Basic C Programming

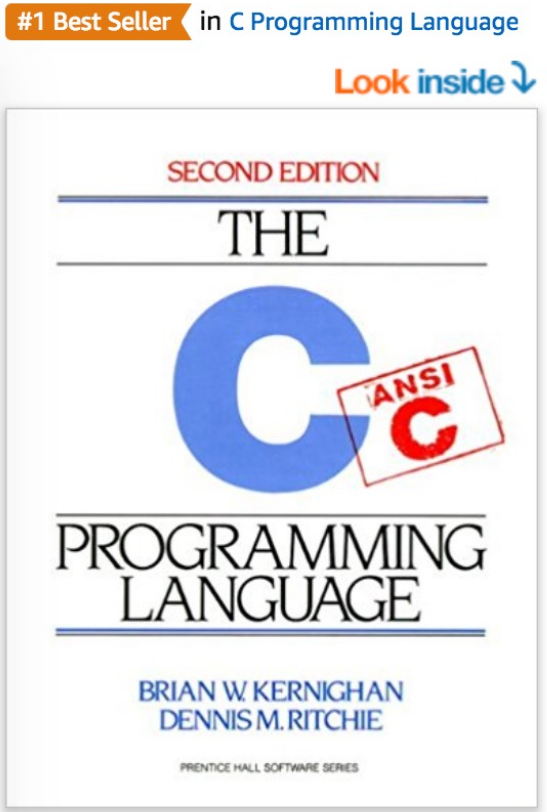
ITSC 3181 Introduction to Computer Architecture
https://passlab.github.io/ITSC3181/

Department of Computer Science
Yonghong Yan
yyan7@uncc.edu
https://passlab.github.io/yanyh/

C Programming Basics: Outline

- A crash course in the basics of C
  - Overview comparison of C and Java
  - Good evening
  - Preprocessor
  - Command line arguments
  - Arrays and structures
  - Pointers and dynamic memory

- The K&R C book for lots more details
  - Tons of info on web
- https://passlab.github.io/ITSC3181/resources/C_Programming.pdf



ISBN-13: 978-0131103627
ISBN-10: 0131103628

main Function to Enter Execution

```
#include <stdio.h>
int main(int argc, char* argv[])
{
  /* print a greeting */
  printf("Good evening!\n");
  return 0;
}
```

```
$ ./goodevening
Good evening!
$
```

Breaking down the code

- `#include <stdio.h>` ➔ `java import`
  - Include the contents of the file stdio.h
    - Case sensitive – lower case only
  - No semicolon at the end of line

- `int main(…)`
  - The OS calls this function when the program starts running.

- `printf(format_string, arg1, …)`
  - Prints out a string, specified by the format string and the arguments.

Command Line Arguments

- `int main(int argc, char* argv[])`
- **`argc`**
  - Number of arguments (including program name)
- **`argv`**
  - Array of char*s (that is, an array of ‘c’ strings)
  - **`argv[0]`**: = program name
  - **`argv[1]`**: = first argument
  - **…**
  - **`argv[argc-1]`**: last argument

Like Java, like C, and Lots of Other Languages

1. Operators same as Java ➔ forming programming expressions and basic statement for **calculations/operations**
   - Arithmetic
     - `i = i+1; i++; i--; i *= 2;`
     - `+, -, *, /, %,`
   - Relational and Logical
     - `<, >, <=, >=, ==, !=`
     - `&&, ||, &, |, !`

2. Syntax same as in Java ➔ **structured program statement**
   - `if ( ) { } else { }`
   - `while ( ) { }`
   - `do { } while ( );`
   - `for(i=1; i <= 100; i++) { }`
   - `switch ( ) {case 1: … }`
   - `continue; break;`

Data Types

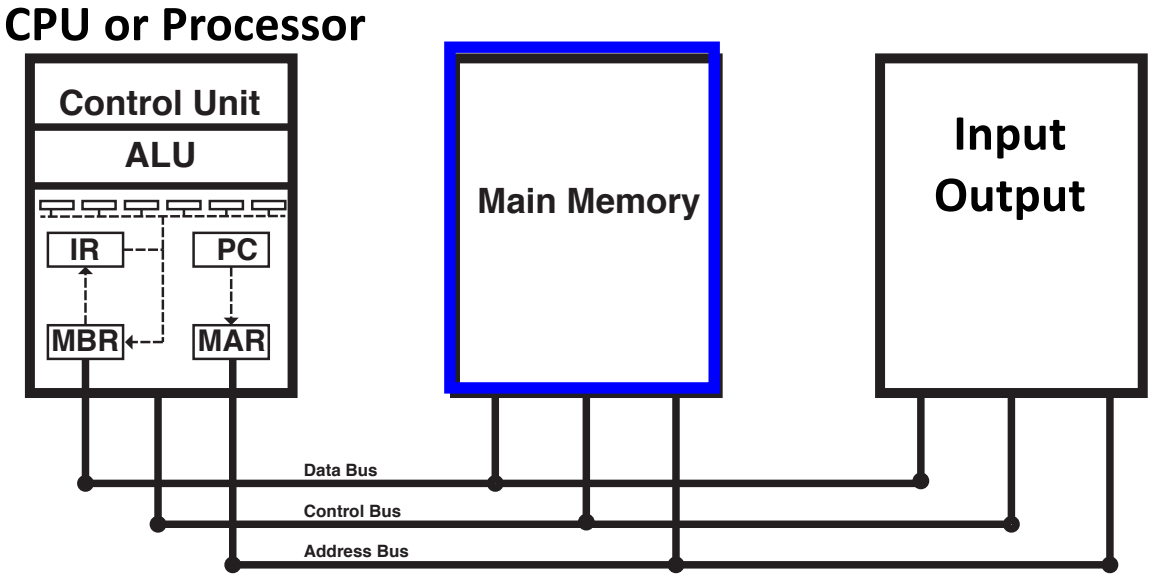
- Simple data types

datatype	size (byte)	values
char	1	-128 to 127
short	2	-32,768 to 32,767
int	4	-2,147,483,648 to 2,147,483,647
long	4	-2,147,483,648 to 2,147,483,647
float	4	3.4E+/-38 (7 digits)
double	8	1.7E+/-308 (15 digits long)

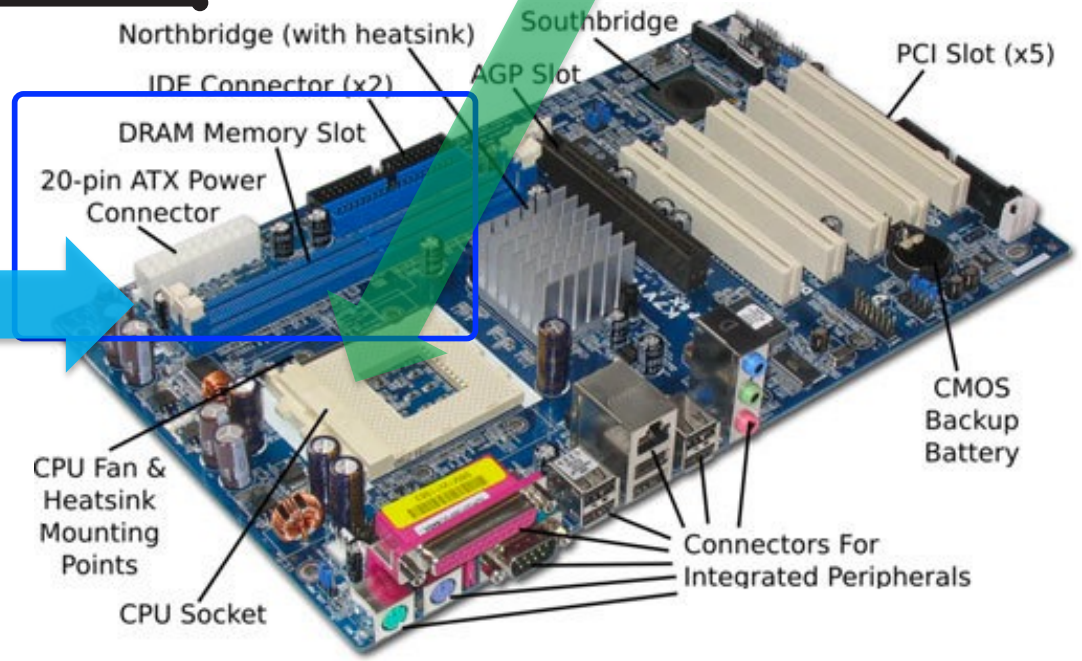
- Complex data types
  - Array: int A[100];
  - struct ~= class

- **Declare a variable: symbol and type. E.g. int a**
  - **Type indicate size**
  - **Symbol: A human-understandable name for a memory location**

