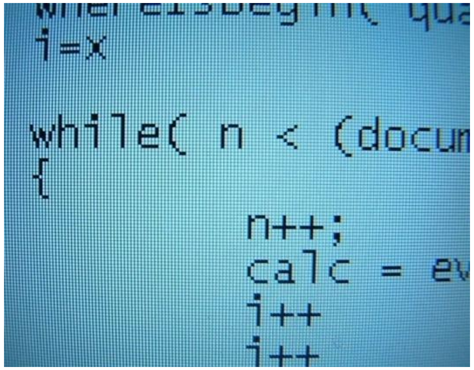
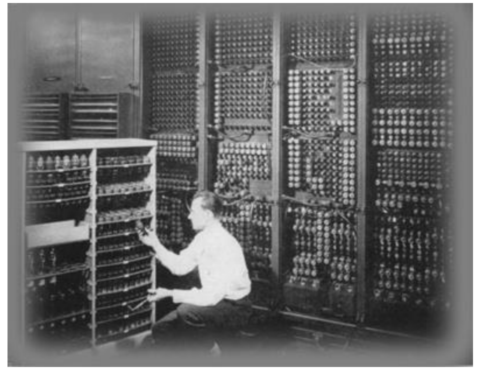




CS1PR16

Pointers: Direct Memory Control

Learning Objectives

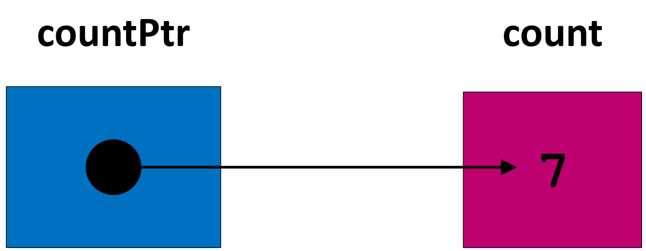
- Utilizing the syntax of pointers and arrays and their operators
- Declaring and manipulating pointer variables
- Performing basic pointer arithmetic
- Sketching the layout of arrays and pointers on the stack

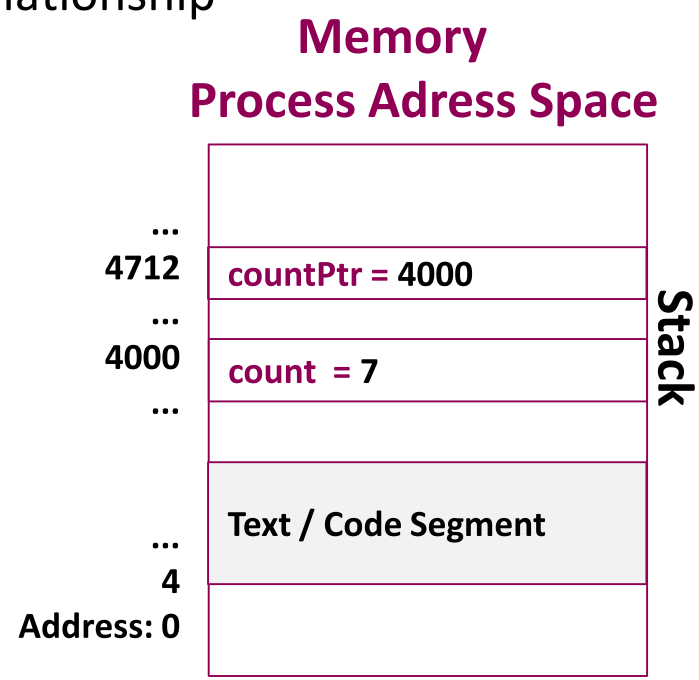
Outline

- Introduction to Pointers (second part next week)
- Differences and similarities to arrays
- Memory layout of arrays
- Examples of multidimensional arrays and their layout
  - A 3D example for your reference

Pointers

- *A **pointer** is a data type whose value refers directly to (or "**points** to") another value stored elsewhere in the computer memory using its address* [Wikipedia]
- Thus, the object contains the address of another object
- Pointers are elementary and important for performance but **dangerous**
- To illustrate, use a schema that draws the relationship
- Example (schema)





Syntax (1)

- Syntax during declaration: <type> * <name>

  ```
  int *countPtr;
  ```

- Assigning a pointer object assigns its value to a memory address

  countPtr = 4000; // the pointer has now the address 4000 (see example before)!

  – How do we know the "address" of a variable?

- Address-of operator: determines the memory address of a variable

  Syntax: & <identifier>

  ```
  countPtr = &count;
  ```

- Dereference operator: unary operator
  – Accesses the data of the object, a pointer is pointing to
  – Syntax: * <identifier>
  – Can be used as lvalue or rvalue

  ```
  int count2 = *countPtr; // assign the object countPtr points to
  *countPtr = 5; // set the object countPtr points to
  ```

- Pointer arithmetic: you can add/subtract addresses of pointers!

  "countPtr + 4" is a valid pointer again!

Syntax (2)

- Pointers can point to every memory location (even invalid)!
  - NULL is a special pointer address, indicating that a pointer is invalid
  - Always initialize your pointers
- Pointers to structs, access of members via * or ->

```
typedef struct {
        int id;
        char name[20];
} user_t;

user_t x;
user_t *p; // this is a pointer to a struct object, it is not yet set to anything

p = &x;

(*p).id = 4; // setting the id the pointer points to, cumbersome
p->id = 4;   // this is an equivalent notation
```

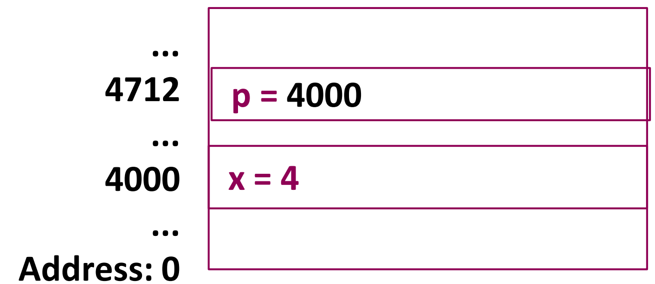
Pointer Arithmetic

University of Reading

- Adding/Subtracting of a pointer is syntactically correct code
  - Increments/decrements in `sizeof(<base-type>)`
    - Compiler knows the size, e.g. +4 for an integer
- Example

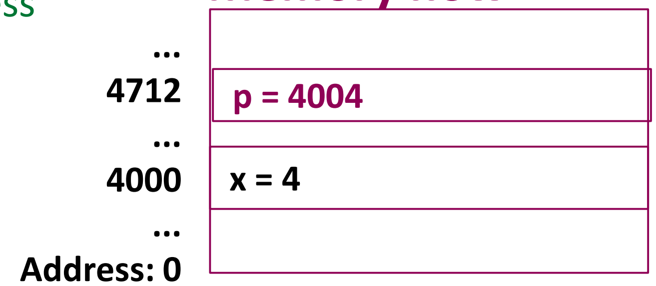
```
int x = 4;
int *p = &x;
```

Memory now




```
p += 1; // point to the integer stored on the next address
```

Memory now



- It might be unclear what is on memory address 4004

Pointer Arithmetic (2)

- [] operator: alternative of using arithmetic like ptr+4
  - ptr[x] is equivalent to *(ptr+x)

```
int *p = &x;
p[1] = 1;
*(p+1) = 1; // equivalent to p[1] = 1
```

- Retrieve the address again using &

  & ptr[x] is equivalent to (ptr+x)

- I prefer this notation as it explicits states dealing with a pointer
- [] notation is familiar: arrays are pointers too! (but special)

