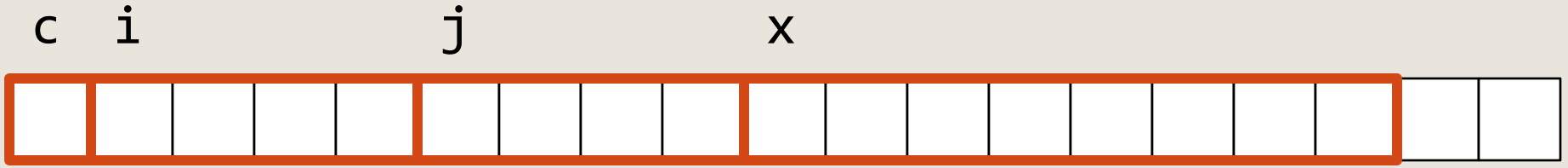
Memory, Arrays & Pointers

Memory

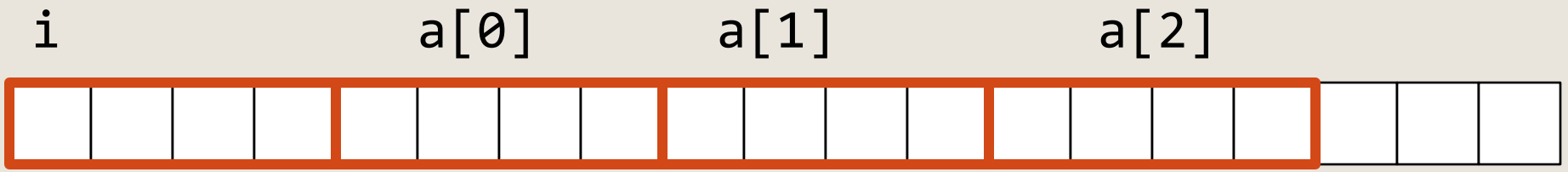
```
int main()
{
    char c;
    int i,j;
    double x;
```



Arrays

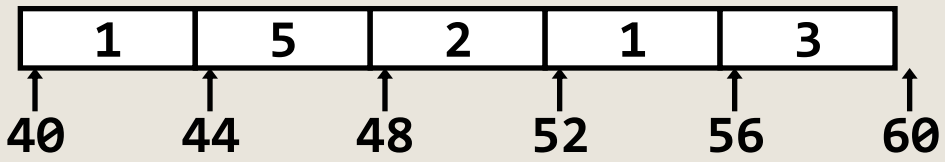
Defines a block of consecutive cells

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



Arrays - the [ ] operator

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



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 (no errors?!)

…but can lead to unpredictable results.

It is the programmer's responsibility to check whether the index is out of bounds…

Arrays

C does not provide array operations:

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

a = b; // illegal
if( a == b ) // legal. ==0, address
comparison.
```

Arrays

a[i] is just translated to (*(a+i)), thus this code will compile and run fine

```
int a[4];
0[a]= 42;// first element of array is 42
1[a]= -1;// second element of array is -1
```

Array Initialization

```
// Works, but IMHO in some cases, bad style
int arr[3] = {3, 4, 5};

// Good (robust to changes)
int arr[] = {3, 4, 5};

// Bad style - Init all items to 0 takes O(n)
int arr[3] = {0};

// Bad style - The last is 0
int arr[4] = {3, 4, 5};
```

Array Initialization

```
// Bad
int arr[2] = {3, 4, 5};

// Bad - array assignment
// only in initialization
int arr[3];
arr = {2,5,7};
```

Array Initialization – multidimensional (more on this later)

```
// Good, works same as arr[2][3]
int arr[][3] = {{2,5,7},{4,6,7}};

// Bad style, but same as above
int arr[2][3] = {2,5,7,4,6,7};

// Bad
int arr[3][2] = {{2,5,7},{4,6,7}};
```

Pointers

Data type for addresses

(almost) all you need to know is:

Pointers

- Declaration

  `<type> *p;  <type>* p;`

  Both, `p` points to objects of type `<type>`

- Pointer → value

