Introduction to C

Programming Workshop in C (67316)
Fall 2017
Lecture 3
31.10.2017

Review

- **Variable** -name/reference to a stored value (usually in memory)
- **Data type** - determines the size of a variable in memory, what values it can take on, what operations are allowed
- **Operator** - an operation performed using 1-3 variables
- **Expression** - combination of literal values/variables and operators/functions

Review - data types

- Various sizes (**char**, **short**, **long**, **float**, **double**)
- Numeric types - **signed**/**unsigned**
- Implementation - little or big endian
- Careful mixing and converting (casting) types

Review - operators

- Unary (++) , binary (+) , ternary (?:)
- Arithmetic (+), relational (<), binary (&&), assignment (=)
- Order of evaluation (precedence, direction) (++x vs. x++)

Boolean types

Boolean types

Boolean type **doesn't exist in C**!

Use *char/int instead (It's possible to manipulate bits)*

```
#define TRUE 1
while (TRUE)
{
}
```

zero => false

non-zero => true

Examples: (Take a second)

```
while (1)
{
}
```

```
if (-1974)
{
}
```

```
i = (3==4);
```

Boolean types

Boolean type **doesn’t exist in C**! (unlike C++ or Java)

Use *char/int instead (It’s possible to manipulate bits)*

```
#define TRUE 1
while (TRUE)
{
}

(infinite loop)
```

zero => false

non-zero => true

Examples:

```
while (1)
{
}

(infinite loop)
```

```
if (-1974)
{
}

(true statement)
```

```
i = (3==4);

(i equals zero)
```

Boolean variables – example

```
int main()
{
   int a = 5;
   while(1)
   {
      if(!(a-3))
      {
         printf(”3");
         break;
      }
      printf(”%d", a--);
   }
   return 0;
}
```

Why does it evaluate to TRUE iff (a==3)?

Booleans in C99

C99 added the _Bool type. You can use it as follows:

```
#include <stdbool.h>
#include <stdio.h>
int main()
{
   bool t = true;
   bool f = false;
   if (t != f)
   {
      printf(“t=%d, f=%d\n”, t, f); // t=1, f=0
      printf(“It is %s that 3 is greater than 4.\n”,
               (3>4) ? “true” : “false”);
   }
   return 0;
}
```

What is the size of bool?

Ternary operator “?:”

```
expr1 ? expr2 : expr3
if(expr1) expr2;
else expr3;
```

Review: If else statements

```
if (expression) {
 // ... (single statement or block)
} else if (expression) {
 // ...
} else {
 // ...
}
```

If else statements

```
if (x % 4 == 0)
 if (x % 2 == 0)
   y = 2;
else
   y = 1;
```

To which if statement does the else keyword belong?

To associate `else` with outer `if` statement: use braces

```
if (x % 4 == 0) {
 if (x % 2 == 0)
   y = 2;
} else
   y = 1;
```

Back to Input/Output

Standard input and output

```
int putchar(int)
```

- put the character c on the standard output
- returns the character printed or EOF on error

```
int getchar()
```

- returns the next character from standard input or EOF on error

Character Input/Output

```
#include <stdio.h>
int main()
{
   int c;
   while( (c = getchar()) != EOF
   {
     if (c >= ‘A’ && c <= ‘Z’)
      {
         c = c - ’A’ + ’a’;
      }
     putchar(c);
   }
   return 0;
}
```

What does the following code do?

Character Input/Output

- To use a file instead of standard input, use ‘<‘ operator

```
> ./getcharExample < input_file.txt
```

- This is an OS (Unix/Linux) feature, not C
- use ‘>’ operator to redirect standard output to file

```
> ./getcharExample < input_file.txt > output_file.txt
```

- use `diff` to compare the output of your program to the “school solution program” provided to you

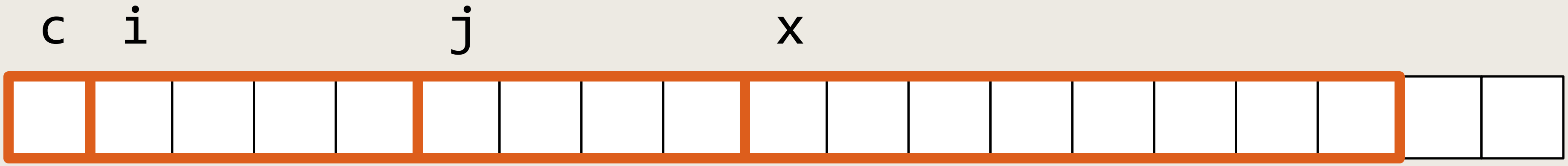
```
> diff output_file1.txt output_file2.txt
```