Main Memory (DRAM) of a Computer

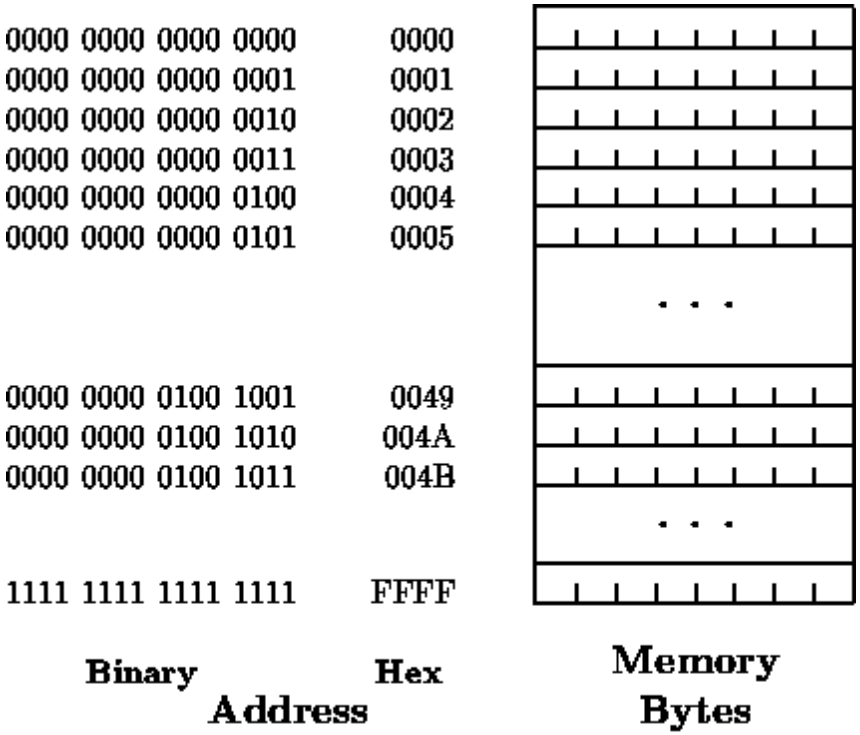


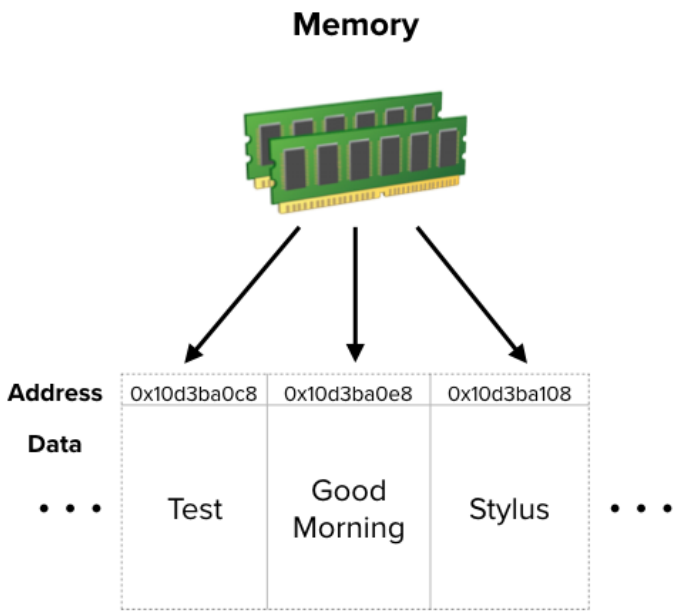
CPU is also called a chip.



Memory and Address

- Memory are accessed via the address of memory cells that store data
  - int a = A[i];
    - Read value from a memory location whose address is represented by A[i];
    - Write value to a memory location whose address is represented by a





Variables ← →Memory Locations

Compiler maps variable → memory location.

Declarations do not initialize!

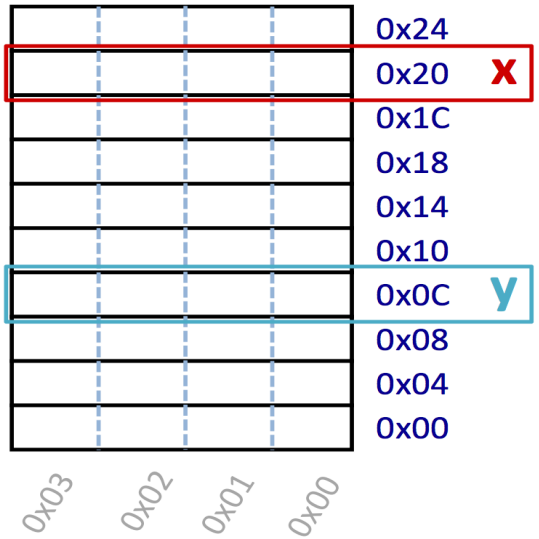
int is a 4-byte data type.

```
int x; // x at 0x20
int y; // y at 0x0C

x = 0; // store 0 at 0x20

// store 0x3CD02700 at 0x0C
y = 0x3CD02700;

// load the contents at 0x0C,
// add 3, and store sum at 0x20
x = y + 3;
```



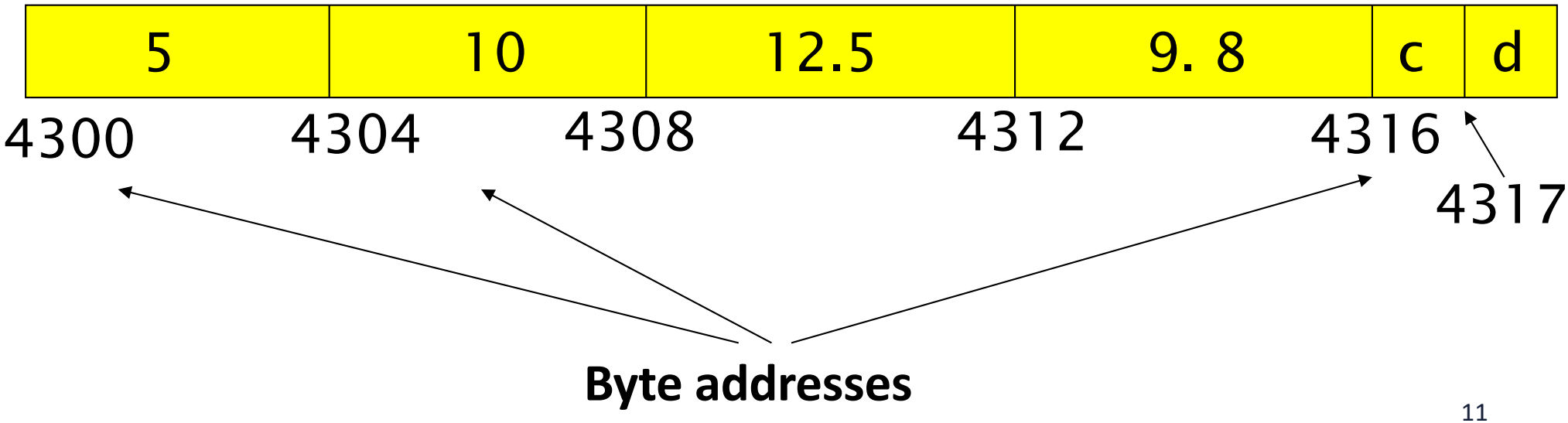
- Variable (x) is symbolic representation of a memory location/addres
- Two types of access to a variable/memory location: Read or write
  - = x: Right value, i.e. appears on the right side of =
    - read/load the content from the memory location
  - x =: Left value, i.e. appears on the left side of =
    - Write a value to the memory location

Memory layout and addresses

```
int x = 5, y = 10;
float f = 12.5, g = 9.8;
char c = 'c', d = 'd';
```

