-bench -t peakflops_avx -W N:1MB:1
MFlops/s:           37732.46
```

Roofline model for AoS/SoA

2. Check with likwid-bench

```
$  likwid-bench -t copy -w S0:100MB
MByte/s:            24975.10

$ likwid-bench -t peakflops -W N:1MB:1
MFlops/s:           9927.61

$  likwid-bench -t peakflops_avx -W N:1MB:1
MFlops/s:           37732.46
```

theo: 25.6 GiB/s

Roofline model for AoS/SoA

2. Check with likwid-bench

```
$  likwid-bench -t copy -w S0:100MB
MByte/s:            24975.10

$ likwid-bench -t peakflops -W N:1MB:1
MFlops/s:            9927.61

$  likwid-bench -t peakflops_avx -W N:1MB:1
MFlops/s:            37732.46
```

theo: 25.6 GiB/s

theo: 9.8 GFLOPs/s

Roofline model for AoS/SoA

2. Check with likwid-bench

```
$  likwid-bench -t copy -w S0:100MB
MByte/s:              24975.10

$ likwid-bench -t peakflops -W N:1MB:1
MFlops/s:             9927.61

$  likwid-bench -t peakflops_avx -W N:1MB:1
MFlops/s:             37732.46
```

theo: 25.6 GiB/s

theo: 9.8 GFLOPs/s

theo: 39.2 GFLOPs/s

Roofline model for AoS/SoA

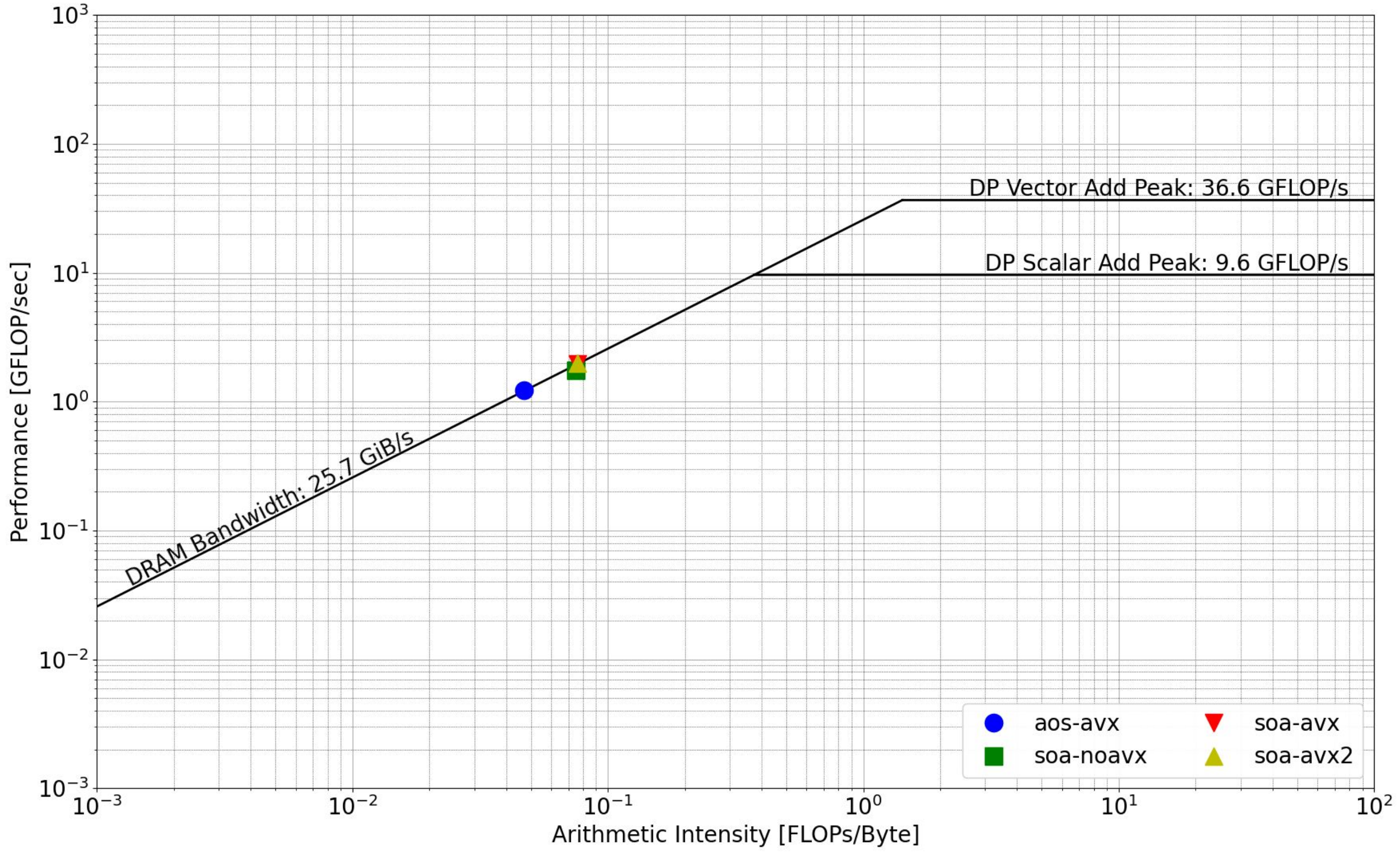
3. Get data for AoS and SoA