Memory and Arrays

For now, we will only discuss static arrays

Memory

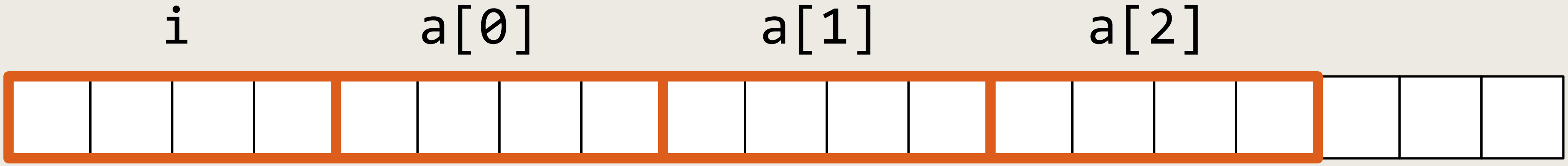
```
int main()
{
    char c;
    int i,j;
    double x;
```



Arrays

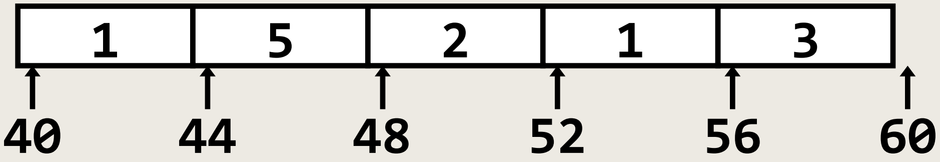
Defines a block of consecutive cells

```
int main()
{
    int i;
    int a[3];
```



Arrays - the [ ] operator

```
int arr[5] = { 1, 5, 2, 1 ,3  };
/*arr begins at address 40*/
```



1 5 2 1 3

40 44 48 52 56 60

Address Computation Examples:

```
1.   arr[0]   40+0*sizeof(int) = 40
2.   arr[3]   40+3*sizeof(int) = 52
3.   arr[i]   40+i*sizeof(int) = 40 + 4*i
4.   arr[-1]  40+(-1)*sizeof(int) = 36  // can be the code
                                        // segment or other variables
```

Arrays

C does not provide any run time checks:

```
int a[4];
a[-1] = 0;
a[4] = 0;
```

This will **compile and run**…

But can lead to unpredictable results/crash.

It is the programmer’s responsibility to check whether the index is out of bound.

Arrays

C does not provide array operations:

```
int a[4];
int b[4];

a = b; // illegal

// and how about:
if( a == b )  // legal, address comparison
```

Array Initialization

```
int arr[3] = {3, 4, 5}; // Good
int arr[]  = {3, 4, 5}; // Good: the same
int arr[3] = {0};       // Init all items to 0, takes O(n)
int arr[4] = {3, 4, 5}; // Bad style - The last is 0
int arr[2] = {3, 4, 5}; // Bad
int arr[2][3] = {{2,5,7},{4,6,7}}; // Good
int arr[2][3] = {2,5,7,4,6,7};     // Good: the same
int arr[3][2] = {{2,5,7},{4,6,7}}; // Bad
int arr[3];     // uninitialized values
arr = {2,5,7};  // Bad (compilation): array assignment only
                // in initialization
```