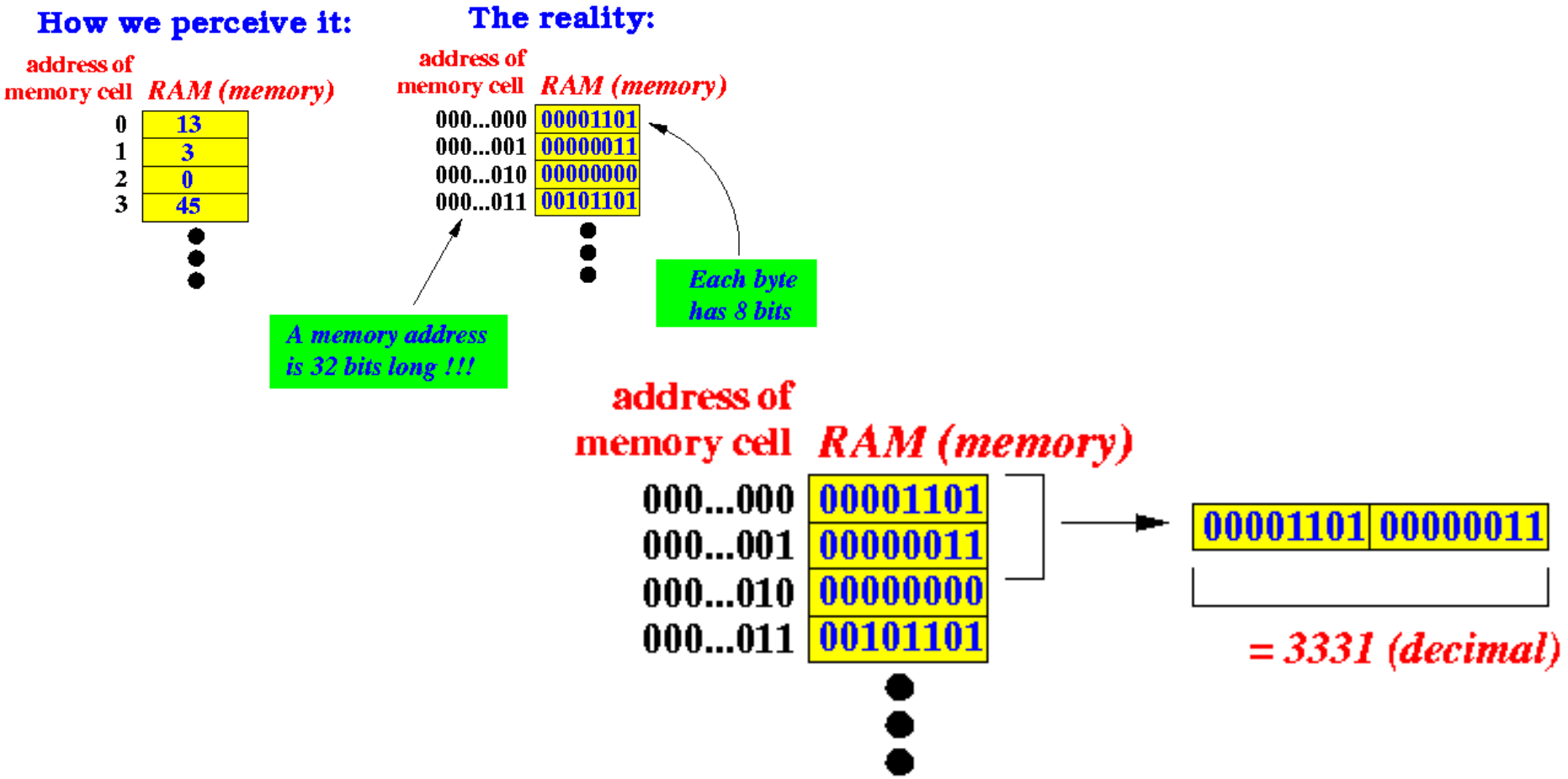
Sizes of data types

int: 4 bytes
float: 4 bytes
char: 1 byte
double: 8 bytes
long: 8 bytes



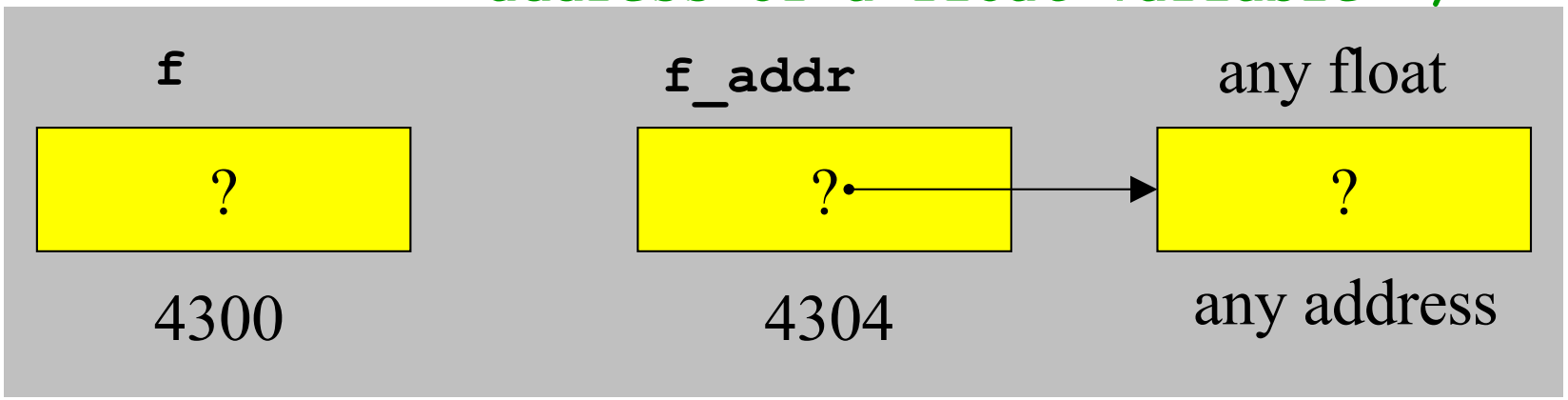
Pointers

- Pointers are variables that hold an address in memory.
- That address *points* to another variable.

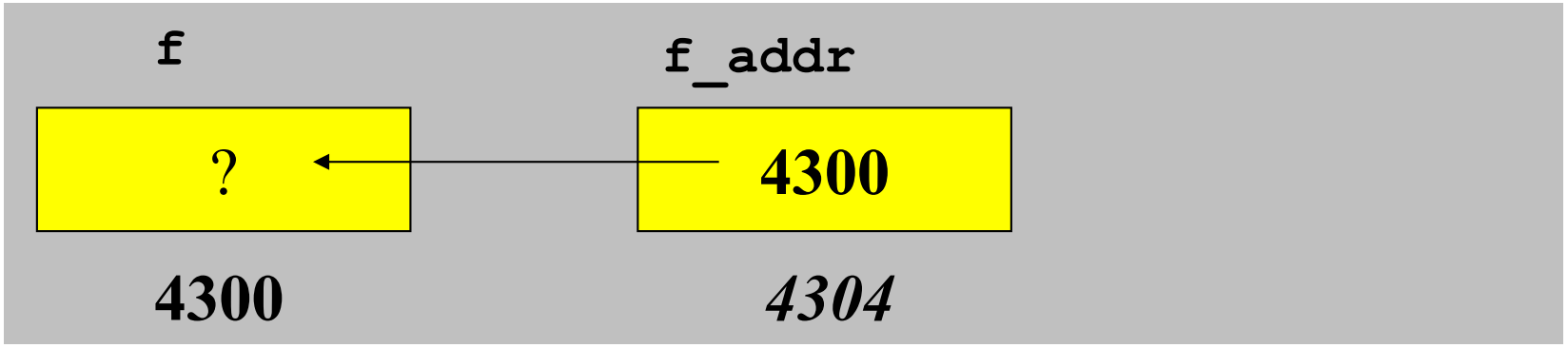


Using Pointers (1)

```
float f;          /* data variable */
float *f_addr;    /* pointer variable: store the memory
                   * address of a float variable */
```

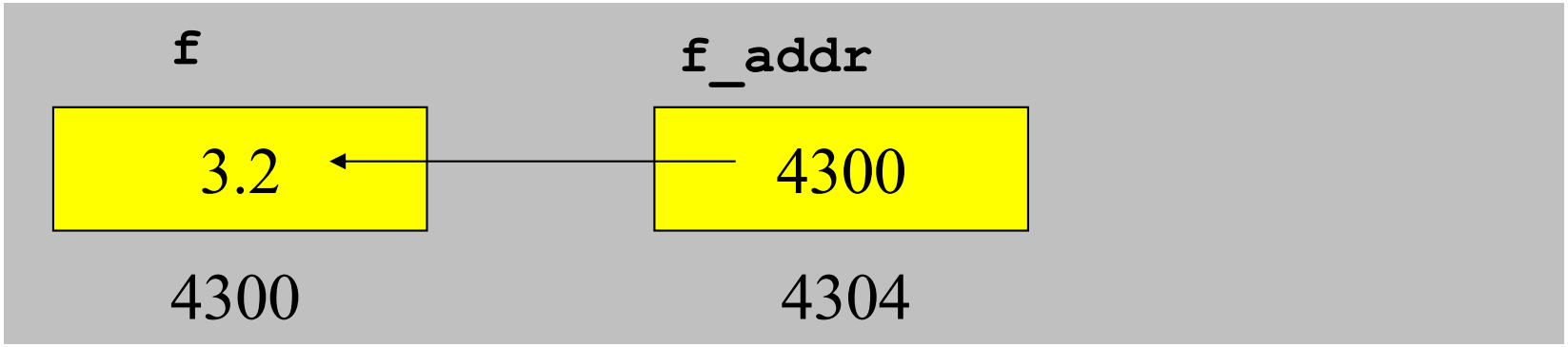



```
f_addr = &f;      /* & = operator to get the address */
```