```
$  sudo likwid-perfctr -C 0 -g MEM_DP ./aos-avx
|    Memory bandwidth [MBytes/s]    | 26127.6211 |
|          DP [MFLOP/s]             |  1232.3386 |

$  sudo likwid-perfctr -C 0 -g MEM_DP ./soa-noavx


$  sudo likwid-perfctr -C 0 -g MEM_DP ./aos-avx


$  sudo likwid-perfctr -C 0 -g MEM_DP ./soa-avx2
```

Roofline model for AoS/SoA

4. Plot results

```
$ ./roofline.py --points "0.047,1.2330,aos-avx 0.075,1.753,soa-noavx 0.0762,1.956,soa-avx
0.0760,1990,soa-avx2"
```



Loop exchange

```
$ gcc  -fopt-info  -mavx2 -DSIMD_WIDTH=8  -O3  soa-le.c -o soa-le-avx2 ; ./soa-le-avx2
./soa-le-avx2 Simulation completed in 0.348 seconds.
```

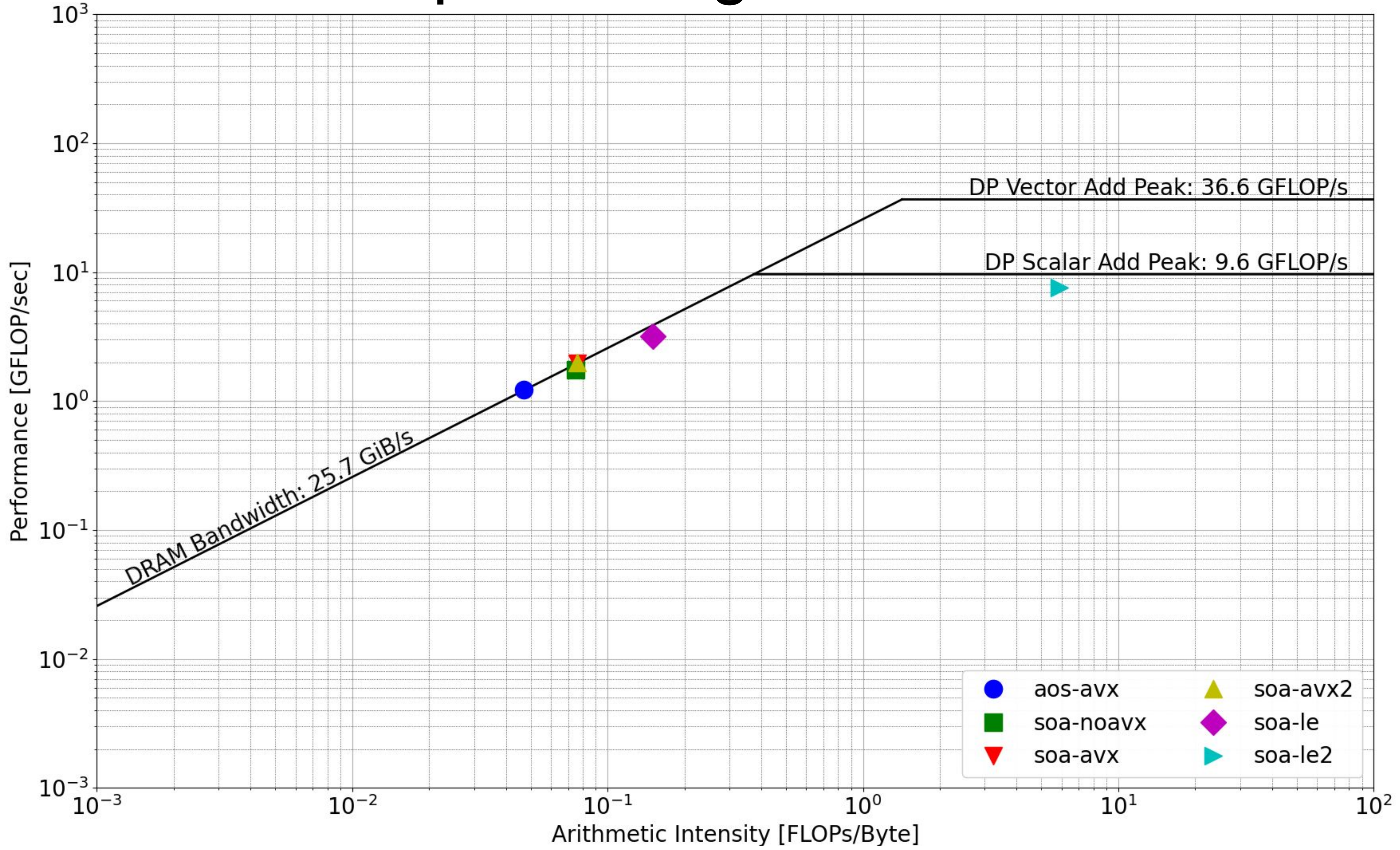
```
for (int step = 1; step <= STEPS; step++) {
  for (int g = 0; g < N; g +=SIMD_WIDTH) {
    for (int i = 0; i < SIMD_WIDTH; i++) {
         bodies.x[g+i] = bodies.x[g+i] * 1.2;
         bodies.y[g+i] = bodies.y[g+i] * 1.2;
    }
  }
}
```

Loop exchange