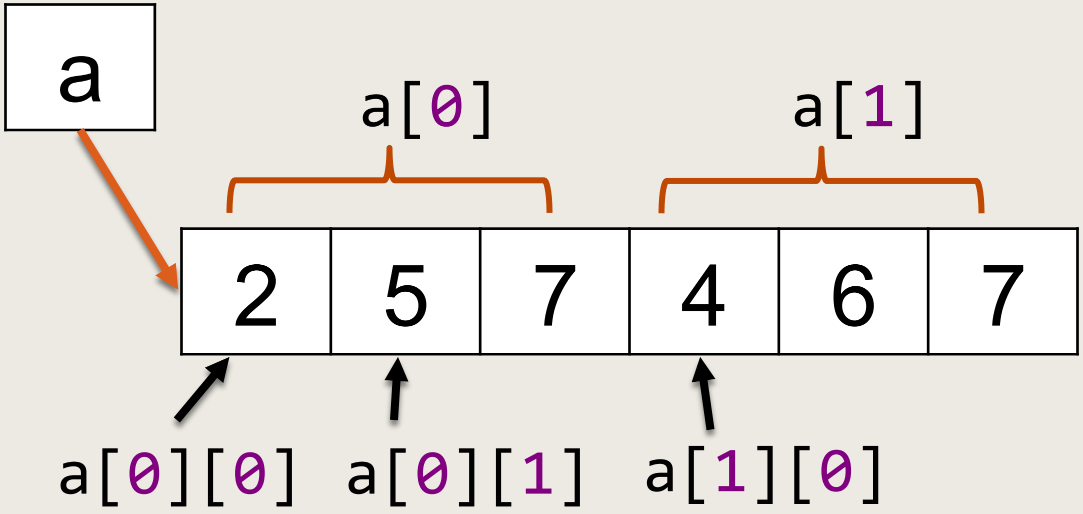
2D Array Memory Map

```
int a[2][3] = {{2,5,7},{4,6,7}};
```

Generally we would look at arrays as

```
int a[ROWS][COLS];
```

2	5	7
4	6	7



Think about a[n][m][k] etc...

Fun with Pointers

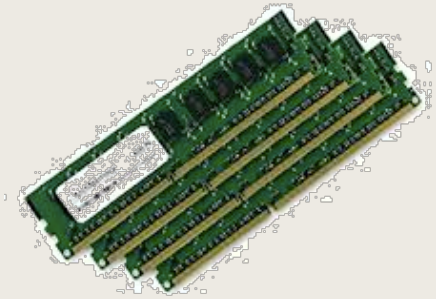
Example – the swap function

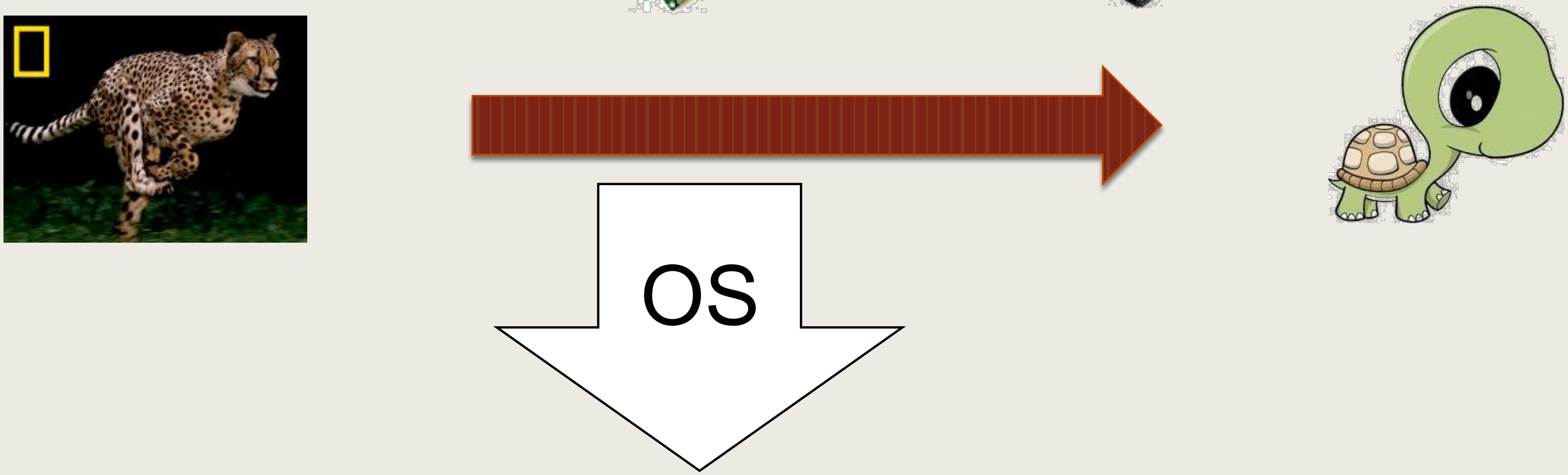
```
void swap(int a, int b)
{
   int temp = a;
   a = b;
   b = temp;
}

int main()
{
   int x, y;
   x = 3; y = 7;
   swap(x, y);
   // now x==?, y==?
}
```