  ```
  *p = x;
  y = *p;
  ```

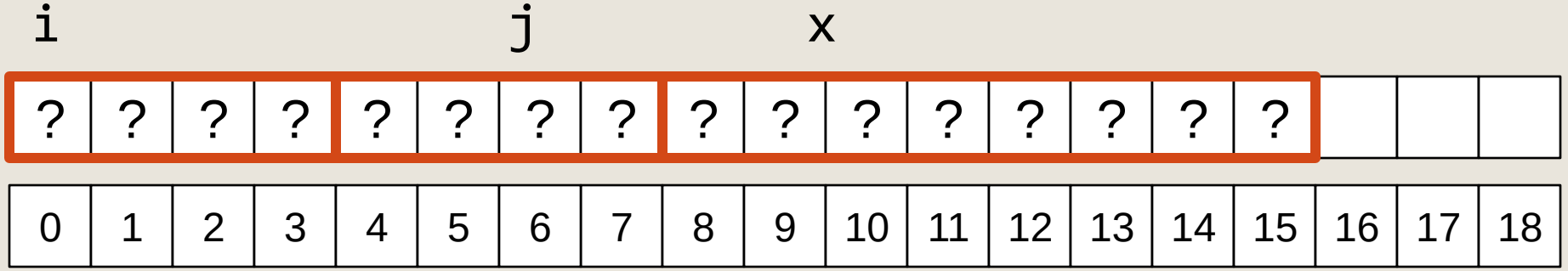
  `*p` refers to the object `p` points to

- Value → pointer

  `&x` - the pointer to `x(x's address)`

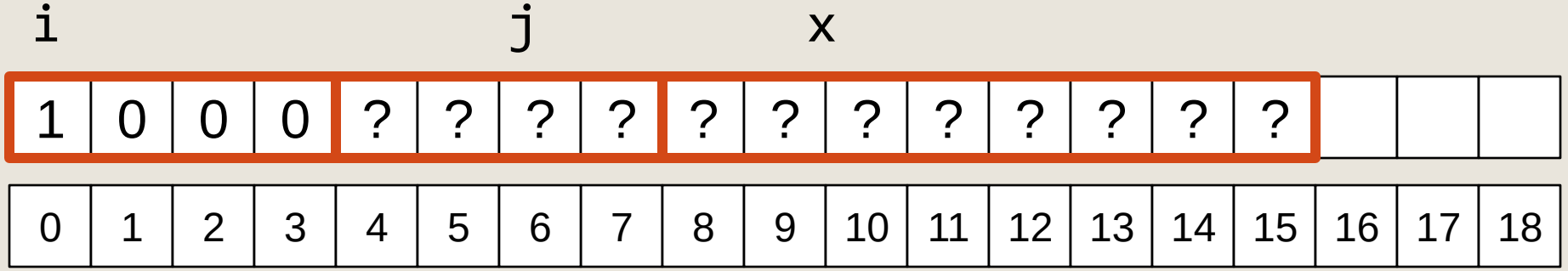
Pointers - 64 bit!

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



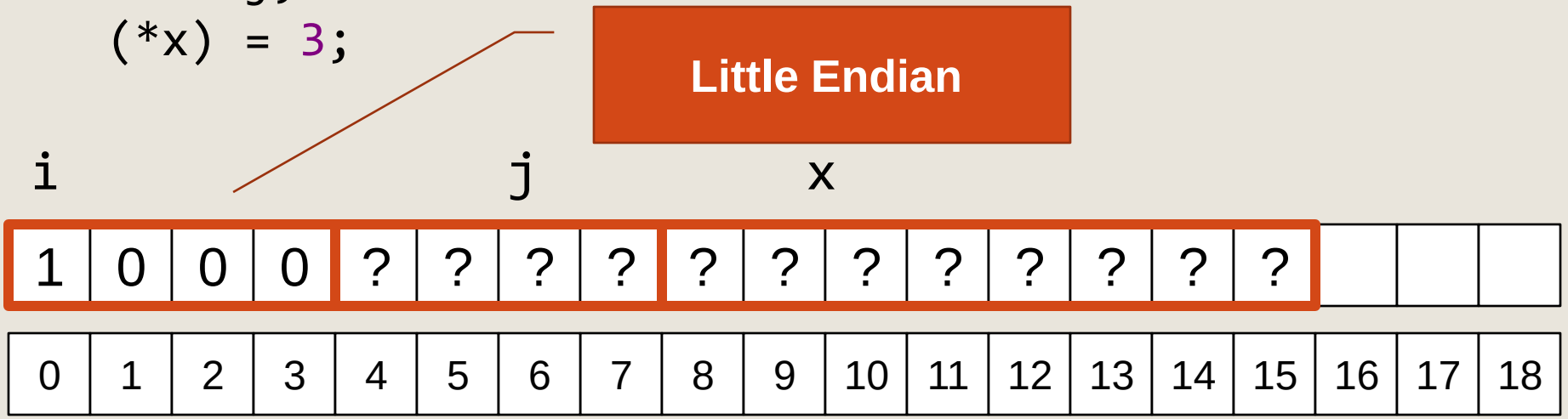
Pointers - 64 bit!

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



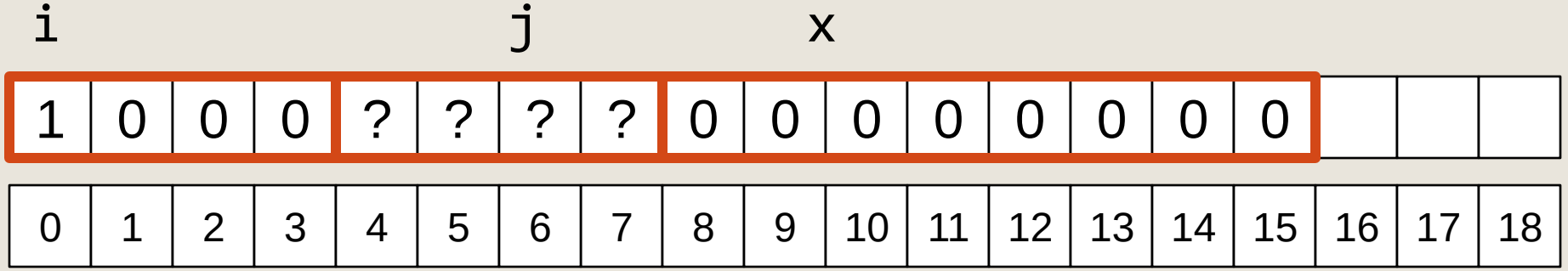
Pointers - 64 bit!