Void Pointers

- A special type for pointers is the void pointer
  - It means we do not know (or tell) the compiler what the type is
  - No pointer arithmetic, as the type it points to is unknown
  - Useful for data structures that work with any type
  - Programmer may give type info explicitly: use pointer casting
- Example for using pointer casting:

```
void inc_generic(void * p, enum type_e typ){
    switch(typ){
            case(TYPE_INTEGER):
            int * i = (int*) p;
            *i++; // do something with i
    } ...
}
int x = 5;
inc_generic(& x, TYPE_INTEGER);
```

Pointers to Pointers

- Pointers can point to pointers, leading to a hierarchy
  - Use more dereference pointers to change the value
- Example

```
int x = 4;
int   *p = &x;
int **dp = &p;
```

Memory

4716	**dp = 4712**
4712	**p = 4000**
...	
4000	**x = 4**
...	
Address: 0	

```
**dp = 3; // change the value of the object the pointer dp points to
```

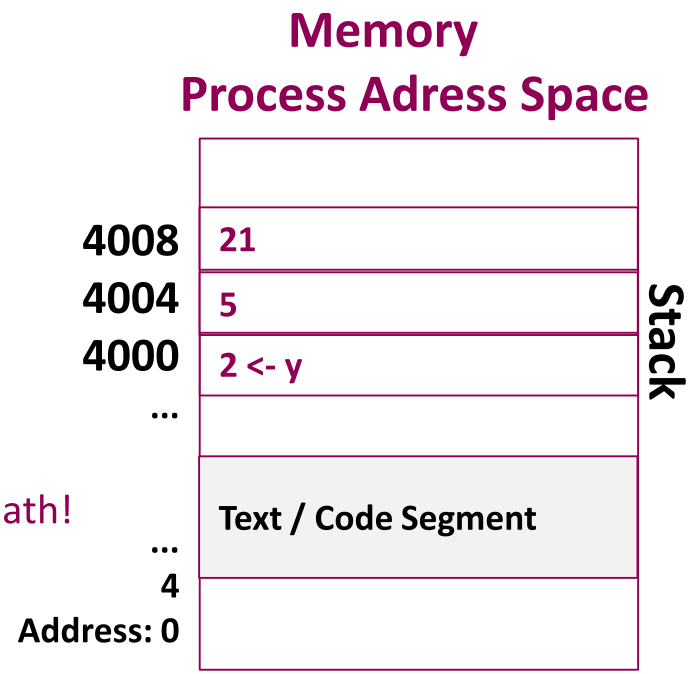
- Pointer arithmetic works too, going to the next pointer

```
dp++;
```

- Indirect Pointer: can be used to access multi-dimensional "arrays"

```
dp[9][20] = 4; // access the 10th pointer, then use an offset of 20 objects
```

Arrays and Memory



- Declaration (and definition)

  int y[] = {2, 5, 21};

  int index = 1;

- To access a single element's value:
  - Set the value

    y[index] = 5; // now know that that means pointer math!
  - Get the value

    x = y[index];

- Since the compiler knows: &y = y the address-of operator is optional
- An array is actually an address to a "serialized" sequence of objects
  - In the example, y could be the address "4000", assigned by the compiler
- The array starts at the address contained by the array variable
  - That is different from an explicit pointer which needs extra space!
  - The compiler knows the location via the symbol table

Indirect Pointers & Memory

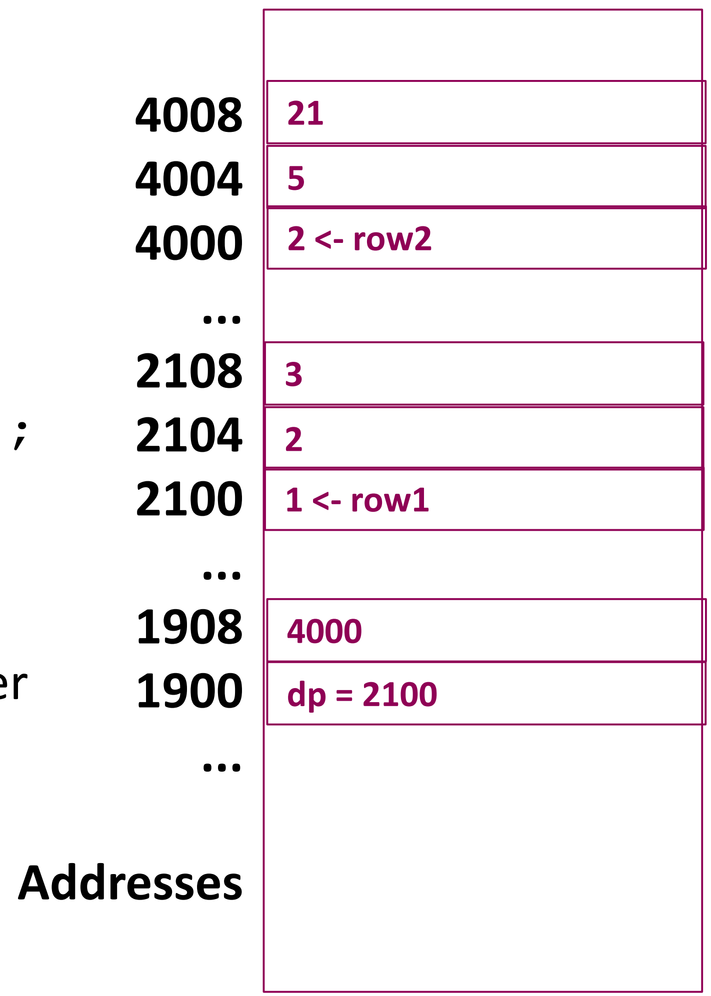
University of Reading

- Example, a 2D array

```
int row1[] = {2,5,21};
int row2[] = {1,2,3};

int * dp[2] = {& row1, & row2};
```

- Compared to a 2D array
  - Pointers are stored explicitly
  - Full control, not managed by the compiler
- We could create "ragged" arrays
  - Like a triangular matrix

Memory
Process Adress Space



Errors Using Pointers

- Pointers are very powerful, but error-prone/used for hacking
- Memory addresses of the application change every time run
- You may produce a program that sometimes works
  - That doesn't mean it is correct; on the contrary, it is likely wrong!
- What can happen when accessing an invalid pointer?
  - Example Code:

```
int *p = 4004; // this means that we point to memory address 4004
*p = 2;
```

- It is typically unclear what is on memory address 4004
- If the memory page is not yet assigned by the operating system
  - The application crashes => segmentation fault
  - You are lucky if the application crashes! Your program has an error that leads to a fault
- If the memory page belongs to your program
  - The address might be somewhere on the stack belonging to a useful object
  - If you change the return address of a function, "things" can happen => hacking

Arrays: Recapitulation

Good practice: define a constant for the size of arrays

```
#define SIZE 10 /* C preprocessor macro */

int x[SIZE];
int i, j;

for(i=0; i<SIZE; i++){
  x[i] = ...
}
...
for(j=0; j<SIZE; j++){
  ...x[j]...
}
```

Wrong code to access invalid members:

```
x[SIZE]
x[SIZE+1]
x[-1]
```

Certainly, the program is wrong!
You are lucky when it crashes!

Your responsibility to enforce arrays bounds

Now we know what happens when accessing invalid members!