Physical and virtual memory

- **Physical memory:** cache, RAM, hard disk





OS

- **Virtual memory:** addressable space accessible by your code

Addressing variables

- Every variable in memory has an address!
- How to find it? the & ampersand operator

```
#include <stdio.h>
int main()
{
    int var;
    int arr[10];

    printf("Address of var: %p\n”, &var);
    printf("Address of arr: %p\n”, &arr);
    return 0;
}
```

Pointers are variables that store the **address of other variables**

- **Pointer:** memory address of a variable
- Address can be used to access/modify a variable from anywhere
- Extremely useful for data structures
- Well known for complicating the code

Pointers declaration

- **Declaration**

  `<type> *p;  (e.g. int *p;)`

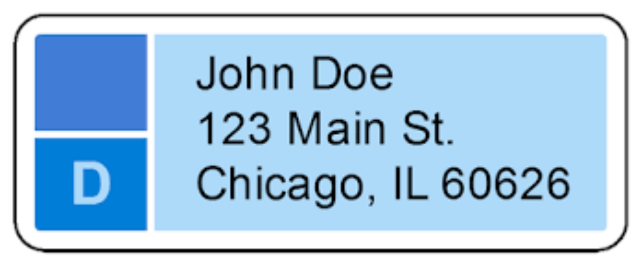
  p points to object of type `<type>`

```
int *ip; /* pointer to an integer */
double *dp; /* pointer to a double */
float *fp; /* pointer to a float */
char *ch; /* pointer to a character */
```

- What is the actual data type of the value of all pointers?
- long hexadecimal number that represents a memory address

* and &

- pointers store the address



John Doe
123 Main St.
Chicago, IL 60626

- to get the value use * operator

- & operator gets the address

* and &

```
#include <stdio.h>
int main()
{
   int var = 20; // variable declaration
   int *ip;      // pointer declaration

   // store the address of var in pointer variable
   ip = &var;

   printf("Address of var: %x\n", &var);      4fc38a78

   printf("Address stored in ip: %x\n", ip);   4fc38a78

   printf("Value of *ip variable: %d\n", *ip);  20

   return 0;
}
```

Pointers are variables that store the **address of other variables**

- **Declaration**

  `<type> *p;   (e.g. int *p;)`

  p points to object of type `<type>`

- **Pointer → value (de-reference)**

  *p refers to the object p points to

  `(e.g. *p = x;  y = *p;)`

- **Value → pointer**

  &x - the address of x `(e.g.  p = &y;)`

Pointers – spaces in declaration

```
int *p; // p is a pointer to an int
int* p; // p is a pointer to an int
int*p;  // p is a pointer to an int
int * p;// p is a pointer to an int

int *p, q; // p is a pointer to an int
           // q is an int

int* p, q; // same, but much less readable
           // so don’t do that
```

Pointers - 64 bit!

```
int main()
{
     int i,j;
⇨    int *x; // x points to an integer
     i = 1;
     x = &i;
     j = *x;
     x = &j;
     (*x) = 3;
```

i						j			x									
?	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?			
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18

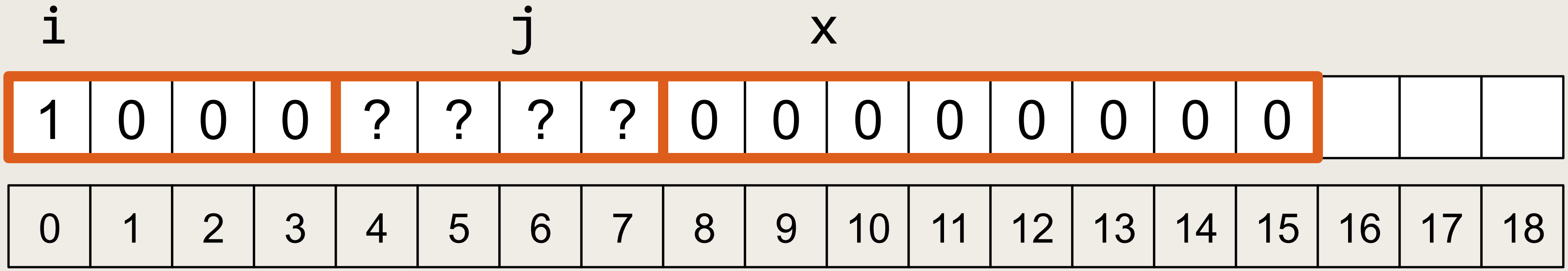
Pointers - 64 bit!

```
int main()
{
    int i,j;
    int *x; // x points to an integer
    i = 1;
    x = &i;
    j = *x;
    x = &j;
    (*x) = 3;
```

i						j				x								
1	0	0	0	?	?	?	?	?	?	?	?	?	?	?	?			
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18

Pointers - 64 bit!