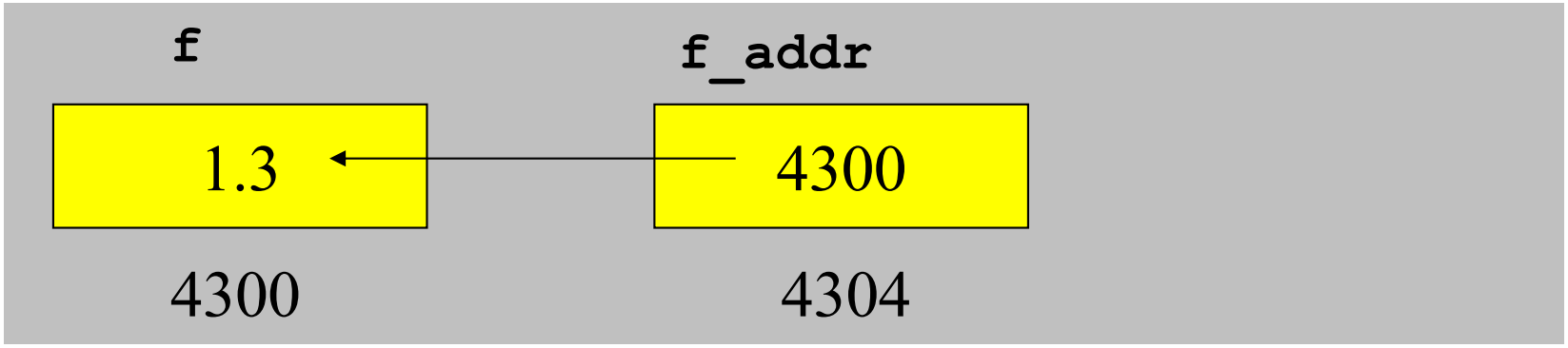


Pointers made easy (2)

```
*f_addr = 3.2; /* indirection operator: assign the value
                  3.2 to the memory cell at address f_addr */
```




```
float g = *f_addr;/* indirection: read value at memory
                     address f_addr, g is now 3.2 */
f = 1.3;     /* but g is still 3.2 */
```



A Variable from CA Point of View

- int a;
- float f;
- int * ap;
- float *fp
- char * str;
- char * argv[];
- A variable
  - Name of the variable is the symbolic representation of the memory address for the first byte of the memory location allocated for the variable
  - Type: size of the memory for the variable
    - char: 1 byte, int/float/long: 4 bytes; double: 8 bytes
    - char *, int *, float *, double *, void *: 4 or 8 bytes depends on whether it is a 32 or 64-bit system
  - Variable reference == address reference
    - On the left of =: load the value of an address, type is used to determine how many bytes to load
    - On the right of =: store a value to the address, type is used to determine how many bytes to store
  - &x = address of x
  - *p = content at address p

C Variable and Pointer

& = address of

*** = contents at**

Declare a variable, p

```
int* p;
```

that will hold the address of a memory location holding an int

```
int x = 5;
int y = 2;
```

Declare two variables, x and y, that hold ints, and store 5 and 2 in them, respectively.

Get the address of the memory location representing x

```
p = &x;
```

... and store it in p. Now, "*p points to x.*"

Add 1 to the contents of memory at the address stored in p

```
y = 1 + *p;
```

... and store it in the memory location representing y.

C Pointer and Memory

C assignment:

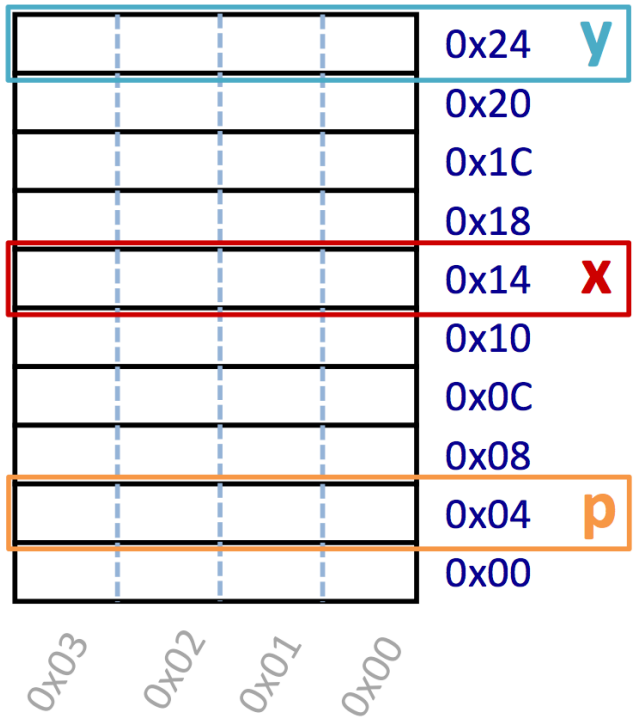
Left-hand-side = right-hand-side;

location value

& = address of

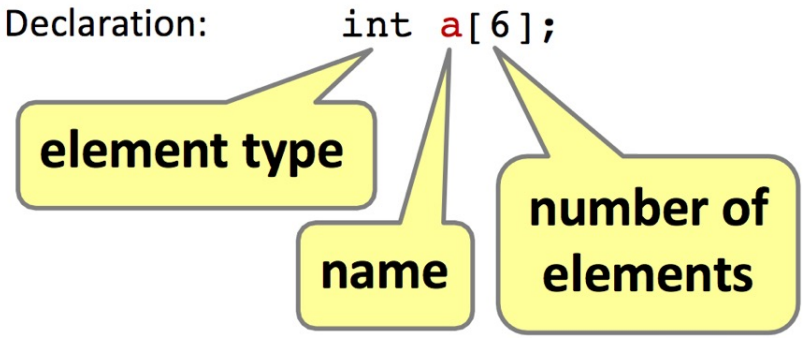
*** = contents at**

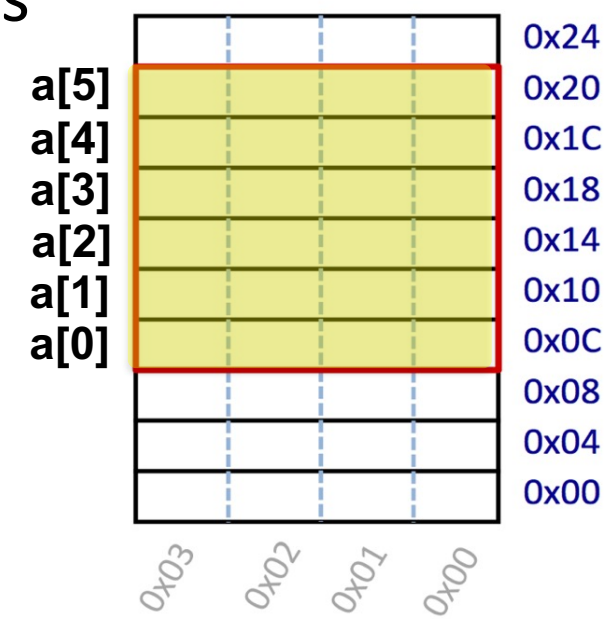
```
int* p;      // p: 0x04
int x = 5;   // x: 0x14, store 5 at 0x14
int y = 2;   // y: 0x24, store 2 at 0x24
p = &x;      // store 0x14 at 0x04
// load the contents at 0x04  (0x14)
// load the contents at 0x14  (0x5)
// add 1 and store sum at 0x24
y = 1 + *p;
// load the contents at 0x04  (0x14)
// store 0xF0 (240) at 0x14
*p = 240;
```



Arrays

- **Adjacent memory locations storing the same type of data**
  - **Elements are packed in memory space**
- `int a[6];` means space for six integers
  - Each int is 4 bytes

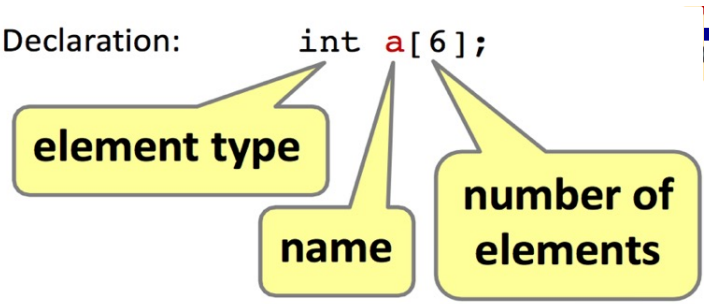


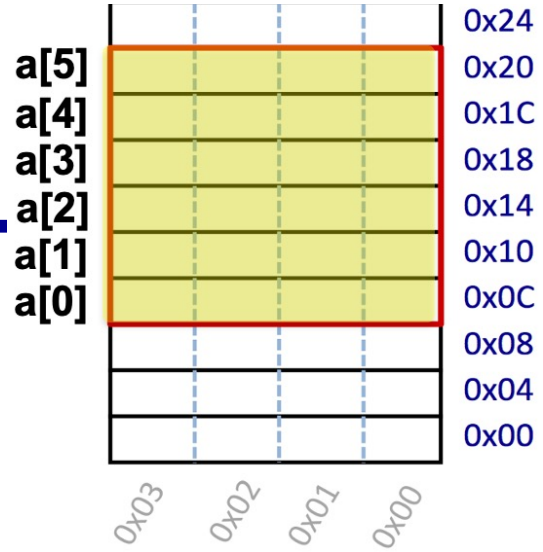


- a is the symbol (variable) representing the array's base address, which is the address of element a[0] as well.
  - 0x0C

