



京都先端科学大学
KYOTO UNIVERSITY OF ADVANCED SCIENCE

京都先端科学大学

京都發世界人財

Information Processing 2 09

memory, addresses, pointers, indirection
equivalence of arrays and pointers
address arithmetic

Ian Piumarta

Faculty of Engineering, KUAS

last week: program structure, multiple source files

declarations in header files

compilation from multiple files

`static` variables

block structure

defining variables within blocks

variable initialisation

recursive functions

`qsort()`, `swap()`

`#include`, `#define`, `#undef`, conditional compilation

4.4 Scope Rules

4.5 Header Files

4.6 Static Variables

4.7 Register Variables

4.8 Block Structure

4.9 Initialization

4.10 Recursion

4.11 The C Preprocessor

this week: memory, addresses, arrays and pointers

memory, addresses, variables, pointers, indirection

`swap(a, b)` that actually works

`getint()`

equivalence of arrays, elements, and pointers

`strlen()` using pointers

passing sub-arrays as arguments

negative indices; address arithmetic

`alloc()` and `afree()`

char array vs. char pointers vs. strings

versions of `strcpy()`, `strcmp()`

Chapter 5. Pointers and Arrays

5.1 Pointers and Addresses

5.2 Pointers and Function Arguments

5.3 Pointers and Arrays

5.4 Address Arithmetic

5.5 Character Pointers and Functions

review

dangling else

`continue` and `break` in loops and `switch`

function declaration, definition, prototype

variable declaration, definition

stacks, LIFO

`static` variables

scope of parameters, locals

macro safety

test

pointers and addresses

programs manipulate values

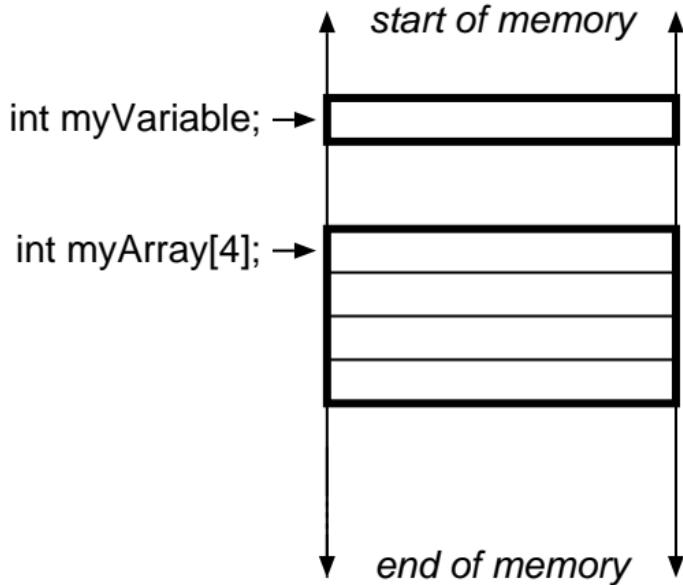
variables and arrays are one kind of value

all these values have to be stored somewhere

⇒ the computer's *memory*

other (invisible) values must be stored as the program runs

- e.g, the location where control should return at the end of a function
- and there can be a long list of these, when functions calls are deeply nested