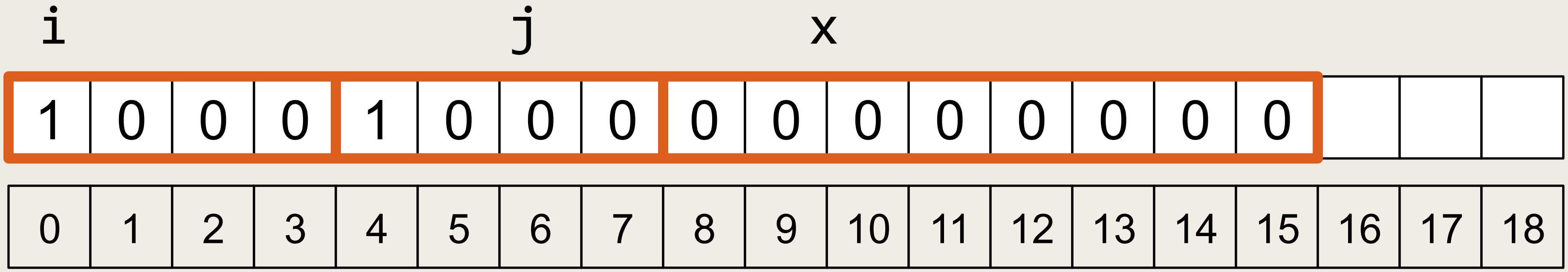
```
int main()
{
    int i,j;
    int *x; // x points to an integer
    i = 1;
    x = &i;
    j = *x;
    x = &j;
    (*x) = 3;
```



i						j				x								
1	0	0	0	?	?	?	?	0	0	0	0	0	0	0	0			
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18

Pointers - 64 bit!

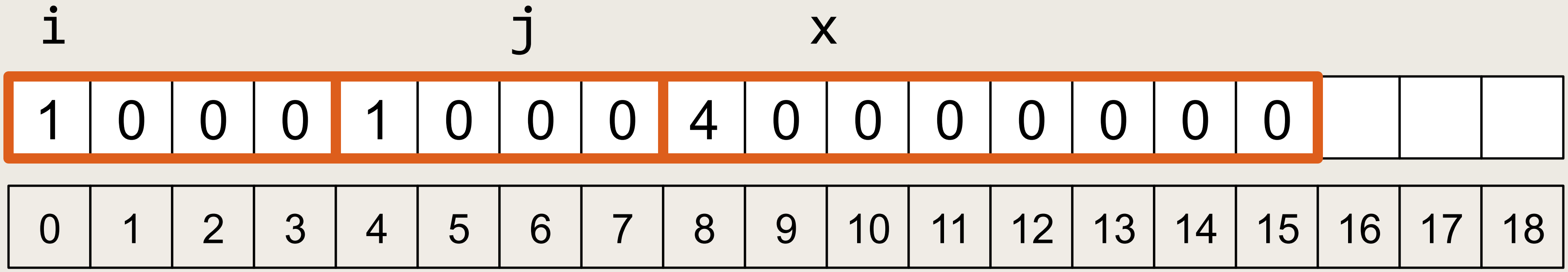
```
int main()
{
    int i,j;
    int *x; // x points to an integer
    i = 1;
    x = &i;
    j = *x;
    x = &j;
    (*x) = 3;
```



i				j				x										
1	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0			
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18

Pointers - 64 bit!