Group Work

Task: Sketch a memory layout and problem for the code:

- Describe the problem with the access

```
#define SIZE 3

int i;
int x[SIZE];
int y = 44;

for(i=0; i <= SIZE; i++)
   x[i] = 2;
```

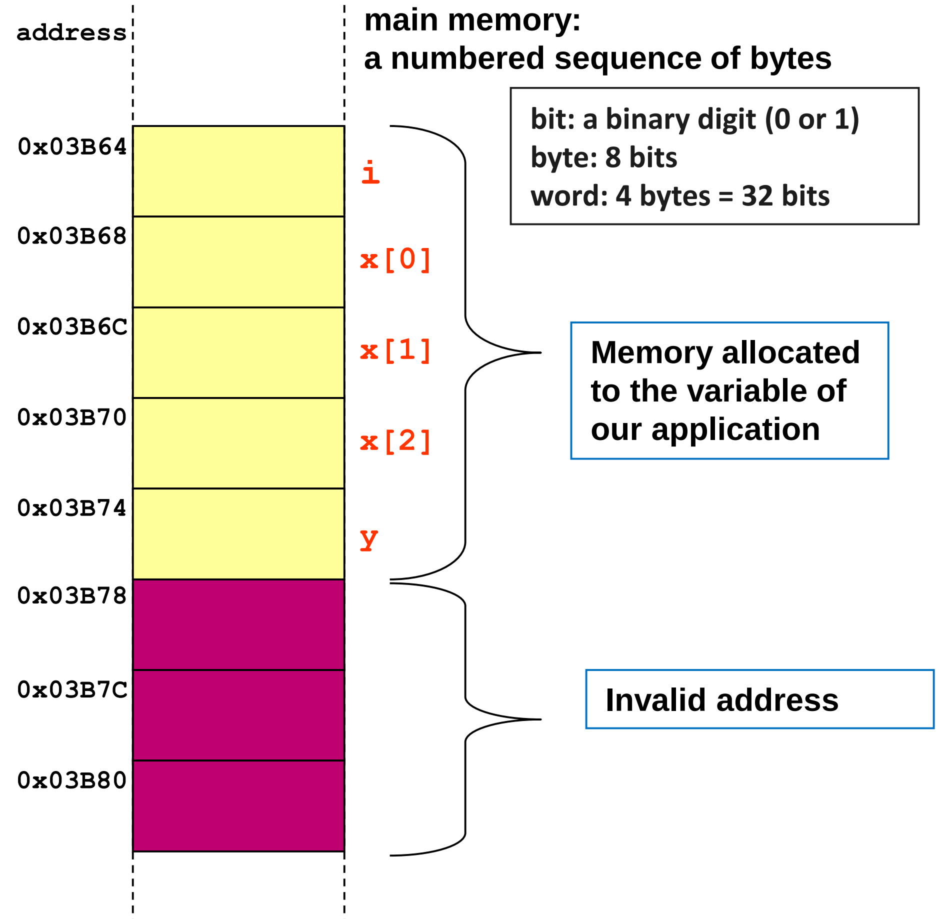
Time: 3 min

Share: 2 min

Think about how the local variables are layout as objects on the stack memory

Array Bounds Overflow Error

```
#define SIZE 3

int i;
int x[SIZE];
int y = 44;

for(i=0; i<SIZE; i++)
    x[i] = 1;

for(i=0; i<=SIZE; i++) //wrong
    x[i] = 2;

for(i=0; i<10; i++) //wrong
    x[i] = 3;
```

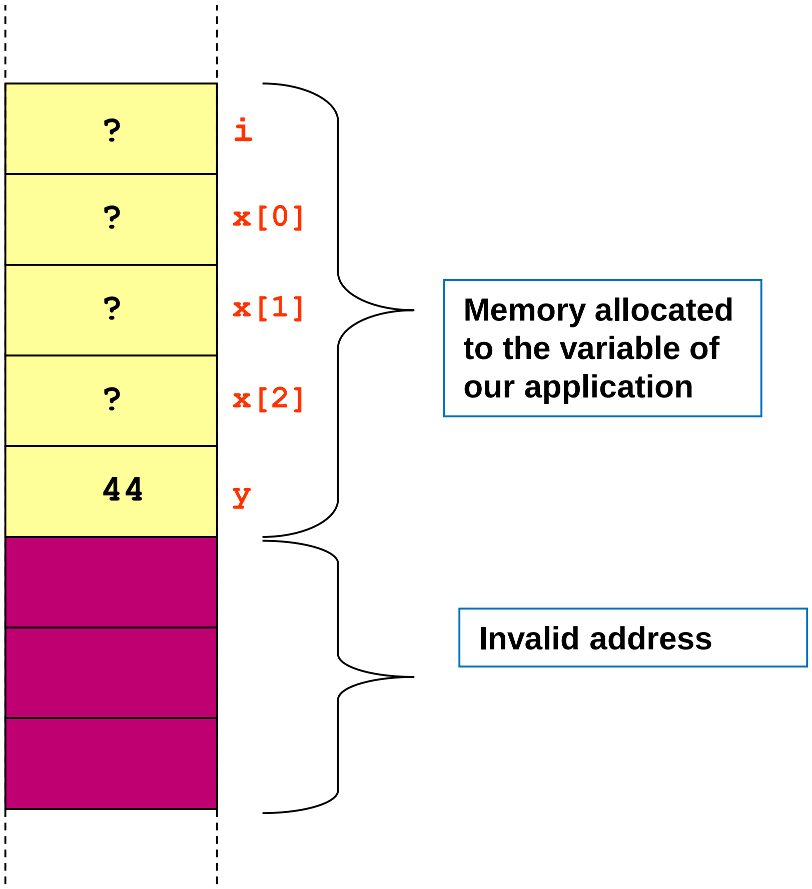


This is just one possible layout of local variables, i and y could be before, or after

Array Bounds Overflow Error

```
#define SIZE 3

int i;
int x[SIZE];
int y = 44;

for(i=0; i<SIZE; i++)
   x[i] = 1;

for(i=0; i<=SIZE; i++) //wrong
   x[i] = 2;

for(i=0; i<10; i++) //wrong
   x[i] = 3;
```

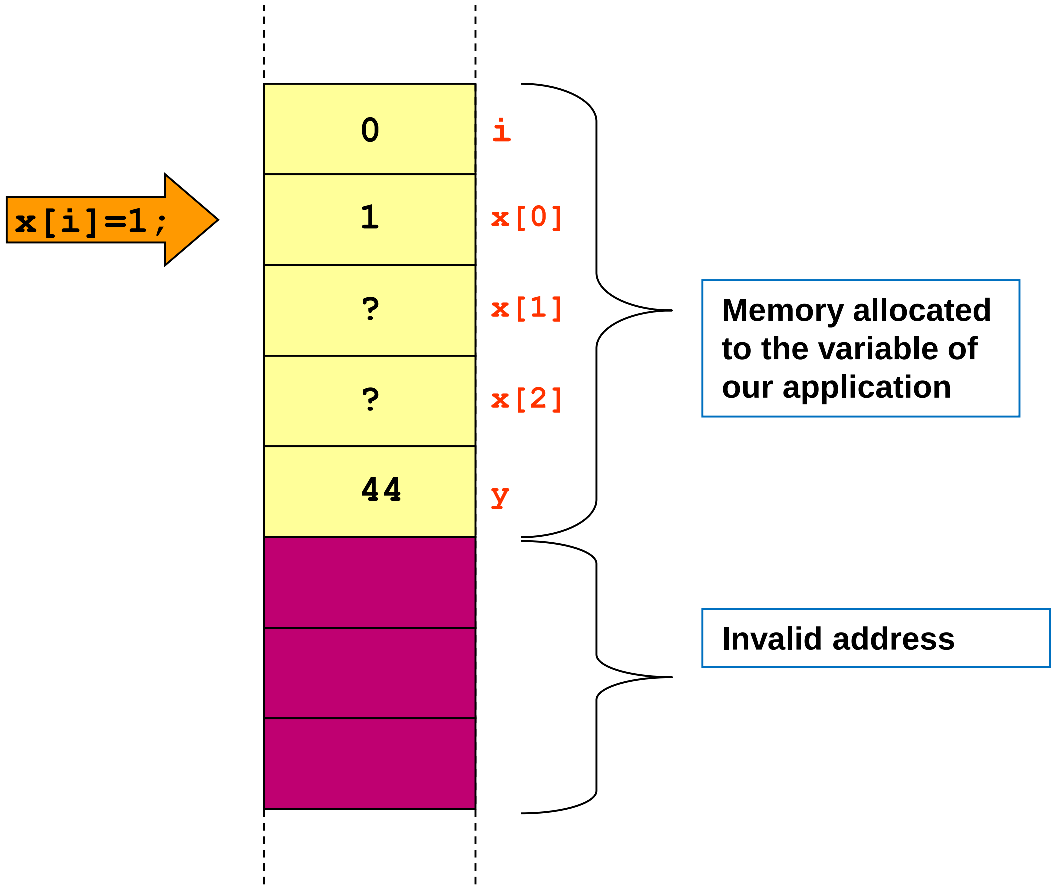
Values after declaration



Array Bounds Overflow Error

```
#define SIZE 3

int i;
int x[SIZE];
int y = 44;

for(i=0; i<SIZE; i++)
    x[i] = 1;

for(i=0; i<=SIZE; i++) //wrong
    x[i] = 2;

for(i=0; i<10; i++) //wrong
    x[i] = 3;
```