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



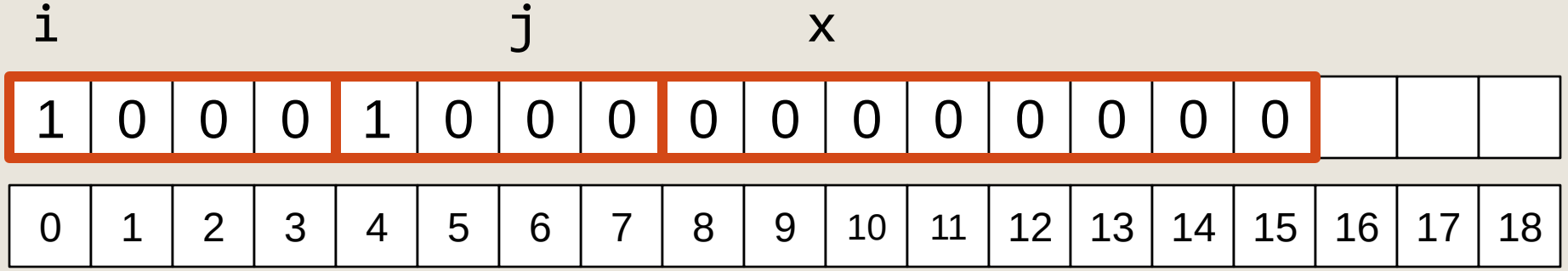
Pointers - 64 bit!

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



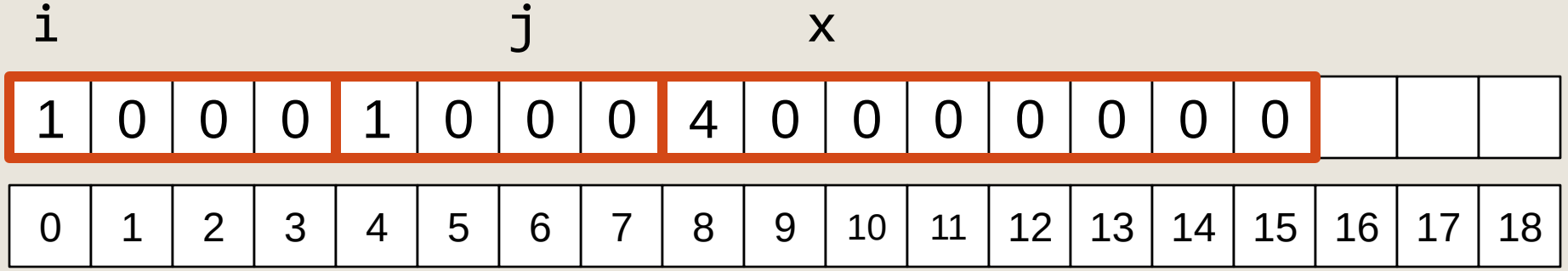
Pointers - 64 bit!

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



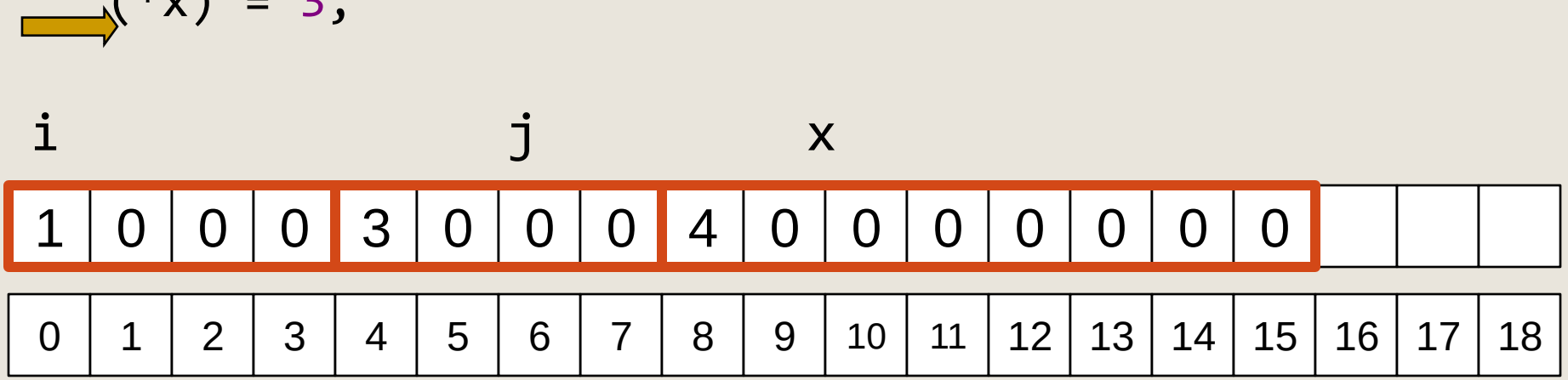
Pointers - 64 bit!

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



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;
```