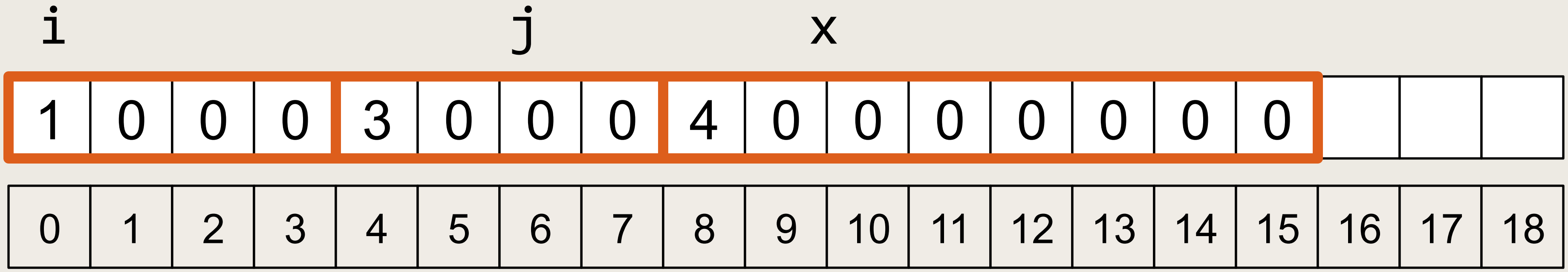
```
int main()
{
    int i,j;
    int *x; // x points to an integer
    i = 1;
    x = &i;
    j = *x;
    x = &j;
    (*x) = 3;
```



i						j			x									
1	0	0	0	1	0	0	0	4	0	0	0	0	0	0	0			
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18

Pointers - 64 bit!

```
int main()
{
    int i,j;
    int *x; // x points to an integer
    i = 1;
    x = &i;
    j = *x;
    x = &j;
➡   (*x) = 3;
```

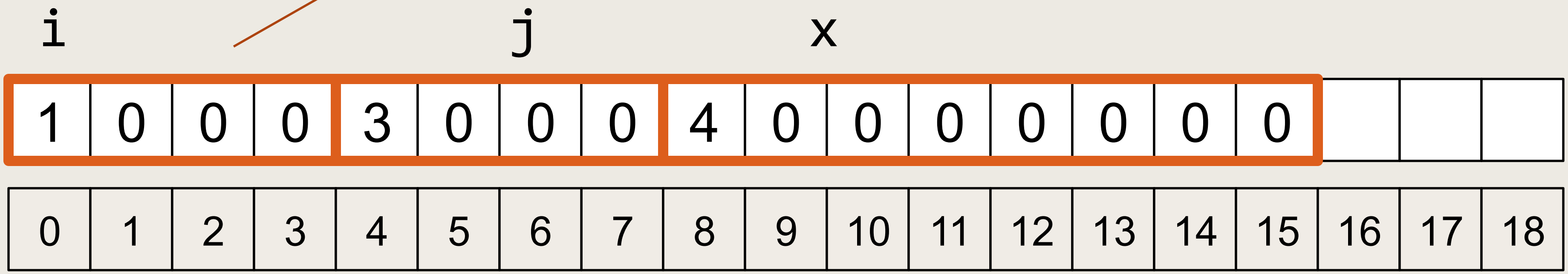


1	0	0	0	3	0	0	0	4	0	0	0	0	0	0	0			
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18

Pointers - 64 bit!

```
int main()
{
    int i,j;
    int *x; // x points to an integer
    i = 1;
    x = &i;
    j = *x;
    x = &j;
⇨   (*x) = 3;
```

X86 works in
Little Endian



i						j				x								
1	0	0	0	3	0	0	0	4	0	0	0	0	0	0	0			
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18

Example – the swap function

Does nothing

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

int main()
{
   int x, y;
   x = 3; y = 7;
   swap(x, y);
   // now x==?, y==?
}
```

Example – the swap function

Does nothing

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

int main()
{
   int x, y;
   x = 3; y = 7;
   swap(x, y);
   // now x==?, y==?
}
```

Works

```c
void swap(int *pa, int *pb)
{
   int temp = *pa;
   *pa = *pb;
   *pb = temp;
}

int main()
{
   int x, y;
   x = 3; y = 7;
   swap(&x, &y);
   // now x == ?, y == ?
}
```

NULL pointer

- Special value: uninitialized pointer or **null pointer**
- constant with a value of zero (defined in `<stdlib.h>`)
- It is always a good practice to assign a NULL value to a pointer variable in case you do not have an exact address to be assigned during variable declaration

```
int main()
{
   int *p = NULL;
   printf("The value of ptr is : %x\n", ptr );
   if( p != NULL )
   {

   }
...
}
```

Dereferencing NULL or uninitialized pointer

```
int *p = NULL;
*p = 1;
```

and also:

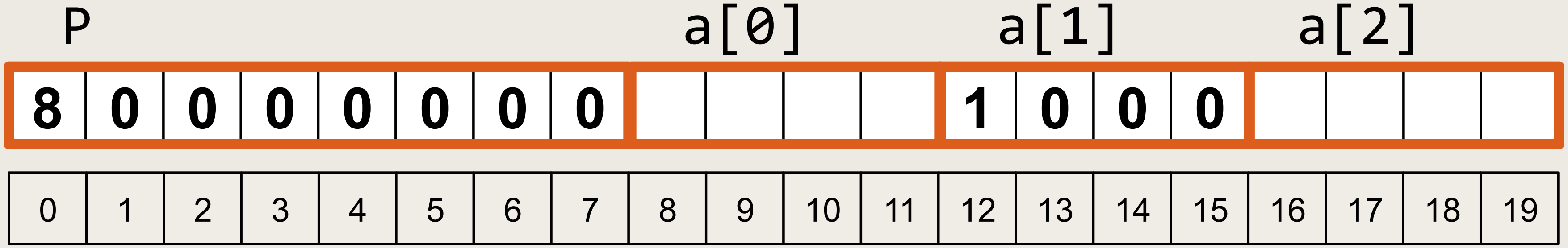
```
int *p;
*p = 1;
```

Will compile… but will (probably) lead to runtime error

Pointers & Arrays

Pointers & Arrays

```
int *p;
int a[3];
p = &a[0];  // same as p = a
*(p+1) = 1; // assignment to a[1]!
```



P								a[0]				a[1]				a[2]			
8	0	0	0	0	0	0	0					1	0	0	0				
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19

Pointers & Arrays

Array name can **sometimes** be treated as the address of the first member.

```
p = a;        // same as p = &a[0];

p[1] = 102;   // same as *(p+1)=102;
*(a+1) = 102; // same as prev. line

p++;          // p == a+1 == &a[1]

a = p;        // illegal
a++;          // illegal
```

Pointers & Arrays - size

Note:

```
int *p;
int a[4];
sizeof (p) == sizeof (void*)
sizeof (a) == 4 * sizeof (int)
```

→ Size of an array is known in compile time

→ Size of a pointer is always constant (no matter what it points to)

Pointers & Arrays

```c
int main()
{
   int arr[4] = {1,3,5,4};
   int i, sum = 0;
   for (i=0; i<sizeof(arr)/sizeof(arr[0]); ++i)
   {
      sum += arr[i];
   }
}
```

Passing pointers to functions

Passing Pointers & Arrays to functions

```
int foo( int *p );
int foo( int a[] );
int foo( int a[NUM] );
```

Are declaring **the same interface**:

In all cases, a ***pointer to int*** is being passed to the function foo

Passing Pointers & Arrays to functions

How about this code?

```c
int sum (int arr[])
{
    int i, sum = 0;
    for (i=0; i<sizeof(arr)/sizeof(arr[0]); ++i)
    {
        sum += arr[i];
    }
    return sum;
}
```

Passing Pointers & Arrays to functions

How about this code?

```
int sum (int arr[])
{
    int i, sum = 0;
    for (i=0; i<sizeof(arr)/sizeof(arr[0]); ++i)
    {
        sum += arr[i];
    }
    return sum;
}
```

Logical error:

```
sizeof (arr) ==
sizeof (int*) ==
sizeof (void*)
```

Passing Pointers & Arrays to functions

```
int sum (int arr[], int n)
{
    int i, sum = 0;
    for (i=0; i<n; ++i)
    {
        sum += arr[i]; // arr[i] = arr + i*sizeof(int)
    }
    return sum;
}
```