Address of Array Elements

- `int a[6];`



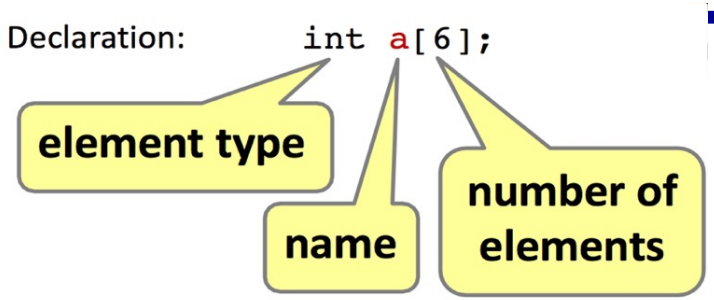


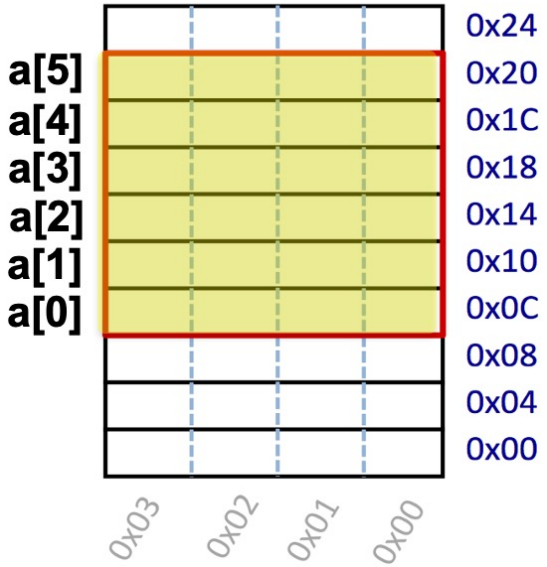
- **Offset of a[i]:** stride (number of bytes) between a[0] and a[i]
  - i*sizeof(int)
- **Byte address of a[i] (&a[i]): base + offset**

  &a[i]: (char*)a + i * sizeof(int)
  - E.g. &a[2]: 0x0C + 2 * 4 = 0x14
  - (char*)a is a cast of (int*) to (char*), to make sure compiler recognizes it as a byte address so it can add up i*sizeof(int)
  - **In C, &a[i] is also a+i since C compiler is able to scale the pointer arithmetic with the size of the data type of the array**
    - **Thus &a[i]: a + i, this is pointer arithmetic, not regular arithme**
- By itself, a is also the address of the first integer
  - *a and a[0] mean the same thing

Address of Array Elements

- `int a[6];`





- Offset of a[i] from a[j]: stride (number of bytes) from a[j] to a[i]
  - (i-j)*sizeof(int)

&a[i]: (char*)&a[j] + (i-j) * sizeof(int), or
&a[j] + i-j

  - Example, given &a[3] is 0x18, what is &a[5]

&a[5]: (char*)&a[3] + (5-3) * sizeof(int), or
&a[3] + 5-3

  - Example, given &a[4] is 0x1c, what is &a[2]

&a[2]: (char*)&a[4] + (2-4) * sizeof(int), or
&a[4] + 2-4

C: Arrays

Arrays are adjacent memory locations storing the same type of data.

a is a name for the array's base address, can be used as an *immutable* pointer.

Address of **a[i]** is base address **a** plus **i** times element size in bytes.

Declaration:

```
int a[6];
```

Indexing:

```
a[0] = 0xf0;
a[5] = a[0];
```

No bounds check:

```
a[6] = 0xBAD;
a[-1] = 0xBAD;
```

Pointers:

```
int* p;
```

equivalent

```
p = a;
p = &a[0];
```

```
*p = 0xA;
```

equivalent

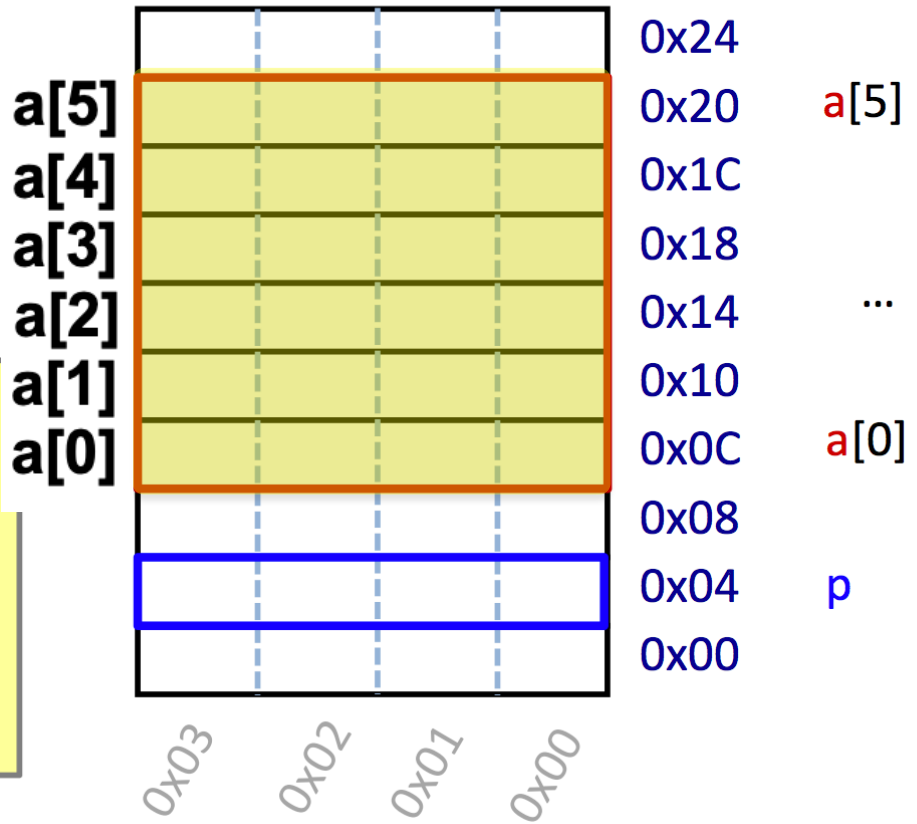
```
p[1] = 0xB;
*(p + 1) = 0xB;
```

```
p = p + 2;
```

array indexing = address arithmetic

Both are scaled by the size of the type.

```
*p = a[1] + 1;
```



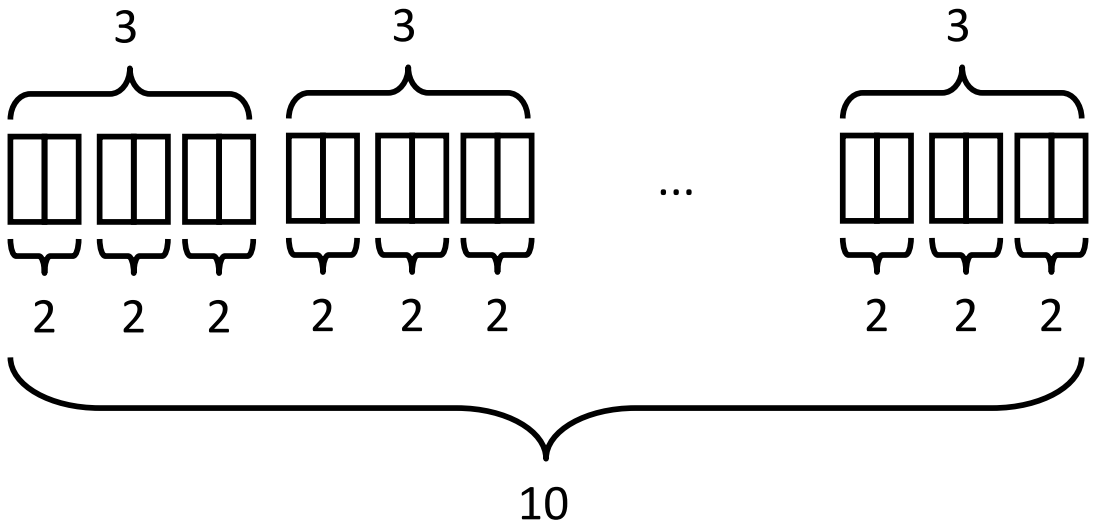
sizeof Arrays

- `int a[6];`
  - `sizeof(a)`
    = 6 × `sizeof(int)`
    = 6 × 4 = 24 bytes

- `char foo[80];`
  - An array of 80 characters
  - `sizeof(foo)`
    = 80 × `sizeof(char)`
    = 80 × 1 = 80 bytes

Multidimensional Arrays

- Array declarations read right-to-left
- int a[10][3][2];
- “an array of ten arrays of three arrays of two ints”
- In memory