Example – the swap function

Does nothing

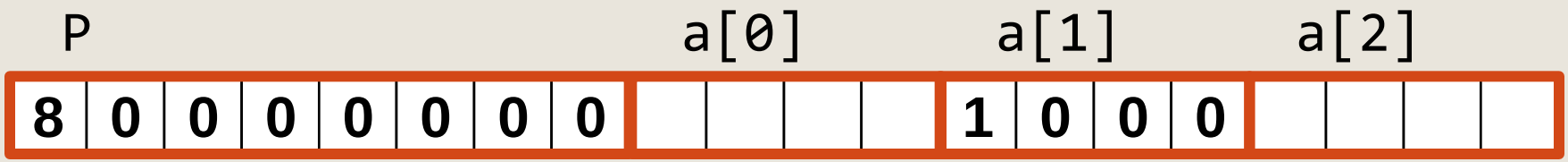
```
void swap(int a, int b)
{
   int temp = a;
   a = b;
   b = temp;
}
int main()
{
   int x, y;
   x = 3; y = 7;
   swap(x, y);
   // now x==3, y==7
```

Works

```
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);
   // x == 7, y == 3
```

Pointers & Arrays

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



Pointers & Arrays

Arrays are can **sometimes** be treated as address of the first member.

```
int *p;
int a[4];
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

But:

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

→ Size of the array is known in compile time

Pointers & Arrays

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

Pointers & Arrays