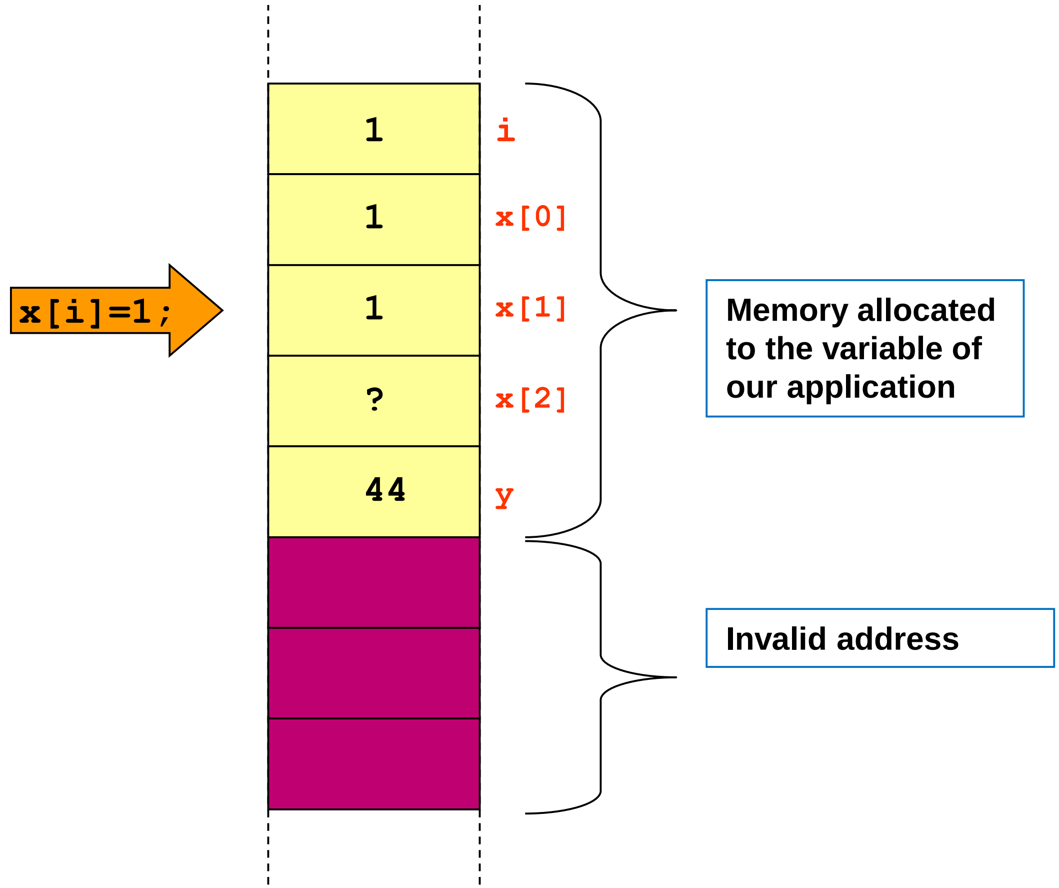
Values at iterations of first for loop



Array Bounds Overflow Error

```c
#define SIZE 3

int i;
int x[SIZE];
int y = 44;

for(i=0; i<SIZE; i++)
    x[i] = 1;

for(i=0; i<=SIZE; i++) //wrong
    x[i] = 2;

for(i=0; i<10; i++) //wrong
    x[i] = 3;
```