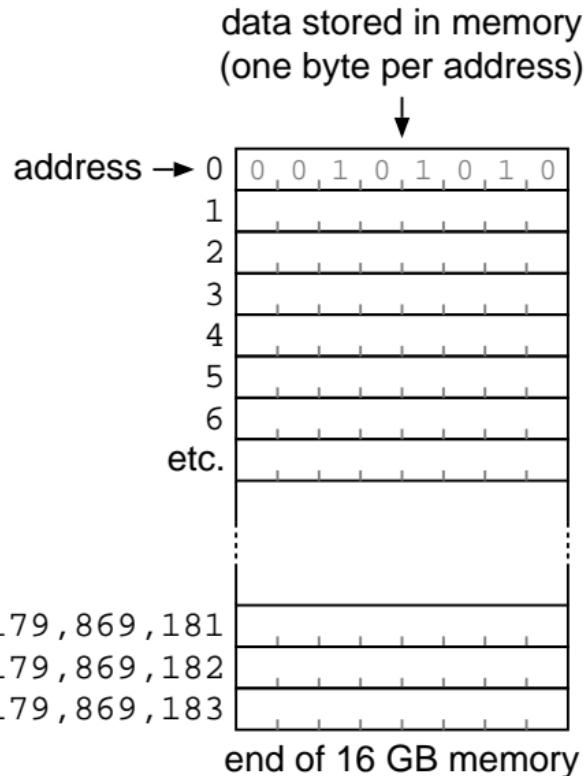
computer memory organisation

memory is a (huge!) array of bytes

each byte in memory has a unique *address*, which works exactly like an array index

the addresses are consecutive

adding 1 to any address gives the address of the next byte in the memory



computer memory organisation

the compiler divides memory into small blocks of storage for variables

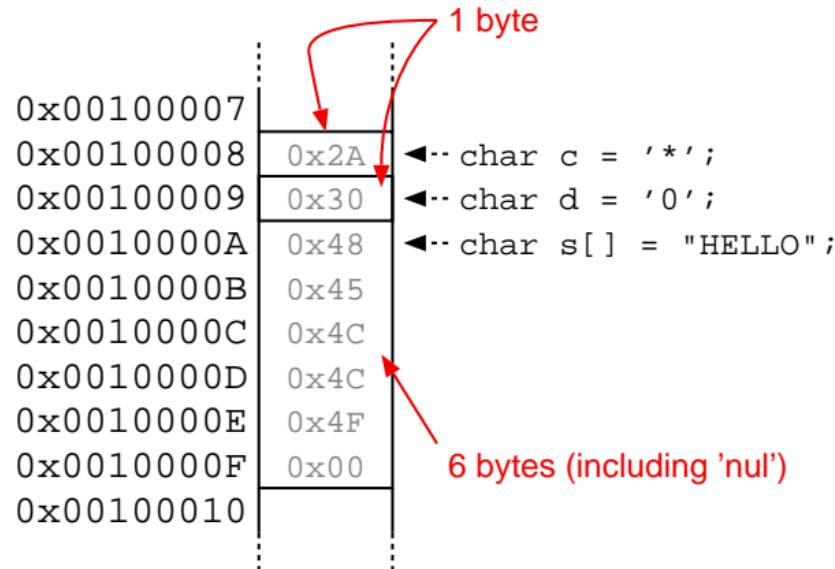
a block is typically 1, 2, 4, or 8 bytes long

- the same as the size of the type of the value stored there

`char` values therefore occupy 1 byte in memory

consecutive `char`s have addresses that differ by 1

an array of `char` occupies as many bytes of memory as it has elements



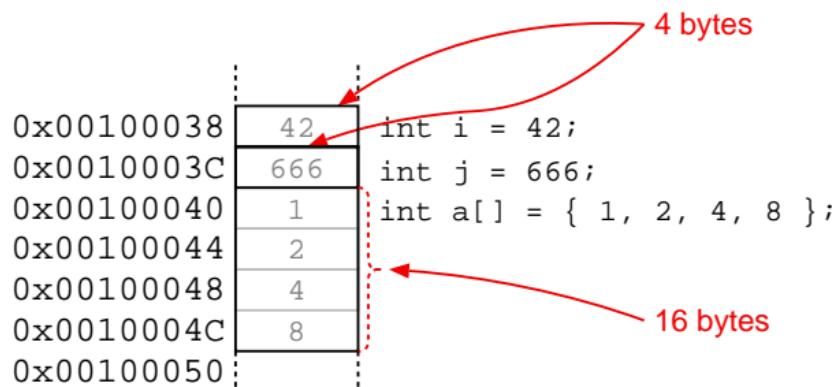
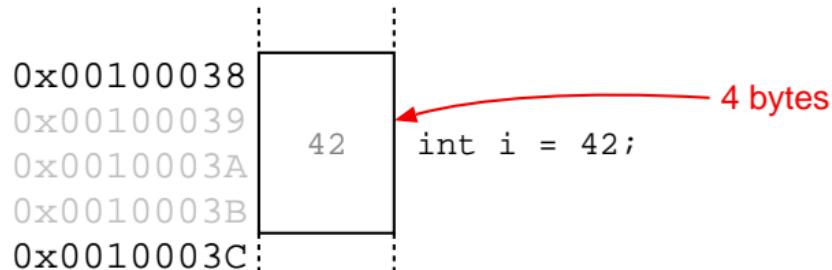
computer memory organisation