```
int foo( int *p );
```

and

```
int foo( int a[] );
```

Are declaring the same interface.

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

Pointers & Arrays

```
int foo( int *p );
```

and

```
int foo( int a[3] );
```

Are declaring the same interface.

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

Pointers & Arrays

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

int* ≡ arr[] !≡ arr[N]

Pointers & Arrays

```
int sum (int arr[42])
{
    int i, sum = 0;
    for (i=0; i<sizeof(arr)/sizeof(arr[0]); ++i)
    {
        sum += arr[i];
    }
    return sum;
}
// error: sizeof (arr) = sizeof (void*)
```

int* ≡ arr[] !≡ arr[N]

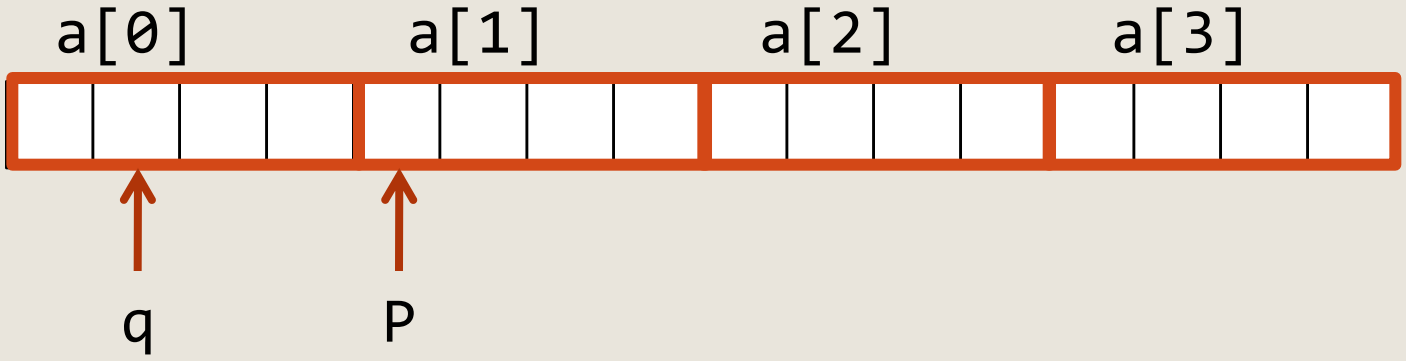
Pointers & Arrays

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

Array size must be passed as a parameter

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;
   while( (p<a+n) && ((*p) == 0) ) ++p;
   return p-a;
}
```

Same as

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

Preferable

Pointer Arithmetic

```
int a[4];
int *p= a;
long i= (long)a;
long j= (long)(a+1); // add 1*sizeof(int)
to a;
long dif= (long)(j-i); // dif= sizeof(int)
```

Be careful: Pointer arithmetic works just with pointers

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
```

void *

void* address jumps in bytes (like char*)

We cannot access to the content of the pointer

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

NULL pointer

Special value: points to “nothing” (defined in stdlib.h, usually as 0 of type void*):

```
int *p = NULL;


if( p != NULL )
{


}
```

NULL pointer

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

Will compile…

… but will lead to runtime error

Pointers to pointers

Pointer is a variable type, it also has an address:

```
int main()
{
    int n = 17;
    int *p = &n;
    int **p2 = &p;
    printf("the address of p2 is %p \n",&p2);
    printf("the address of p is %p \n",p2);
    printf("the address of n is %p \n",*p2);
    printf("the value of n is %d \n",**p2);
    return 0;
}
```

const

C’s “const”

C’s “const” is a qualifier that can be applied to the declaration of any variable to specify its value will not be changed.

C's "const"

Examples:

```
const double E = 2.71828;
E= 3.14;                    // compile error!
```

```
const int arr[] = {1,2};
arr[0] = 1;                 // compile error!
```

C's "const"

- C's "const" can be cast away

```
const int arr[] = {1,2};
int* arr_ptr = (int*)arr;

arr_ptr[0] = 3;      // compile ok!
                     // but might give a run-time error
```

- Helps to find errors.
- **Doesn't protect from evil changes !**

Const and Pointer's Syntax

Const protects his left side, unless there is nothing to his left and only then it protects his right side (examples next slide).

Const and Pointer's Syntax

```
// Both, cannot change the int pointed by p
// using p
const int * p = arr;
int const * p = arr;