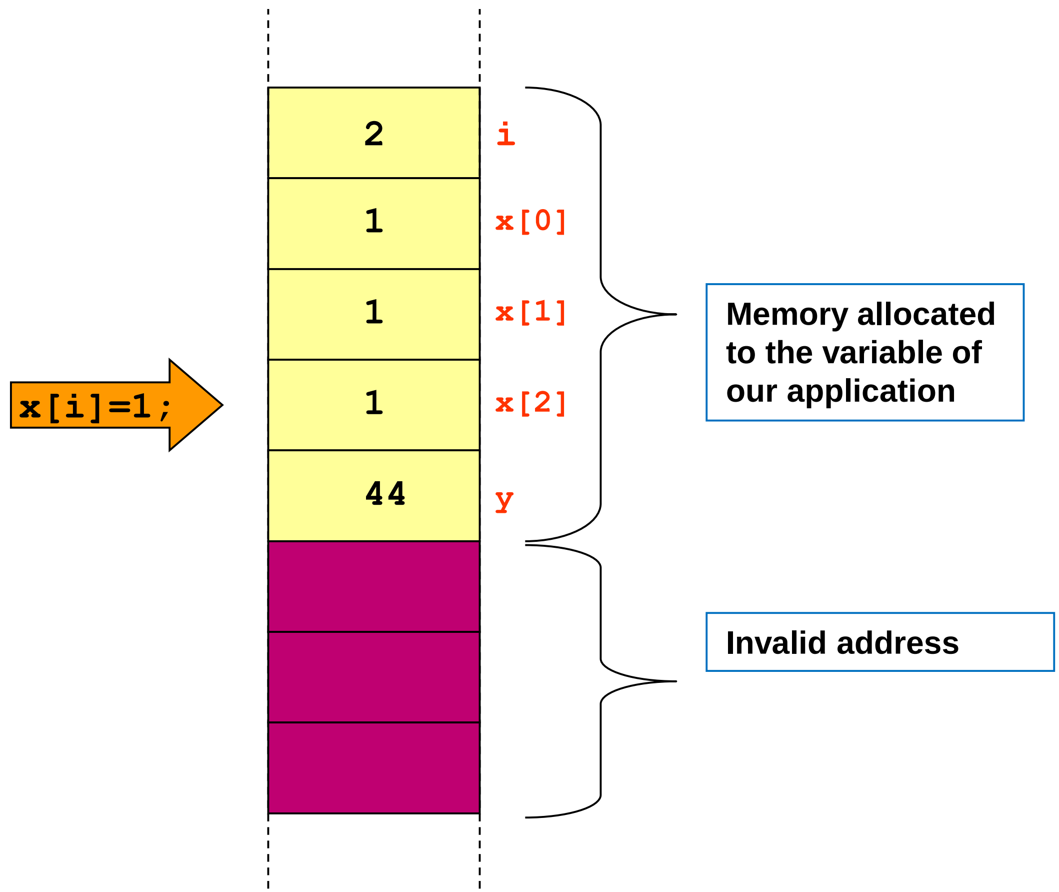
Values at iterations of first for loop



Array Bounds Overflow Error

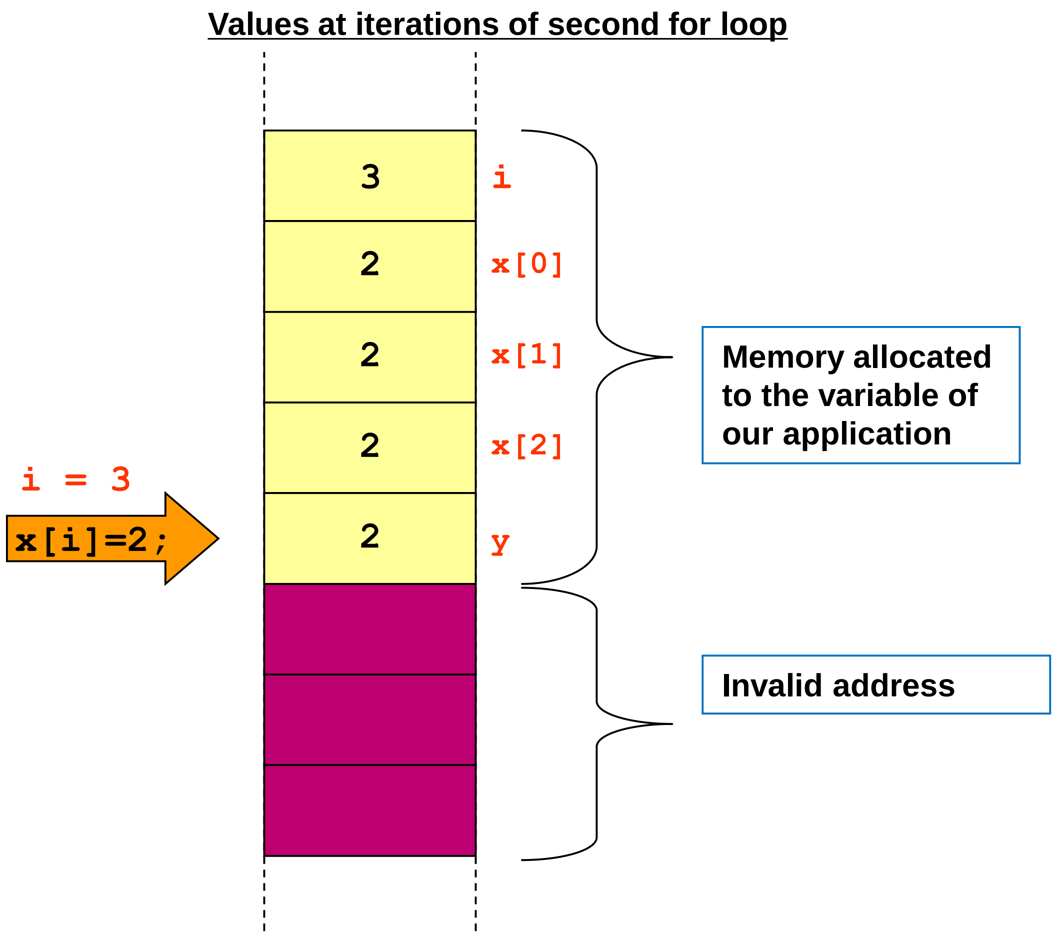
```
#define SIZE 3

int i;
int x[SIZE];
int y = 44;

for(i=0; i<SIZE; i++)
    x[i] = 1;

for(i=0; i<=SIZE; i++) //wrong
    x[i] = 2;

for(i=0; i<10; i++) //wrong
    x[i] = 3;
```

Values at iterations of first for loop



Array Bounds Overflow Error

```
#define SIZE 3

int i;
int x[SIZE];
int y = 44;

for(i=0; i<SIZE; i++)
   x[i] = 1;

for(i=0; i<=SIZE; i++) //wrong
   x[i] = 2;

for(i=0; i<10; i++) //wrong
   x[i] = 3;
```