a 32-bit `int` occupies four bytes of memory

consecutive `int`s have addresses that differ by 4

an array of `int` occupies 4 times as many bytes of memory as it has elements

similarly for:
 `short` 2 bytes per value
 `long` 8 bytes
 `float` 4 bytes
 `double` 8 bytes



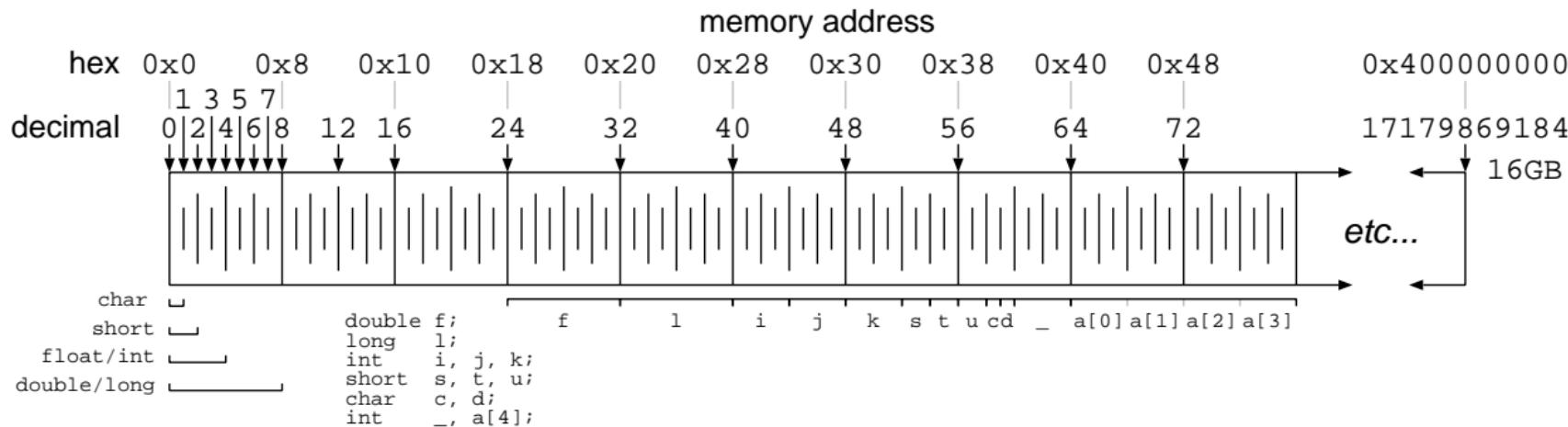
computer memory organisation

addresses can be written in any base but usually hexadecimal is used

- decimal is not so convenient because types have sizes that are powers of 2

octal can be used, but hex is more convenient

- because 32- and 64-bit addresses contain an exact number of hex digits



addresses and pointers

consider a `char` variable `c` at memory address `0x1004`

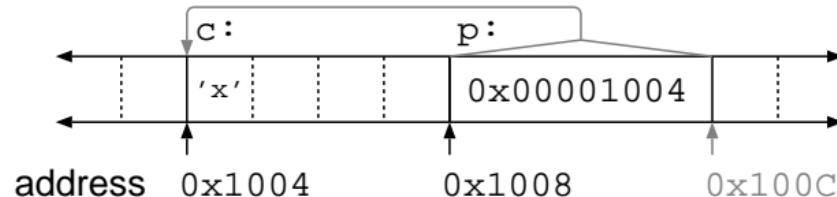
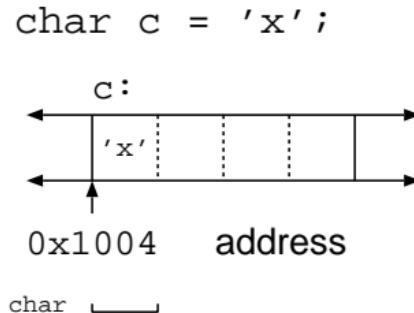
the value `0x1004` is the *address of* the variable `c`

the operator `&` gives you the address of a variable

⇒ the address of `c` is written '`&c`' and has value `0x1004`

you can store this value in another variable, e.g: `p = &c;`

- `p` is a *pointer variable* that now 'points to' (contains the address of) `c`



printing addresses using printf()

the `printf()` function can print addresses using the `%p` conversion specifier

```
char    a, b, c;
short   s, t, u;
int     i, j, k;
double  d, e, f;

int main() {
    printf("%p %p %p\n", &a, &b, &c);    // 0x1042d0000 0x1042d0001 0x1042d0002
    printf("%p %p %p\n", &s, &t, &u);    // 0x1042d002c 0x1042d002e 0x1042d0030
    printf("%p %p %p\n", &i, &j, &k);    // 0x1042d0020 0x1042d0024 0x1042d0028
    printf("%p %p %p\n", &d, &e, &f);    // 0x1042d0008 0x1042d0010 0x1042d0018
    return 0;
}
```

note the addresses increasing by the sizes of the types stored there

note that the order in memory is **not predictable** from the order of the declarations:

- the `doubles` come between the `chars` and `ints`, and the `shorts` are last