// Cannot change the address stored in p
int * const p = arr;

// Cannot change the address stored in p
// and the int pointed by p using p
int const * const p = arr;
```

Const and Pointer's Syntax

```
// Both, cannot change the int pointed by p
// using p
const int * p = arr;
int const * p = arr;
```

Very Useful

```
// Cannot change the address stored in p
int * const p = arr;

// Cannot change the addres
// and the int pointed by p using p
int const * const p = arr;
```

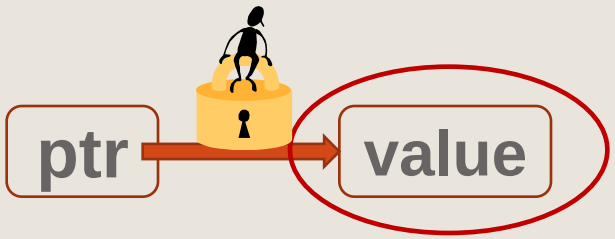
Never saw it being used

C's "const"

Do not confuse what the "const" declaration "protects"!

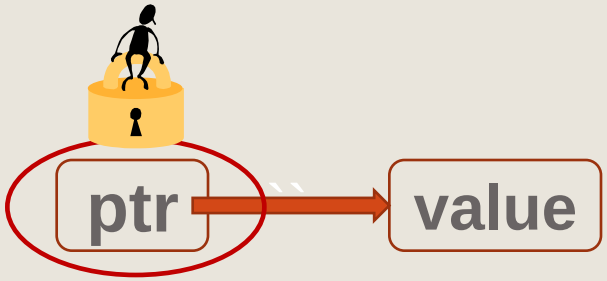
- A pointer to a **const variable**:

```
int arr[] = {1,2,3};
int const * p = arr;
p[1] = 1;              // illegal!
*(p+1) = 1;            // illegal!
p = NULL;              // legal
```



- A **const pointer** to a variable:

```
int arr[] = {1,2,3};
int* const const_p = arr;
const_p[1] = 0;      // legal!
const_p = NULL;      // illegal!
```



C's "const"

How about *arr* itself?

```
int arr[] = {1,2,3};
int* const const_p = arr;
const_p[1] = 0;      // legal!
const_p = NULL;      // illegal!

int *p;
arr=p;               // compile error!
```

Const and User Defined Types

```
typedef struct Complex
{
   int _real, _img;
   int *_imgP;
} Complex;

Complex const COMP1 = comp2;  // ok, init. using comp2
Complex const COMP3 = {3,2,&someInt};  // ok, copying values

COMP1._img = 3;                  // illegal!
COMP3._imgP = &someOtherInt;  // illegal!
*(COMP3._imgP) = 5;  // legal!
```

All the members of a const variable are immutable!

Compare to Java's "final"

- All (methods as well as data) are Class members.

```
final int LIMIT = 10;
LIMIT = 11;                                  // illegal!

final MyObject obj1 = new MyObject();
MyObject obj2 = null;
MyObject obj3 = new MyObject();
obj2 = obj1; // fine, both point now to the same object
obj1 = obj3;                                 // illegal!
obj1.setSomeValue(5);                        // legal!
```

- "final" makes primitive types constants and references to objects constant.
- The values inside the referred final objects are not necessarily constant !

“Const” Usage

The const declaration can (and should!) be used in the definition of a function’s arguments, to indicate it would not change them:

```
int strlen(const char []);
```

Why use? (This is not a recommendation but a **must**)

- clearer code
- avoids errors
- part of the interfaces you define!
- can be used with const objects

More of “const” meaning and usage in C++

C strings manipulation

```
#include <string.h>
```

strcmp
strcpy
strlen
…
http://www.cplusplus.com/reference/cstring/

C Strings

Strings

Java:

- Char: is 2 bytes (Unicode)
- String: an object which behaves as a primitive type (an immutable object, passed by value)

C:

- char: usually 1 byte. An Integer.
- string: an array of characters.

```
char* txt1 = "text";

char txt2[] = "text";
char txt3[] =
{'t','e','x','t','\0'};
```

C Strings

Strings are always terminated by a *null character,* (a character with integer value 0).