```
$ gcc  -fopt-info  -mavx2 -DSIMD_WIDTH=8 -O3  soa-le2.c -o soa-le2-avx2 ; ./soa-le2-avx2
./soa-le2-avx2 Simulation completed in 0.046 seconds.
```

```
for (int g = 0; g < N; g +=SIMD_WIDTH) {
 for (int step = 1; step <= STEPS; step++) {
    for (int i = 0; i < SIMD_WIDTH; i++) {
         bodies.x[g+i] = bodies.x[g+i] * 1.2;
         bodies.y[g+i] = bodies.y[g+i] * 1.2;
    }
  }
}
```

Roofline with loop exchange



Sparsity

The idea of exploiting sparsity is to avoid storing and computing on zeroes.
"The fastest way to compute is not to compute at all."

$$y = \begin{pmatrix} 5 & 0 & 0 & 0 & 0 \\ 0 & 0 & 3 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 2 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \end{pmatrix} * \begin{pmatrix} 5 \\ 4 \\ 4 \\ 6 \\ 1 \end{pmatrix}$$

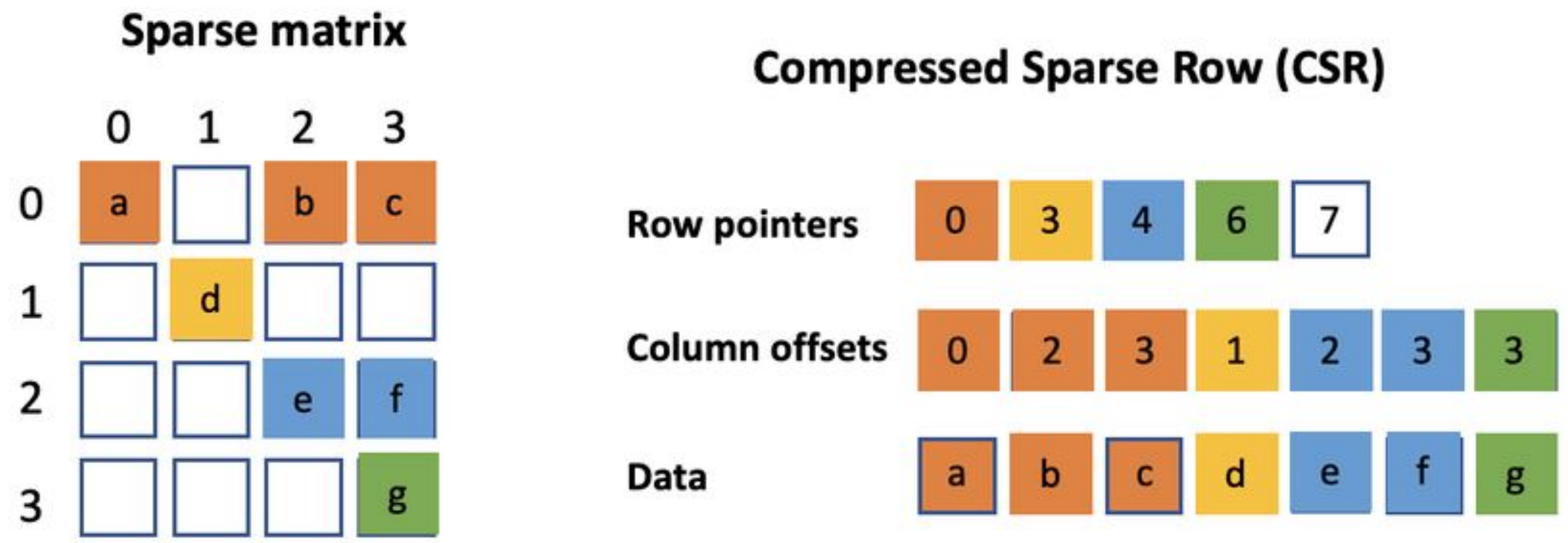
Dense matrix-vector multiplication performs $n^2 = 25$ scalar multiplies, but only 4 entries are nonzero.

Compressed Sparse Row (CSR)

A CSR matrix is represented by three arrays:

- values[]: all non-zero elements.
  - length= ***nnz***
- col_index[]: the column indices for each non-zero value.
  - length= ***nnz***