accessing a value using its address

the *dereference* operator * can be applied to an address

- the result is gives you the thing that the pointer points to
 - p contains a *reference to* (the address of) c
 - *p is ‘the thing that p points to’, i.e., the variable c itself
- ⇒ *p and c both refer to the same memory location, and thus the same value 'x'

the type of c is char, so the type of p is ‘pointer to char’

```
char    c = 'x';  
char *p;           // declare p as ‘pointer to char’  
p = &c;           // p points to (contains the address of) c  
printf("%c\n", c); // prints x  
*p = 'y';         // store a new value at the address in p (i.e., in c)  
printf("%c\n", c); // prints y
```

note the declaration of p: writing ‘*p’ in your program produces a value of type ‘char’, hence ‘char *p;’

pointer declarations

the declaration of `c` is familiar:

```
char c = 'x';
```

the declaration of `pc` is new:

```
char *pc = &c;
```

declarations in C are written in a way that shows *how they are used* in an expression

'`char *pc`;' declares that using '`*pc`' in an expression will result in a `char` value

all C declarations work in exactly the same way; some examples:

<i>declaration</i>	<i>expression</i>	<i>expression result type</i>
<code>char c;</code>	<code>c</code>	<code>char</code>
<code>short *s;</code>	<code>*s</code>	<code>short</code>
<code>int a[8];</code>	<code>a[i]</code>	<code>int</code>
<code>float *q[8];</code>	<code>*q[i]</code>	<code>float</code>
<code>long *q[8];</code>	<code>q[i]</code>	<code>long *</code>
<code>double atof(char *);</code>	<code>atof("3.14")</code>	<code>double</code>

using pointers in expressions

after the declarations

```
int i = 42, *pi = &i;
```

you can use `*pi` anywhere you would use `i` (assuming you don't modify `pi`!)