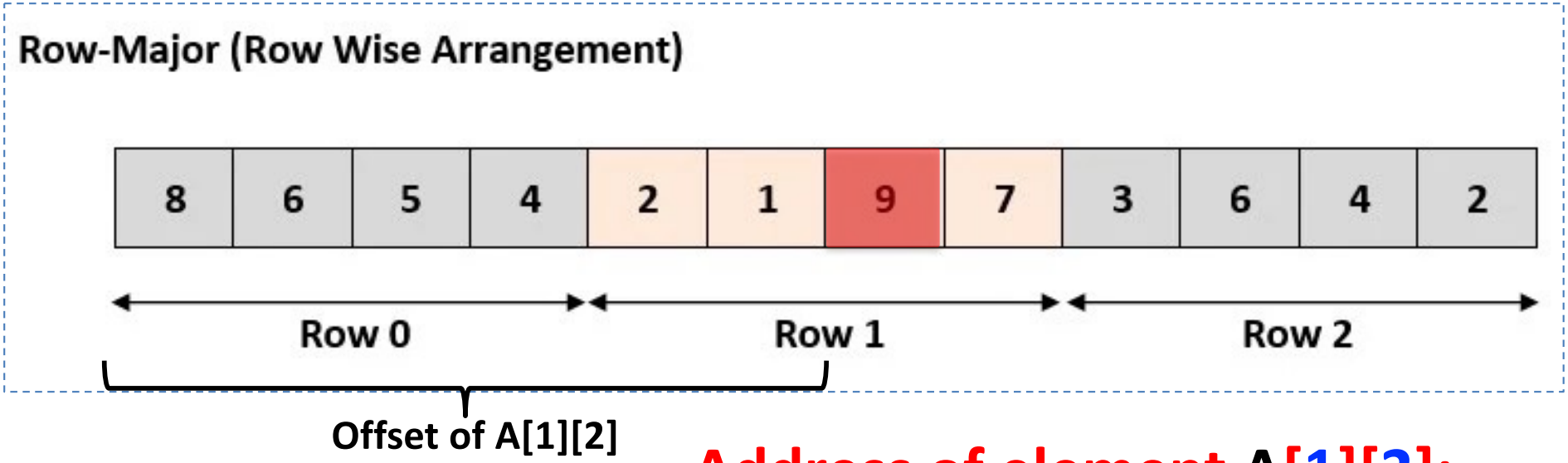


Seagram Building, Ludwig Mies van der Rohe,1957

C Stores Array in Memory in Row Major

int A[3][4];

8	**6**	**5**	**4**
2	1	9	7
3	6	4	2



Address of element A[1][2]:

= (char*) A + offset (from A to A[1][2])

= (char*) A + sizeof (int) * (1 * 4 + 2)

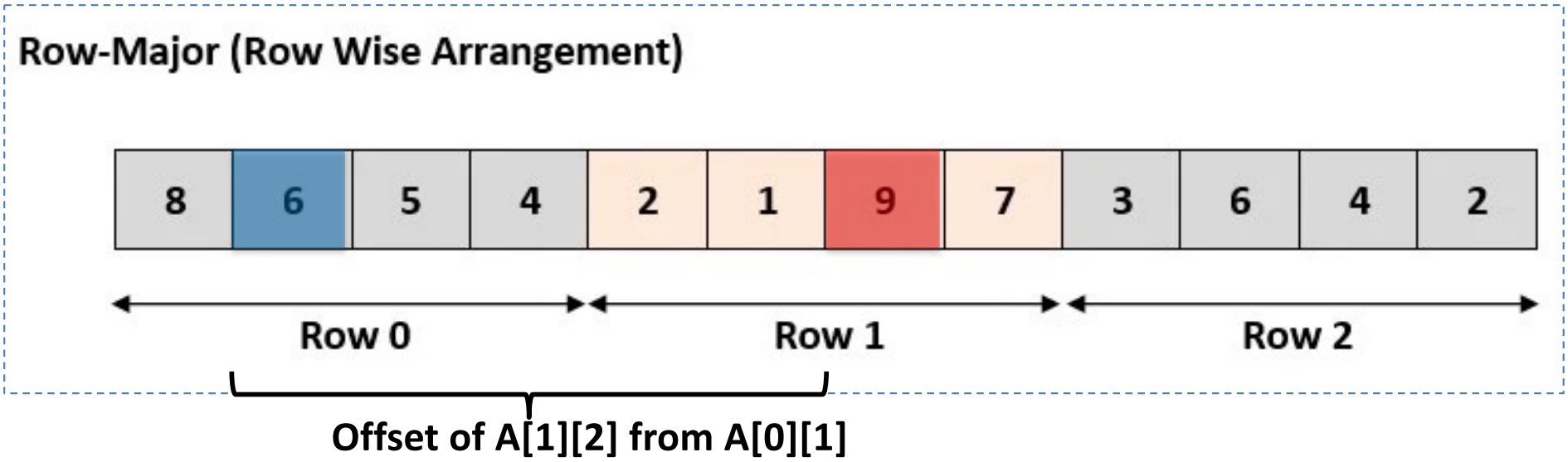
= (char*) A + 4 * 6 = (char *)A + 24

Offset of A[i][j] from A[of an array A[M][N]:
i*N + j

C Stores Array in Memory in Row Major

int A[3][4];

8	6	5	4
2	1	9	7
3	6	4	2



Given the address of A[0][1], find the address of element A[1][2]:

= (char*) A[0][1] + offset (from A[0][1] to A[1][2])

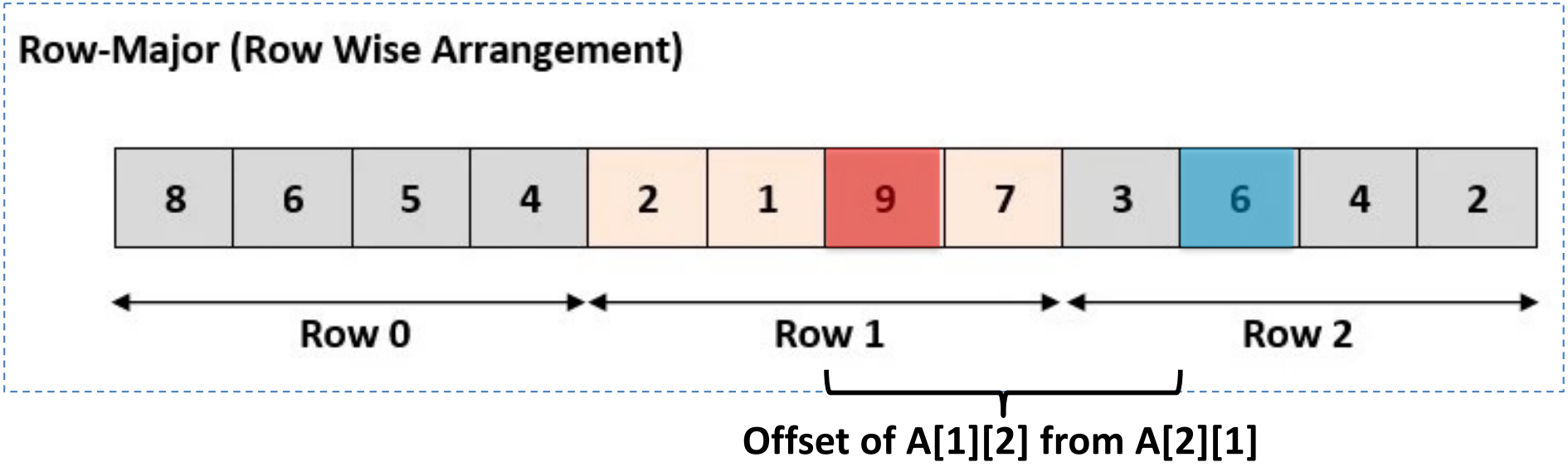
= (char*) A[0][1] + sizeof (int) * ((1-0) * 4 + 2-1)

= (char*) A[0][1] + 4 * 5 = (char*) A[0][1] + 20

C Stores Array in Memory in Row Major

int A[3][4];

8	6	5	4
2	1	9	7
3	6	4	2



Given the address of A[2][1], find the address of element A[1][2]:

= (char*) A[2][1] + offset (from A[2][1] to A[1][2])

= (char*) A[2][1] + sizeof (int) * ((1-2) * 4 + 2-1)

= (char*) A[2][1] + 4 * -3 = (char*) A[2][1] - 12

Structures

- Similar to Java class, but no methods

```
#include <stdio.h>

struct person {
   char*      name;
   int        age;
}; /* <== DO NOT FORGET the semicolon */
int main(int argc, char* argv[])
{
   struct person bovik;
   bovik.name = "Harry Bovik";
   bovik.age = 25;

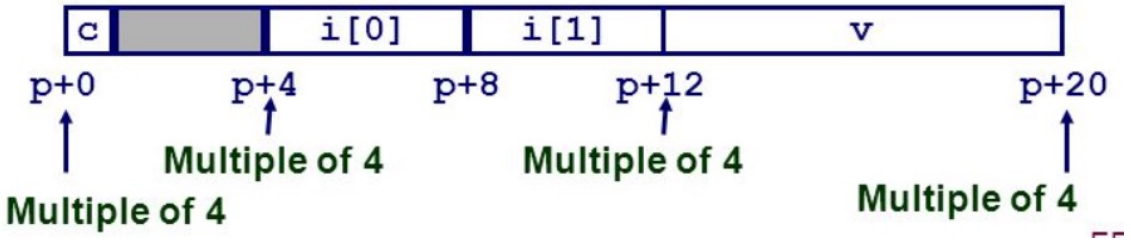
   printf("%s is %d years old\n", bovik.name, bovik.age);
   return 0;
}
```