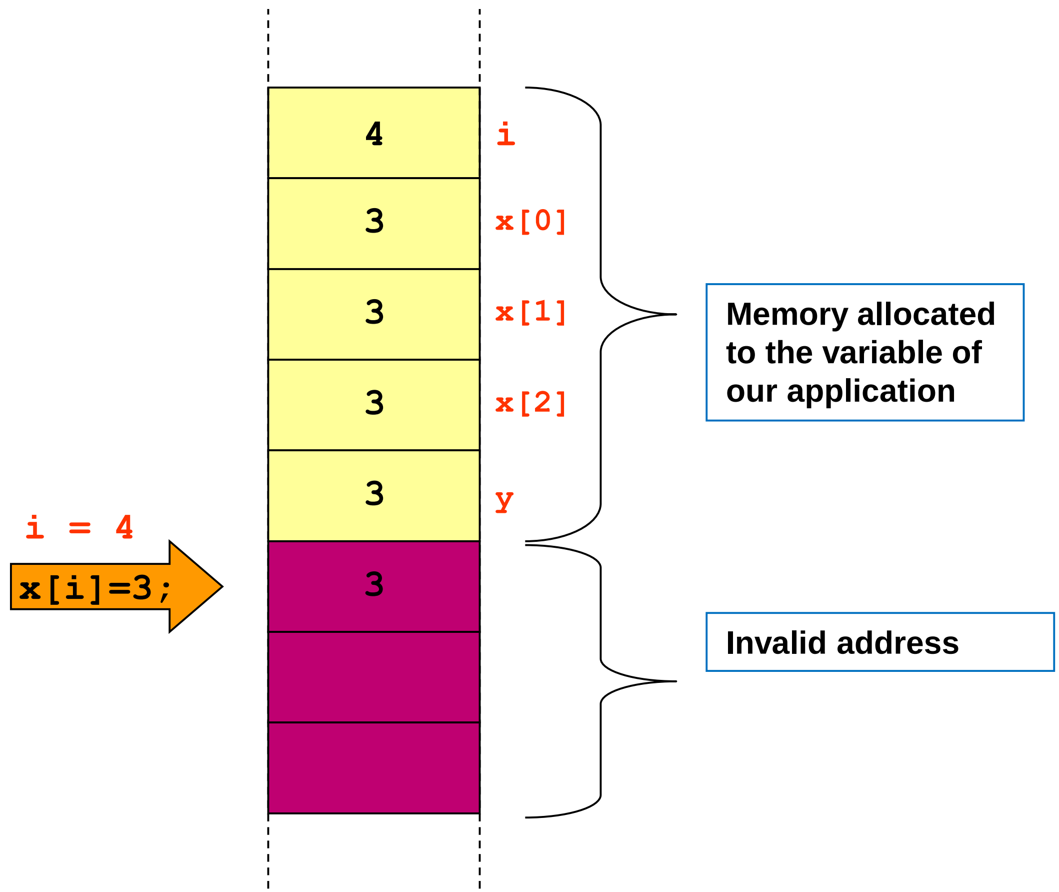


Array Bounds Overflow Error

University of Reading

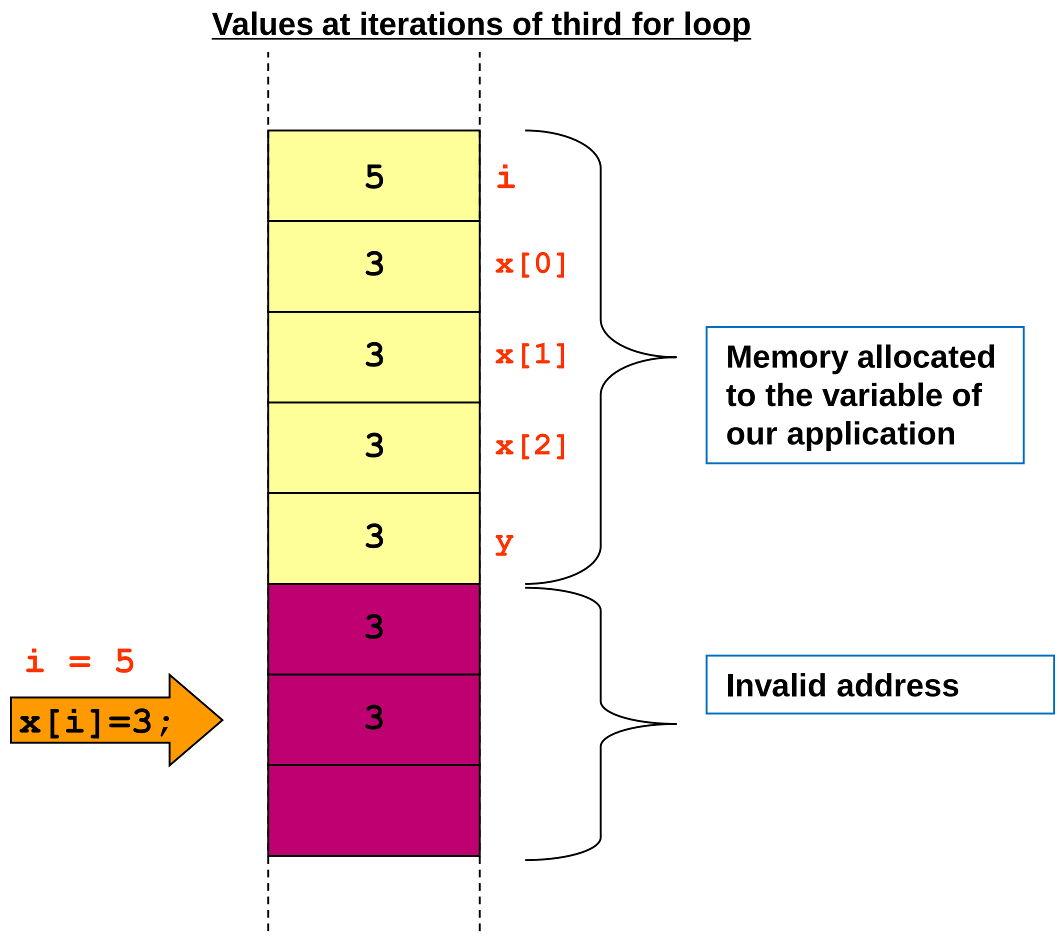
```c
#define SIZE 3

int i;
int x[SIZE];
int y = 44;

for(i=0; i<SIZE; i++)
    x[i] = 1;

for(i=0; i<=SIZE; i++) //wrong
    x[i] = 2;

for(i=0; i<10; i++) //wrong
    x[i] = 3;
```

Values at iterations of third for loop



Array Bounds Overflow Error

```
#define SIZE 3

int i;
int x[SIZE];
int y = 44;

for(i=0; i<SIZE; i++)
   x[i] = 1;

for(i=0; i<=SIZE; i++) //wrong
   x[i] = 2;

for(i=0; i<10; i++) //wrong
   x[i] = 3;
```



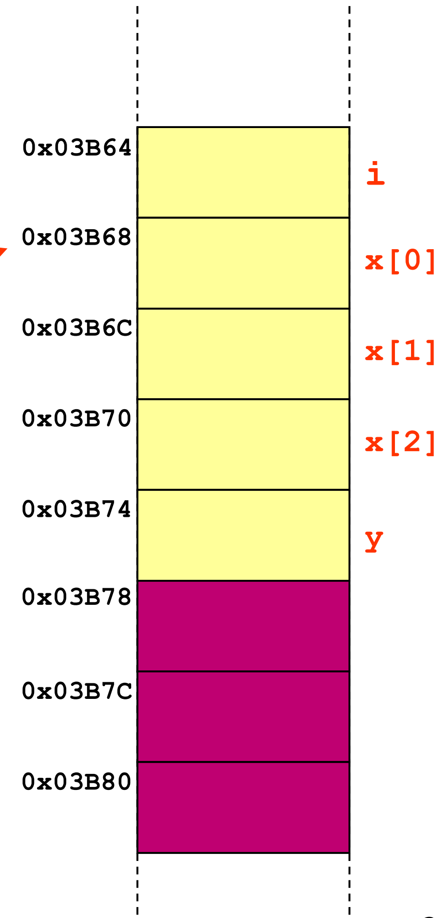
The Array Identifier: Recapitulation

The identifier of an array is actually a constant and its value is the memory address of the **first element** of the array

```
#define SIZE 3

int i;
int x[SIZE];
int y = 44;

printf("the address of the array is: %X", x);
```

stdout:

```
the address of the array is: 0x03B68
```



The Notation [] of the Array

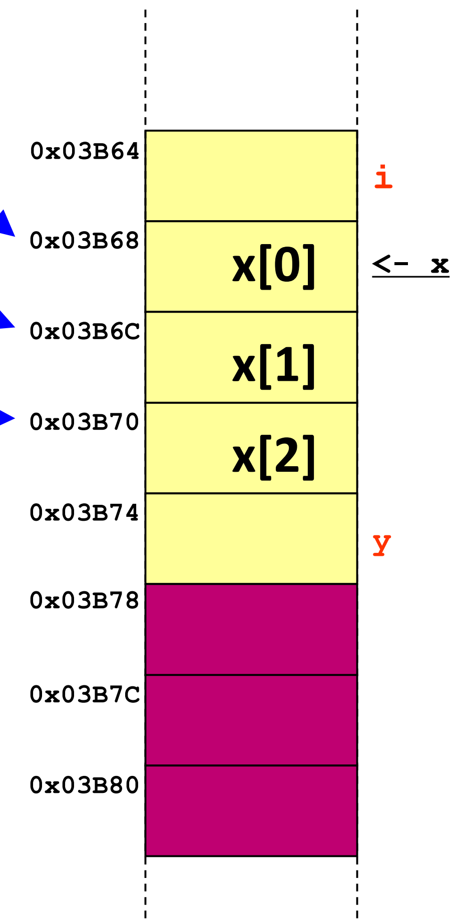
x[0] means: the value stored at the memory location of the array "x" + 0 elements

x[1] means: the value stored at the memory location of the array "x" + 1 element

x[2] means: the value stored at the memory location of the array "x" + 2 elements

and

x[any_number] means: the value stored at the memory location of the array "x" + "any_number" elements



Multidimensional Arrays: 1-D

```
#define SIZE1 2 /* # of columns */

int c;
int x[SIZE1];


for(c=0; c<SIZE1; c++)
   x[c] = 0;
```

	col 0	col 1
row 0	x[0]	x[1]

- ❑ 1 row (implicit)
- ❑ 2 columns

<u>ONE 1-D array</u> of two elements

2-Dimensional Arrays

```
#define SIZE1 2 /* # of columns */
#define SIZE2 3 /* # of rows */

int r, c, i;
int x[SIZE2][SIZE1];


for(r=0; r<SIZE2; r++) //r: row
   for(c=0; c<SIZE1; c++) //c: column
        x[r][c] = 0;
```

	col 0	col 1
row 0	x[0][0]	x[0][1]
row 1	x[1][0]	x[1][1]
row 2	x[2][0]	x[2][1]

- ❑ **3 row**
- ❑ **2 columns**

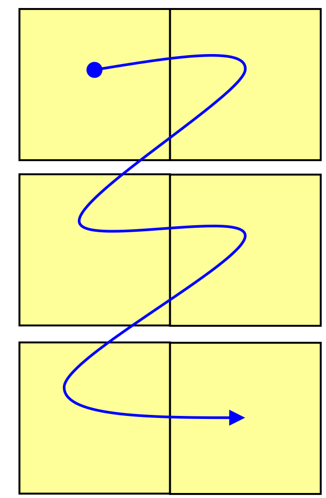
- **There are <u>THREE 1-D arrays</u> of 2 elements**
- **In total we have <u>6 elements</u>**

2-Dimensional Arrays

```
#define SIZE1 2 /* # of columns */
#define SIZE2 3 /* # of rows */

int r, c;
int x[SIZE2][SIZE1];


...


for(r=0; r<SIZE2; r++) //r: row
    for(c=0; c<SIZE1; c++) //c: column
        print("x[%d][%d] = %d\n",
                r,   c,    x[r][c]);
```

	col 0	col 1
row 0	x[0][0]	x[0][1]
row 1	x[1][0]	x[1][1]
row 2	x[2][0]	x[2][1]