```
char* text = "string";
// means text[5] = g and text[6] = \0
// 7 chars are allocated!
```

There is no way to enforce it automatically when you create your own strings, so:

- remember it's there
- allocate memory for it
- specify it when you initialize char by char

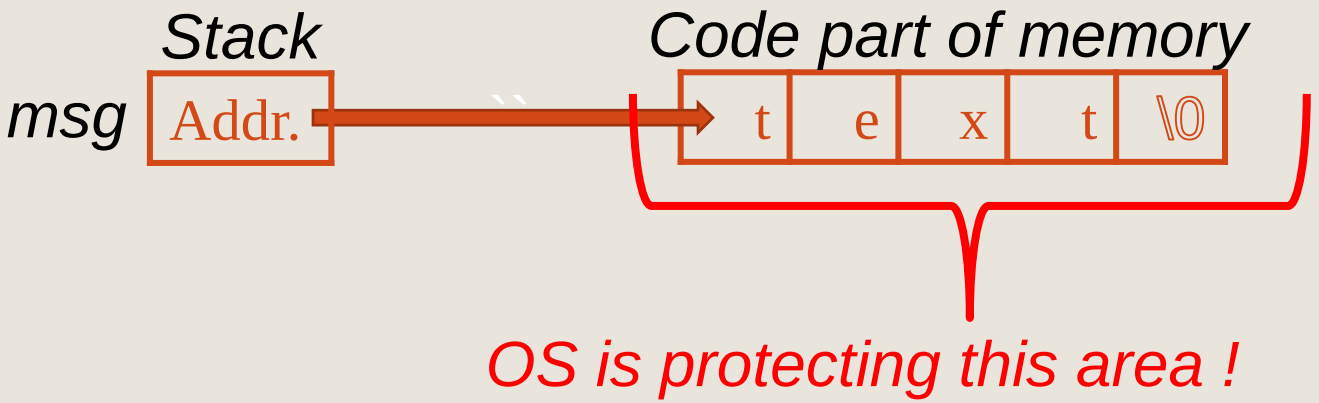
C string literals ("")

When working with char*, C string literals ("") are written in the **code segment** (part) of the memory.
Thus, you **can't change them**!

C string literals (" ")

When working with char*, C string literals (" ") are written in the **code segment** (part) of the memory.
Thus, you **can't change them**!

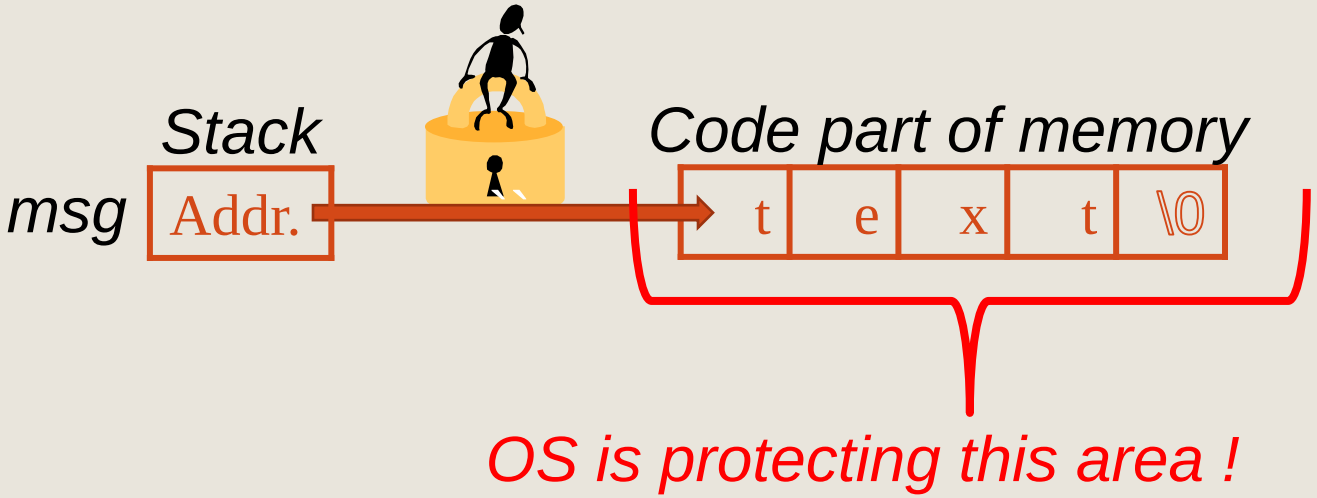
```
char* msg = "text";
msg[0] = ‘w’; // seg fault! – error caught only in runtime !
```



C string literals ("") with const

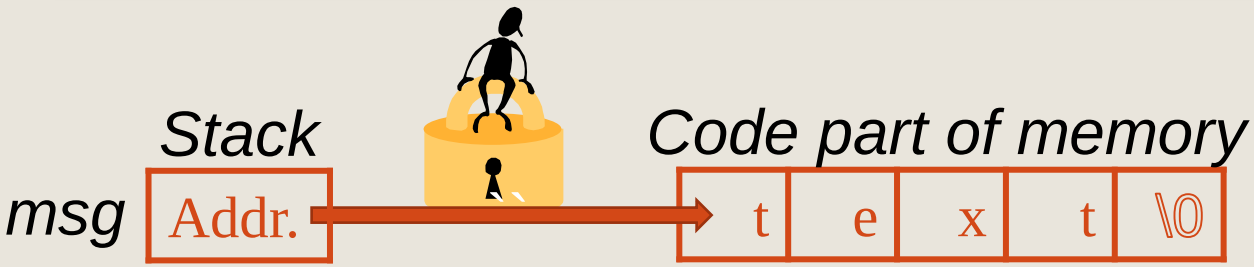
So, what we do is:

```
const char* msg = "text";
msg[0] = ‘t’;          // compile error!
                       // better!
```



Difference between initialzing a pointer and an array with C string literals (" ")

```
char* msg = "text";
// msg is a pointer that points to a memory
that is in the code part
```




```
char msg2[] = "text";
// msg2 is an array of chars that
are on the stack
```

Difference between initialzing a pointer and an array with C string literals ("")

```
char* msg = "text";
msg[0]= 'n';      // seg fault - trying to change
                     what is written in the code
                     part of the memory




char msg2[] = "text";
msg2[0]= 'n';     // ok - changing what is written
                     in the stack part of the memory
```

C Strings Examples

```
char txt1[] = "text";
char* txt2 = "text";

int i = strlen(txt1); // i = 4, same for strlen(txt2)

txt1[0] = ’n’;        // now txt1="next“
*txt2 = ’n’;          // illegal! “text” is in the code
                      // segment
txt2 = txt1;          // legal. now txt2 points to the
                      // same string.
txt1 = txt2;          // illegal!

if (! (strcmp(txt2,"next")) //This condition is now true
{
  ...
```

C Strings Manipulation

To manipulate a single character use the functions defined in *ctype.h*

```
#include <ctype.h>
char c = ‘A’;
isalpha(c); isupper(c); islower(c);   …
```

Manipulation of Strings is done by including the *string.h* header file

```
// copy a string
char* strcpy(char * dest, const char* src);
// append a string
char* strcat(char * dest, const char* src);
```

C Strings Manipulation (2)

```
// compare two strings.
// when str1 < str2 lexicographically return < 0
// when str1 > str2 lexicographically return > 0
// when identical return 0
int strcmp(const char * str1, const char* str2);

// return strings length, not including the \0!!
size_t strlen(const char * str);

// Other functions:
strncpy(),strncat(),strncmp()   …
```

NOTE :

- All C library functions assumes the usages of ‘\0’ and enough storage space.
- No boundary checks! You are responsible.
- http://www.cplusplus.com/reference/cstring/

C Strings Functions

An “array” version of strcpy():

```
void strcpy(char * dest, const char* src)
{
   int i = 0;
   while ((dest[i] = src[i])!= '\0')) i++;
}
```

C Strings Functions

A “pointers” version of strcpy():

```
void strcpy(char * dest, const char* src)
{
   while ((*dest = *src)!= '\0'))
   {
      dest++;
      src++;
   }
}
```

C Strings Functions (2)

A shorter version::

```
void strcpy(char * dest,const char* src)
{
   while ((*dest++ = *src++)!= '\0'));
}
```

Actually the comparison against \0 is redundant:

```
void strcpy(char * dest,const char* src)
{
   while (*dest++ = *src++);
}
```

Style note: Unlike K&R book, **we do NOT encourage you to write such code**. However, you should be able to read and understand it.