Array size must be passed as a parameter

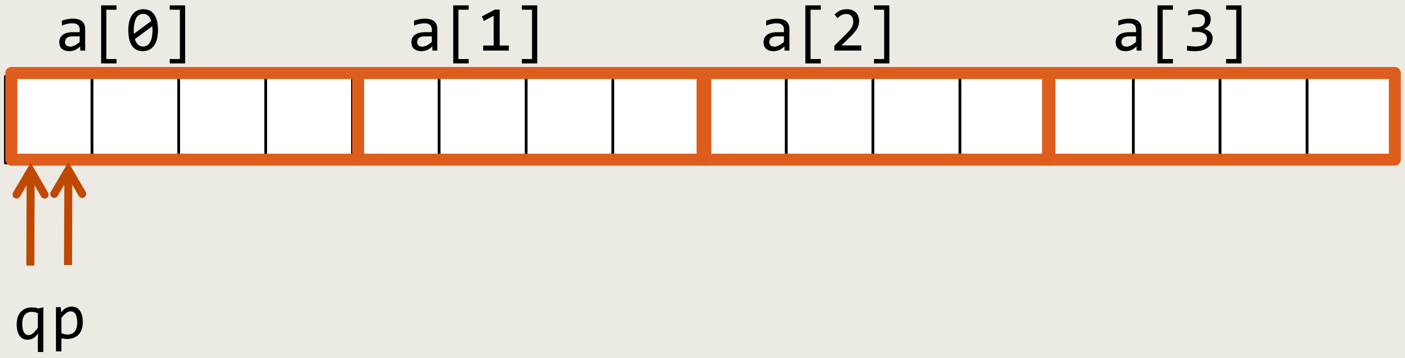
Pointer Arithmetic

Pointer Arithmetic

```
int a[3];
int *p = a;
char *q = (char *)a; // Explicit cast
// p and q point to the same location
p++;
q++;
```

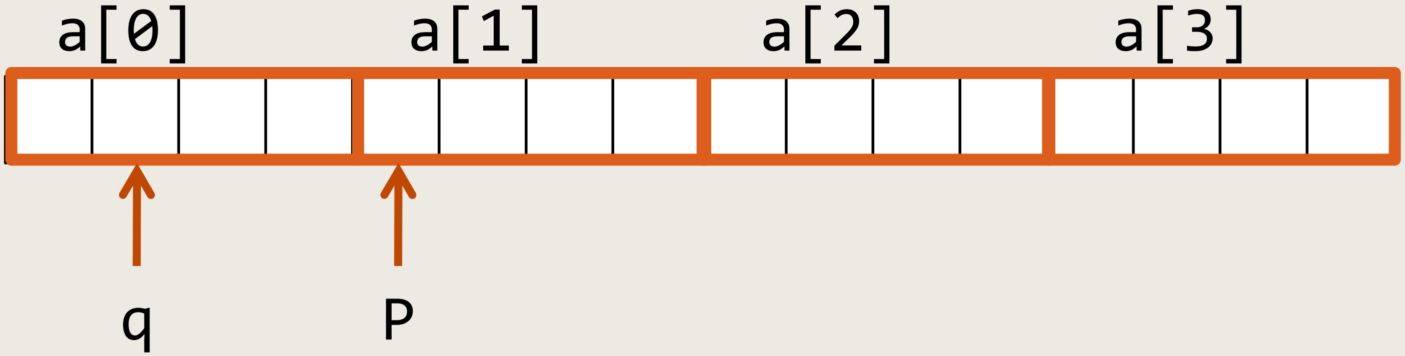
What is the difference?

```
p += sizeof(int);
q += sizeof(char);
```



Pointer Arithmetic

```
int a[3];
int *p = a;
char *q = (char *)a; // Explicit cast
// p and q point to the same location
p++; // increment p by 1 int (4 bytes)
q++; // increment q by 1 char (1 byte)
```



Pointer Arithmetic

```
int FindFirstNonZero( int a[], int n )
{
    int *p=a;
    for( ; a < p+n && (*a) == 0; a++ );
    return a-p;
}
```

```
int FindFirstNonZero( int a[], int n )
{
    int i;
    for( i = 0; i < n && a[i] == 0; i++ );
    return i;
}
```

Same - Preferable

Pointer Arithmetic

```
int a[4];
int *p = a;
long i = (long)a;
long j = (long)(a+1);   // adds 1*sizeof(int) to ‘a’
long dif = (long)(j-i);// dif = sizeof(int), not 1
```

Be careful:
Pointer arithmetic works just with pointers

```
int* p = 100;
int* q = 92;
printf("%d\n", p-q);  // 2
```

void *

`void *p` defines a pointer to undetermined type

```
int j;
int *p = &j;
void* q = p;  // no cast needed
p = (int*)q ; // cast is needed
```

All pointers can be casted one to the other, it may be useful sometimes, but beware…

void *

- No pointer arithmetic is defined for void*
  (gcc has an extension, treating the size of a void as 1)
- We cannot access the content of the pointer – dereferencing is not allowed

```
int j;
void *p = &j;
int k = *p;         // illegal
int k = (int)*p ; // still illegal
int k = *(int*)p; // legal
```