How we iterate:



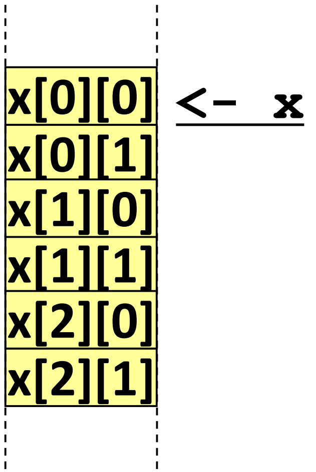
stdout:

```
x[0][0] = …
x[0][1] = …
x[1][0] = …
x[1][1] = …
x[2][0] = …
x[2][1] = …
```

2-Dimensional Arrays Memory Allocation

University of Reading

```
#define SIZE1 2 /* # of columns */
#define SIZE2 3 /* # of rows */

int r, c;
int x[SIZE2][SIZE1];
 → the compiler needs to allocate 2*3=6 elements
```



- The elements are arranged in main memory in 1-D
  - The rightmost index changes more quickly
  - The leftmost index changes more slowly
- In two nested for loops
  - The inner loops changes more quickly ← rightmost index
  - The outer loop changes more slowly ← leftmost index

2-Dimensional Arrays

University of Reading

```
#define SIZE1 2 /* # of columns */
#define SIZE2 3 /* # of rows */

int r, c, i;
int x[SIZE2][SIZE1];
int y[][2] = {1,6,3,7,0,2};
```

```
int y[][2] = {  1, 6,
                3, 7,
                0, 2};
```

x[0][0]	<- x
x[0][1]	
x[1][0]	
x[1][1]	
x[2][0]	
x[2][1]	

This is exactly the same layout of a 1-D array of 6 elements!
int x[SIZE1*SIZE2];

But the compiler knows how to organize

2-D Array using Pointers

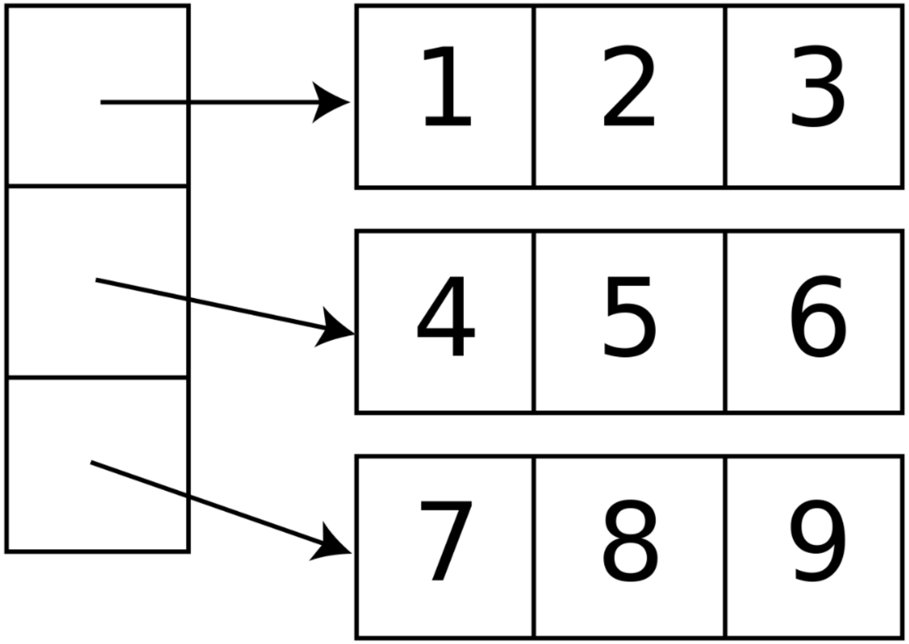
- Alternative to a 2D array
- Array of pointers

```
int *a[3]; // array of 3 pointers
a[0] = (int[3]){1,2,3};
a[1] = (int[3]){4,5,6};
a[2] = (int[3]){7,8,9};
```

- Element access is the same:

  a[i][j]

- The storage layout is very different



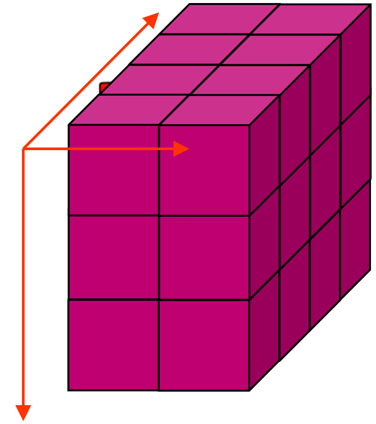
Initialization of the Array

- How does the casting work?

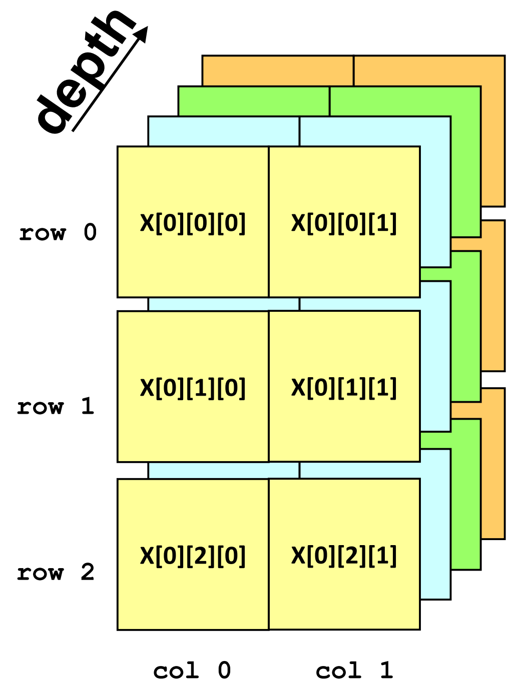
```
int *a[5]; // array of 5 pointers
a[0] = (int[3]){1,2,3}
```

- The compiler knows that a is an array of 5 pointers
- (int[3]){1,2,3} creates an array with 3 integers
  - somewhere in memory
- This is also a pointer, we assign a[0] to this memory location

3-Dimensional Arrays



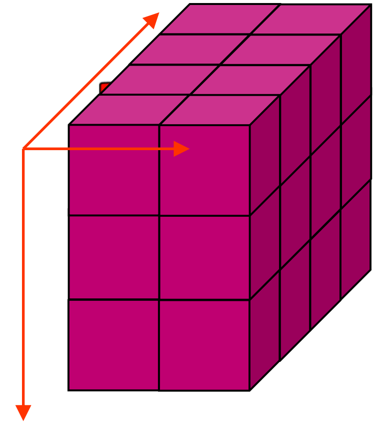
```
#define SIZE1 2 /* 1st dimension (columns) */
#define SIZE2 3 /* 2nd dimension (rows) */
#define SIZE3 4 /* 3rd dimension (depth) */

int i1, i2, i3;
int x[SIZE3][SIZE2][SIZE1];

for(i3=0; i3<SIZE3; i3++)
    for(i2=0; i2<SIZE2; i2++)
          for(i1=0; i1<SIZE1; i1++)
                    x[i3][i2][i1] = ...
```



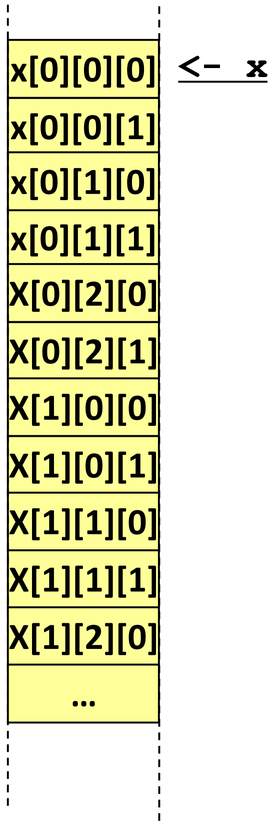
- ❑ **4 layers**
- ❑ **3 rows**
- ❑ **2 columns**