- row_ptr[]: prefix sum array of non-zero counts per row.
  - The row_ptr[] tells you **where each row starts and ends** inside the values[] and col_index[] arrays.
  - length is $\mathbf{n_r + 1}$. (#rows+1)
  - row_ptr[0] is always zero, row_ptr[n_r] is always *nnz*

Compressed Sparse Row (CSR)



CSR matrix-vector multiplication

```
void csr_matvec(
    int num_rows,
    const double *values,
    const int *col_index,
    const int *row_ptr,
    const double *x,
    double *y)
 {
    for (int i = 0; i < num_rows; i++) {
        double sum = 0.0;
        for (int j = row_ptr[i]; j < row_ptr[i + 1]; j++) {
            sum += values[j] * x[col_index[j]];
        }
        y[i] = sum;
    }
}
```

CSR matrix-vector multiplication

```
void csr_matvec(
    int num_rows,
    const double *values,
    const int *col_index,
    const int *row_ptr,
    const double *x,
    double *y)
 {
    for (int i = 0; i < num_rows; i++) {
        double sum = 0.0;
        for (int j = row_ptr[i]; j < row_ptr[i + 1]; j++) {
            sum += values[j] * x[col_index[j]];
        }
        y[i] = sum;
    }
}
```

rows can be processed in parallel

CSR matrix-vector multiplication