Address of Fields of Struct Object

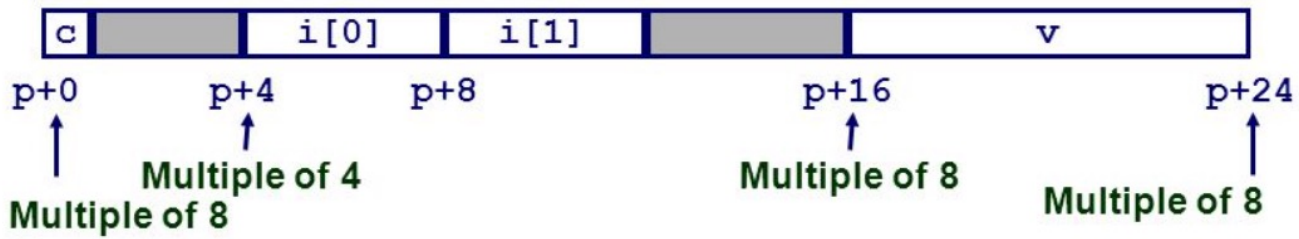
- Similar to array that pack struct fields togethe
  - Complicated because of alignment
    - Char: 1 byte, int: 4 bytes, double: 8 bytes

```
struct S1 {
  char c;
  int i[2];
  double v;
} *p;
```

  - 4-byte alignment



  - 8-byte alignment (one way)



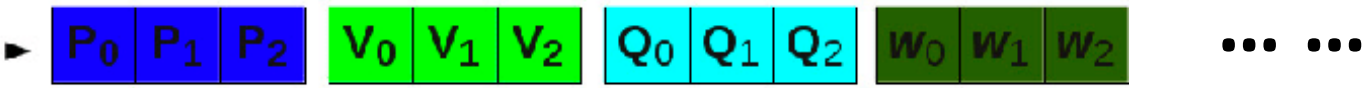
Extend to Array of Structs and Struct of Arrays

```
1   struct astruct  {
2       int P, V, Q, W;
3   };
4   struct astruct anArrayOfStruct [100];
5
6   struct aStructOfArrayStruct {
7     int P[100];
8     int V[100];
9     int Q[100];
10    int W[100];
11  };
12  struct aStructOfArrayStruct aStructOfArray;
```

Memory Layout

Array of Structs AOS

Struct of Arrays SOA

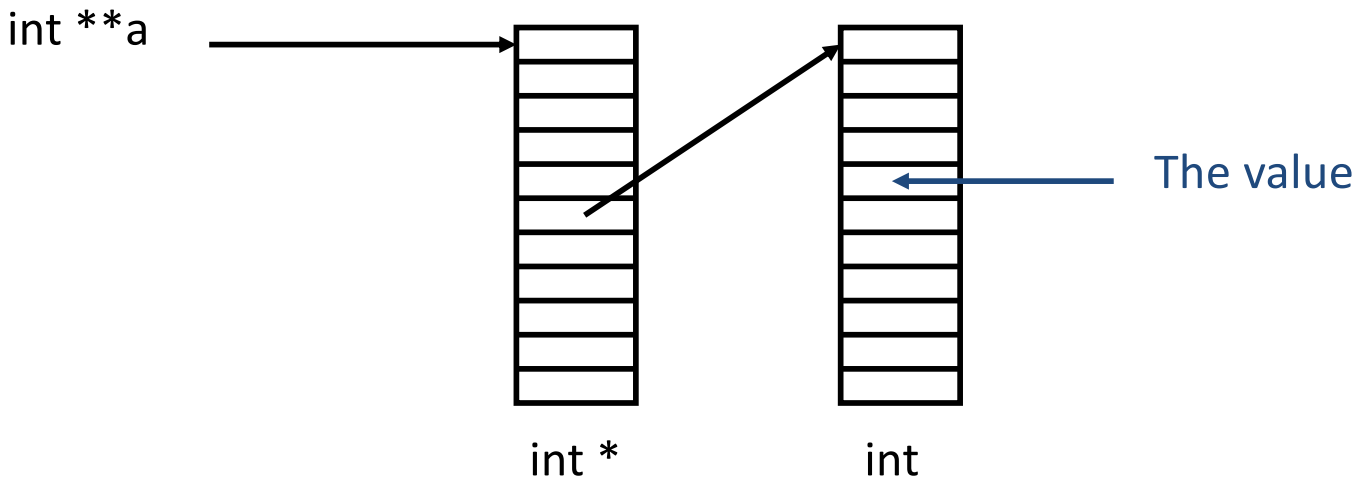


Multidimensional Arrays the Java Way

- Use arrays of pointers for variable-sized multidimensional arrays
  - Java's approach for multi-dimensional array
  - Need to allocate space for and initialize the arrays of pointers

```
int **a;
```