- **There are <u>FOUR 2-D arrays</u> of 3 rows and 2 columns**
- **There are <u>TWELVE 1-D arrays</u> of 2 elements**
- **In total we have <u>24 elements</u>**

3-Dimensional Arrays



```
#define SIZE1 2 /* 1st dimension (columns) */
#define SIZE2 3 /* 2nd dimension (rows) */
#define SIZE3 4 /* 3rd dimension (depth) */

int i1, i2, i3;
int x[SIZE3][SIZE2][SIZE1];
 → the compiler needs to allocate 2*3*4=24 elements

for(i3=0; i3<SIZE3; i3++)
    for(i2=0; i2<SIZE2; i2++)
          for(i1=0; i1<SIZE1; i1++)
                    x[i3][i2][i1] = ...
```



- The elements are arranged in main memory, a 1-D structure.
  - The rightmost index changes more quickly
  - The leftmost index changes more slowly

- In three nested for loops
  - The inner loops changes more quickly ← rightmost index
  - The outer loop changes more slowly ← leftmost index

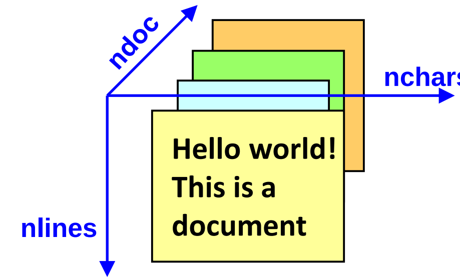
4-Dimensional Arrays

```
#define SIZE1 2 /* 1st dimension */
#define SIZE2 3 /* 2nd dimension */
#define SIZE3 4 /* 3rd dimension */
#define SIZE3 5 /* 4th dimension */

int i1, i2, i3, i4;
int x[SIZE4][SIZE3][SIZE2][SIZE1];
 → the compiler needs to allocate 2*3*4*5=120 elements

for(i4=0; i4<SIZE4; i4++)
  for(i3=0; i3<SIZE3; i3++)
   for(i2=0; i2<SIZE2; i2++)
     for(i1=0; i1<SIZE1; i1++)
       printf("x[%d][%d][%d][%d] -> %d\n",i4, i3, i2, i1, x[i4][i3][i2][i1]);
```

- **Think of each additional dimension as a collection of objects**
  - **Here we have a collection of FIVE <u>3-Dimensional arrays</u>**

Example: A Collection of Documents (1/4)

University of Reading

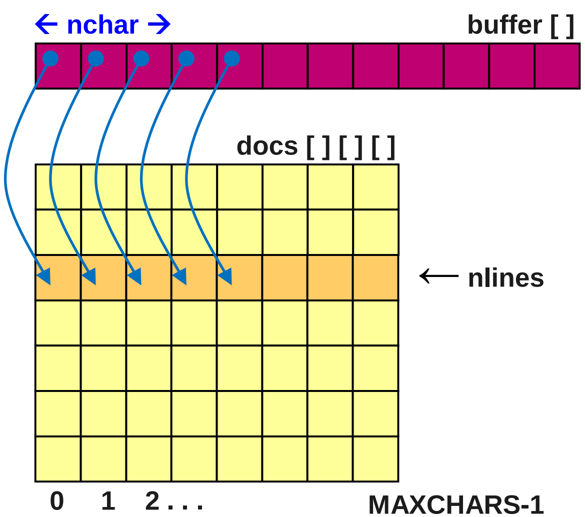



```
#include <stdio.h>

#define MAXDOCS  10  /* max number of documents */
#define MAXLINES 100 /* max number of lines per document */
#define MAXCHARS 80  /* max number of chars per line */

#define MAXBUFFERSIZE 1024  /* max number of chars in text buffer */

int main(){
    /* 'docs' is our main data structure: a collection of documents */
    char docs[MAXDOCS][MAXLINES][MAXCHARS]; /* 3-D array */
    int ndoc=0, nline=0, nchar=0; /* 3 indexes, one for each dimension */

    char buffer[MAXBUFFERSIZE]; /* temporary var to store a string from stdin */
    char answer;
    int tot_docs, tot_lines, tot_chars;

    printf("Hello there! You can type documents (at most %d).\n\n", MAXDOCS);

    do{ /* repeat for each document */
          printf("You can now enter a document of text.\n");
          printf("Enter an empty line when your input is complete.\n\n");
```

Example: A Collection of Documents (2/4)

University of Reading

```
nline=0;
gets(buffer);    /* read first line */
while(strlen(buffer) != 0){
   /* copy the buffer into our main data structure and check the array's bounds */
   if(strlen(buffer) >= MAXCHARS){ /* max number of chars in a line is MAXCHARS */
         for(nchar=0; nchar<MAXCHARS-4; nchar++){
            docs[ndoc][nline][nchar] = buffer[nchar];
         }
         docs[ndoc][nline][nchar++] = '.';
         docs[ndoc][nline][nchar++] = '.';
         docs[ndoc][nline][nchar++] = '.';
         docs[ndoc][nline][nchar] = '\0';
   } else {
         for(nchar=0; nchar<strlen(buffer); nchar++){
            docs[ndoc][nline][nchar] = buffer[nchar];
         }
         docs[ndoc][nline][nchar] = '\0';
   }

   nline++;
   if(nline >= MAXLINES){
         printf("Warning: max number of lines in a doc is %d\n", MAXLINES);
         printf("Sorry, you cannot enter more text for this document.\n");
         break;
   }
```



Example: A Collection of Documents (3/4)

University of Reading

```
         gets(buffer); /* read next string from stdin */
      } /* end of while loop for a document */
      docs[ndoc][nline][0] = '\0'; /* end of this document's lines */

      ndoc++;
      if(ndoc < MAXDOCS){
         printf("\nWould you like to enter another document? (y/n): ");
         fflush(stdin);
         answer = getchar();
      } else {
         printf("\nSorry you have already entered the max number of documents
(%d).\n", MAXDOCS);
         answer = 'n';
      }
} while(answer == 'y');
/* end of do-while loop for the documents */
```

docs [ ] [ ] [ ]

H	e	l	l	o	\0		
T	h	i	s		i	s	\0
a		d	o	c	\0		
\0							

Example: A Collection of Documents (4/4)

```
    /* Let's count our data: */
    tot_docs = ndoc;
    tot_lines = 0;
    tot_chars = 0;
    for(ndoc=0; ndoc<tot_docs; ndoc++){
          printf("\n> Document n. %d\n", ndoc+1);
          nline=0;
          nchar=0;
          while(strlen(docs[ndoc][nline]) > 0){
             printf("%s\n", docs[ndoc][nline]);
             printf("> line n. %d has %d chars \n", nline+1, strlen(docs[ndoc][nline]));
             nchar=0;
             while(docs[ndoc][nline][nchar] != '\0'){
                    tot_chars++; /* the counter */
                    nchar++; /* the index */
             }
             tot_lines++;
             nline++;
          }
    }
    printf("You have entered %d docs, %d lines and %d chars.\n", tot_docs, tot_lines, tot_chars);
    printf("Press enter to quit.\n");
    fflush(stdin);
    getchar();
    return(0);
} /* end of main */
```

Summary

- Pointers point to objects (variables) in memory
  - Read from right to left

    int const * ptr; // ptr is a pointer to constant integer(s)
- Pointers and arrays are similar
  - ptr[1] is the value of the pointer on position 1
  - & ptr[1] is the address on position 1
  - pointer arithmetics “ptr + 1“ is the memory address of the next object
- Arrays are layouted consecutively in memory
  - They differ from pointers as a variable is the address of the first element
- The rightmost dimension of an array is sequentially in memory