```
void csr_matvec(
    int num_rows,
    const double *values,
    const int *col_index,
    const int *row_ptr,
    const double *x,
    double *y)
 {
    for (int i = 0; i < num_rows; i++) {
        double sum = 0.0;
        for (int j = row_ptr[i]; j < row_ptr[i + 1]; j++) {
            sum += values[j] * x[col_index[j]];
        }
        y[i] = sum;
    }
}
```

#FLOPS =2**nnz*

Sparsity

Sparse representation can be used in different domains:

- **Linear algebra: dense** matrix-matrix multiplication is $O(N^3)$. With sparse representation we can have $O(n_{nz} \times N)$

- **Sparse Neural Networks:** in neural networks, many weights can be **zero** after pruning or quantization. The weights matrix is sparse.

- **Sparse Graph Representation:** many graph algorithms (BFS, PageRank, shortest path) use sparsity, since in a real-world network (social, web, transport), each node connects to only a few others — the adjacency matrix is mostly zeros.

Reducing DTLB Misses

- Algorithms that does random accesses into a large memory region will likely suffer from DTLB misses. Examples are:
  - binary search in a big array,
  - accessing a large hash table,
  - traversing a graph.
- **Huge pages** can speed up such applications (usually up to 30%, if TLB pressure is the primary bottleneck).

However:

- On long-running systems, physical memory becomes fragmented over time, making it difficult to allocate large contiguous blocks.
- In a multithreaded application, process threads share the virtual address space. But TLB coherence is no HW assisted (like caches). OS must send specific type **of Inter Processor Interrupt** (IPI), called ***TLB shootdown**,* which is expensive.

Explicit Huge Pages

Explicit Huge Pages (EHP) are available as part of the system memory, and are exposed as a huge page file system *hugetlbfs.*

- They are reserved before an application starts (or at boot time)
- The simplest method of using EHP in a Linux application is to call `mmap`

```
void ptr = mmap(nullptr, size, PROT_READ | PROT_WRITE,
MAP_PRIVATE | MAP_ANONYMOUS | MAP_HUGETLB, -1, 0);
```

- Developers can write their own arena-based allocators that exploit EHPs.

Transparent Huge Pages

Linux also offers **Transparent Huge Page (THP)**, which has two modes of operation:

- *system-wide*: when THP is enabled system-wide, the kernel manages huge pages automatically and it is transparent for applications.
- *per-process*: the kernel only assigns huge pages to individual processes' memory areas attributed to the `madvise` system call.

Per-process THP

`madvise(addr, length, hint)` lets a process give *hints* to the kernel about how it intends to use a certain memory region.

For THP, the hints are:

MADV_HUGEPAGE: *use Transparent Huge Pages here if possible*

MADV_NOHUGEPAGE: *Do not use Transparent Huge Pages here*

These are *advisory*: the kernel will try to follow them, but doesn’t guarantee success.

Explicit vs. Transparent Huge Pages

- **EHPs** statically sit in memory, consuming precious DRAM, even when they are not used. EHPs are not subject to memory fragmentation and cannot be swapped to disk, so they incur much less latency overhead.

- **System-wide THP** are great for quick experiments. However, it may negatively affect other running programs.

- The allocation process of **per-process THP** can potentially involve several kernel processes responsible for making space in virtual memory, which may include swapping memory to disk, fragmentation, or promoting pages. This creates non-deterministic latency overhead.

Software Memory Prefetching

Modern CPUs try to lower the penalty of cache misses by predicting which memory locations will access in the future and prefetch them ahead of time. They use 2 automatic methods:

1. **OOO execution** looks **N** instructions into the future and issues loads early.
2. **Hardware prefetch** anticipates requests on repetitive memory access patterns. They fail when data access patterns are too complicated to predict.

```
for (int i = 0; i < N; ++i) {
  size_t idx = random_distribution(generator);
   int x = arr[idx]; // cache miss
  doSomeExtensiveComputation(x); // too far for the OOO engine
}
```

Software Memory Prefetching

```
size_t idx = random_distribution(generator);
for (int i = 0; i < N; ++i) {
  int x = arr[idx];
  idx = random_distribution(generator); // compute next index
  __builtin_prefetch(&arr[idx]);        // prefetch next element
  doSomeExtensiveComputation(x);        // compute current element
}
```

This code issues the memory request early since it already know the next index in the array. This transformation makes our prefetching window much larger and fully hides the latency of a cache miss.

Software Memory Prefetching

- A prefetch hint must be inserted well ahead of time so that will be in the cache when requested.
- Shouldn't be inserted too early to avoid cache pollution.
- Setting the PREFETCH distance can be hard to analyze

```
process_burst(struct buf **pkts, uint16_t nb_pkts)
{
    uint16_t i;

    for (i = 0; i < nb_pkts && i < PREFETCH_OFFSET;
  i++)
        __builtin_prefetch(pkts[i]);

    for (i = 0; i < nb_pkts - PREFETCH_OFFSET; i++) {
        __builtin_prefetch(pkts[i+ PREFETCH_OFFSET]);
     process_packet(pkts[i]);
    }

    /* Process remaining prefetched packets */
    for (; i < nb_pkts; i++)
        process_packet(pkts[i]);
}
```