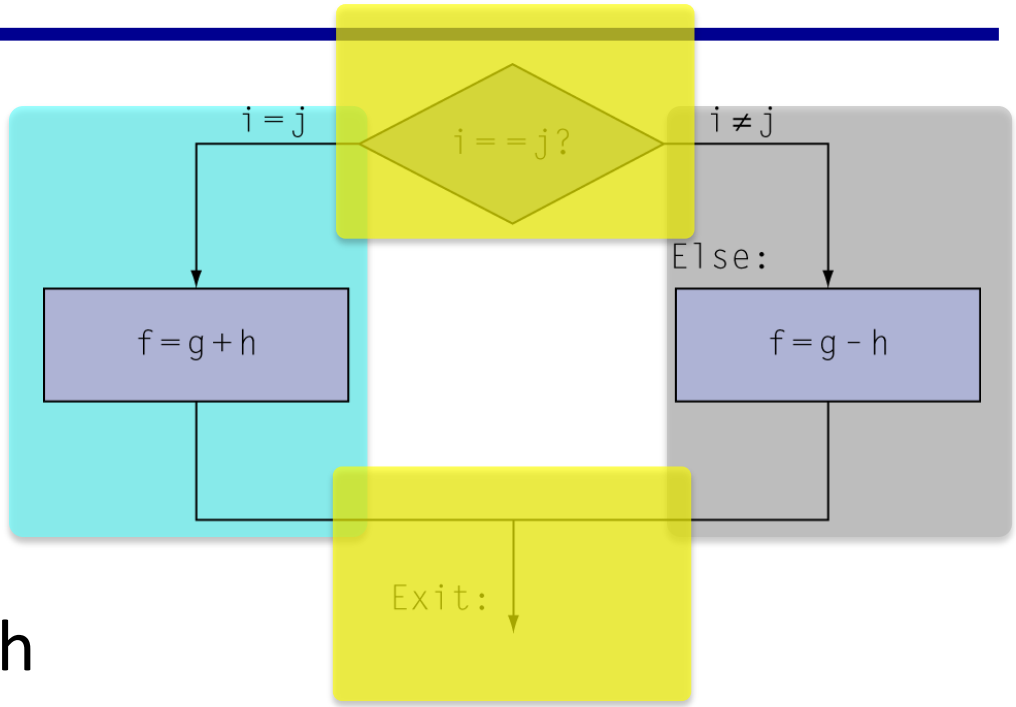
`a[5][4]` expands to `*(*(a+5)+4)`



If-else Statements

- C code:

```
if (i==j) f = g+h;
else f = g-h;
```

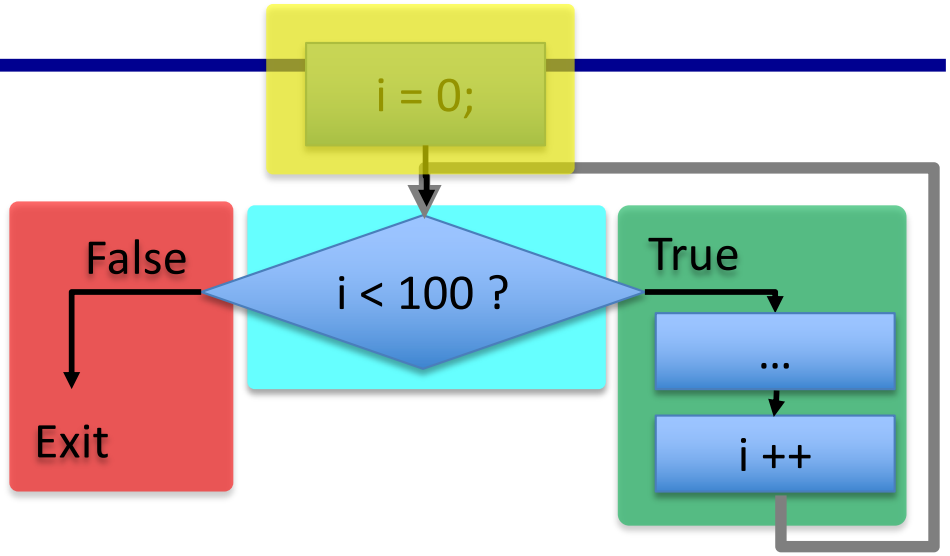


- Condition check i==j?
- Arithmetic operation: g+h, g-h

Loop Statement

```
for (i=0; i<100; i++) { … }

while (i<100) { …; i++; }
```

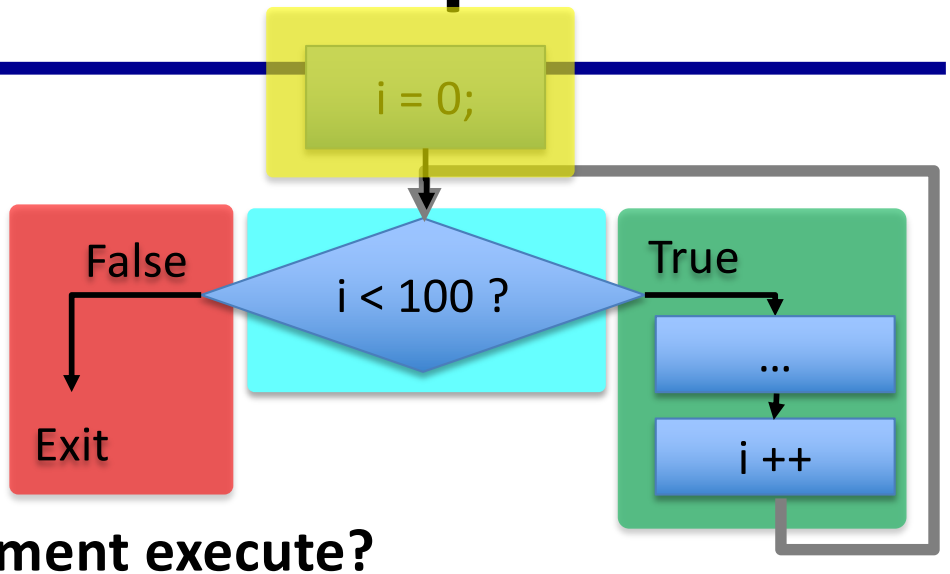


- Loop execution:
  - Init condition
  - Loop condition check
  - True path (the loop body)
  - Loop back
  - False path (break the loop)

Loop Statement: for loop

- C code:

  for (i=0; i<100; i++) …



How many times does each each statement execute?

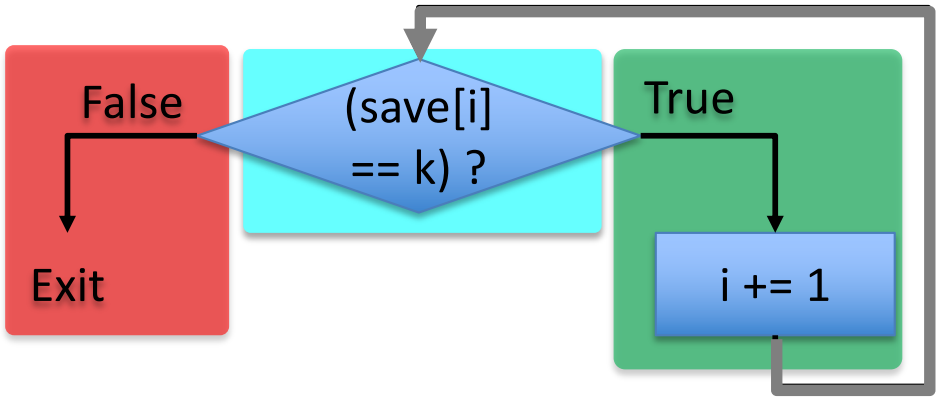
- i=0:
  - only executes once
- i<100:
  - execute 100 times
- i++:
  - execute 100 times
- …:
  - execute 100 times (if they don't modify i)

Loop Statement: while loop (textbook 2.7)

- C code:

```
int save[100];

while (save[i] == k) i += 1;
```



How many times does each each statement execute?

- save[i] == k:
  - check every iteration
- i+1=1:
  - execute every iteration
- What is the problem of this code?

Counting Operations of A C Program

```
float sum(int N, float X[], float a) {
    int i;
    float result = 0.0;
    for (i = 0; i < N; ++i) {
        result += a * X[i];
    }
    return result;
}
```

- N = 1000

- **Arithmetic and logic operation:**
  - i++, +, *: three operations per iteration ➔ 3000
- **Array reference**
  - X[i]: one array reference per iteration ➔ 1000
- **Condition check (if-else, loop)**
  - i < N: one check per iteration ➔ 1000

Counting Operations of A C Program

```
int N = 1000;
int A[N][N], B[N][N], C[N][N]
int i, j;
for (i=0; i < N; i++) {
   for(j=0; j < N; j++) {
     A[i][j] += B[i][j] * C[N][N] * C[N][N];
  }
}
```

- **Arithmetic and logic operation:**
  - j++, +, *, *: four operations per iteration of inner loop ➔ 4M
  - i++: one operation per outer loop iteration ➔ 1000
- **Array reference**
  - A[i][j], B[i][j], C[i][j], C[i][j], A[i][j]: 5 array reference per iteration of inner loop ➔ 5M
- **Condition check (if-else, loop)**
  - j < N: one check per iteration of inner loop ➔ 1M
  - i<N: one check per iteration of outer loop ➔ 1000

Counting Operations of A C Program

```
int N = 1000, i;
int A[N], B[N], C[i]
for (i=0; i < N; i++) {
    if (B[i] != 0) A[i] += B[i] * C[i] + C[i];
    else A[i] += C[i]; //Else
}
```

- **40% of `B[N]` elements are NOT zeros**
  - `if (B[i] != 0):` executed for each iteration
  - `True: A[i] += B[i] * C[i] + C[i];` executed 40% of total iteration
  - `False: A[i] += C[i];` executed 60% of total iteration
- **Arithmetic and logic operation:**
  - i++: one operation for each iteration ➔ 1000
  - True: +, *, +: three operations of 40% of total iterations ➔ 3*1000 * 40% = 1200
  - False: +: one operation of 60% of total iterations ➔ 1*1000 * 60% = 600
  - Thus in total: 2800
- **Array reference**
  - If (B[i] …): one per iteration ➔ 1000
  - True: A[i], B[i], C[i], C[i], A[i]: 5 of 40% of total iterations ➔ 5*1000*40% = 2000
  - False: A[i], C[i], B[i]: 3 of 60% of total iterations ➔ 3*1000*60% = 1800
  - Thus total: 4800
- **Condition check (if-else, loop)**
  - i<N: one check per iteration ➔ 1000
  - If (B[i] …): one check per iteration ➔ 1000
  - Total: 2000

End of Introduction of C Basics

Additional Topics for C Programming

- C Preprocessing
- Dynamic memory
- Function parameters
  - Pass by value
  - Pass a pointer

C Preprocessor

```
#define FIFTEEN_TWO_THIRTEEN \
  "The Class That Gives CMU Its Zip\n"

int main(int argc, char* argv[])
{
  printf(FIFTEEN_TWO_THIRTEEN);
  return 0;
}
```

After the preprocessor (gcc -E)

```
int main(int argc, char* argv)
{
  printf("The Class That Gives CMU Its Zip\n");
  return 0;
}
```

Conditional Compilation

```
#define CSCE212

int main(int argc, char* argv)
{
  #ifdef CSCE212
  printf("The Class That Gives CMU Its Zip\n");
  #else
  printf("Some other class\n");
  #endif
  return 0;
}
```

After the preprocessor (gcc –E)

```
int main(int argc, char* argv)
{
  printf("The Class That Gives CMU Its Zip\n");
  return 0;
}
```

Dynamic Memory

- Java manages memory for you, C does not
  - C requires the programmer to *explicitly* allocate and deallocate memory
  - Unknown amounts of memory can be allocated dynamically during run-time with **`malloc()`** and deallocated using **`free()`**

Not like Java

- No **`new`**
- No garbage collection
- You ask for *n* bytes
  - Not a high-level request such as "I'd like an instance of class **`String`**"

malloc

- Allocates memory in the heap
  - Lives between function invocations
    - Functional variables disappear after a function return

- Example
  - Allocate an integer
    - `int* iptr = (int*) malloc(sizeof(int));`
  - Allocate a structure
    - `struct name* nameptr = (struct name*) malloc(sizeof(struct name));`

free

- Deallocates memory in heap.
- Pass in a pointer that was returned by **`malloc`**.
- Example
  - ```
    int* iptr =
      (int*) malloc(sizeof(int));
    free(iptr);
    ```
- Caveat: don't free the same memory block twice!
    ```

Function Parameters

- Function arguments are passed “by value”.

- What is “pass by value”?
  - The called function is given a copy of the arguments.
- What does this imply?

  - The called function can’t alter a variable in the caller function, but its private copy.

- Three examples

Example 1: swap_1

```
void swap_1(int a, int b)
{
   int temp;
   temp = a;
   a = b;
   b = temp;
}

void call_swap_1( ) {
   int x = 3;
   int y = 4;
   swap_1(x, y);
   /* values of x and y ? */
}
```

Q: Let x=3, y=4, after swap_1(x,y); x =? y=?

~~A1: x=4, y=3;~~

A2: x=3; y=4;

Example 2: swap_2

```
void swap_2(int *a, int *b)
{
   int temp;
   temp = *a;
   *a = *b;
   *b = temp;
}

void call_swap_2( ) {
   int x = 3;
   int y = 4;
   swap_1(&x, &y);
   /* values of x and y ? */
}
```

Q: Let x=3, y=4, after swap_2(&x,&y); x =? y=?

~~A1: x=3; y=4;~~

A2: x=4; y=3;

Example 3: scanf (read an input)

```
#include <stdio.h>

int main()
{
   int  x;
   scanf("%d\n", &x);
   printf("%d\n", x);
}
```

Q: Why using pointers in scanf?

A: We need to assign the value to x.