the following expressions are all equivalent

```
i      = i + 10;  
*pi    = *pi + 10;  
*pi += 10;
```

beware: unary operators all have the same precedence and associate right-to-left

```
++*pi;        // increment i (* then ++)  
(*pi)++;    // increment i (* then ++)  
*pi++;       // increment pi (++) then fetch i (*)
```

(beware: the last example is very often **not** the intended result!)

pointers are values too

pointers are ‘scalar’ values¹ just like numbers²

you can assign to pointer variables, pass pointers values as arguments, compare them, perform some kinds of arithmetic on them, etc.

```
int *pi = &i;      // pi points to i
int *pj = &j;      // pj points to j

// swap pi and pj
{
    int *tmp = pi;
    pi = pj;      // pi now points to j
    pj = tmp;      // pj now points to i
}                      // did the values of i and j change?
```

¹a scalar value is a single, atomic unit of data that holds one value at a time (unlike, e.g., an array)

²at the machine level, pointers are *integers* (wide enough to represent any memory address)

pointers and function arguments

function arguments are passed by value

- *parameters* are local variables containing the *argument values*

the function cannot directly alter a variable used by the caller as an *argument*

```
void swap(int x, int y)
{
    int t = x;
    x = y;
    y = t;
}

...
swap(i, j); // does not work!
```

pointers and function arguments

to allow a function to modify a variable's value

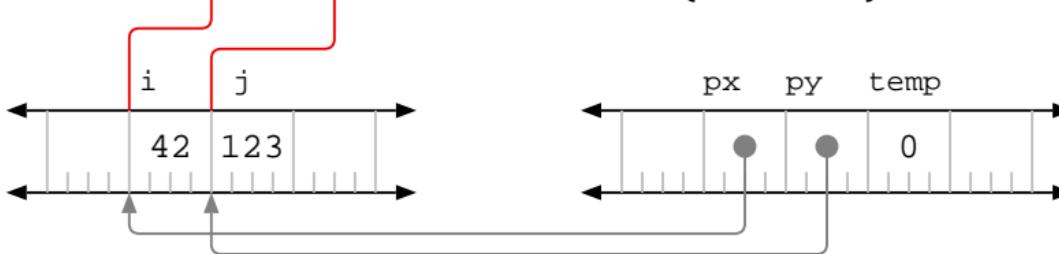
- pass the *address* of the variable as the argument
- use the pointer to access *indirectly* the contents of the variable

```
// swap integers *px and *py (their addresses are px and py)
void swap(int *px, int *py)
{
    int temp = *px;      // make a copy of the integer stored at address px
    *px = *py;          // move integer at address py to integer at address px
    *py = temp;         // set integer at py to the original value stored at px
}

int main() {
    int i = 42, j = 123;
    printf("%d %d\n", i, j);    // => 42 123
    swap(&i, &j);            // addresses of the variables
    printf("%d %d\n", i, j);    // => 123 42
    ...
}
```

pointers and function arguments

```
int i = 42;  
int j = 123;  
swap(&i, &j);  
void swap(  
    int *px,  
    int *py  
) { ... }
```



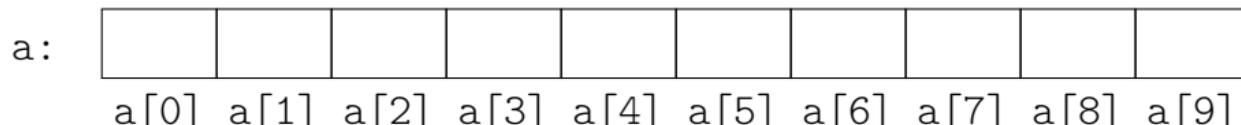
```
void swap(int *px, int *py) {  
    int temp = *px; // fetch i, assign to temp  
    *px = *py; // fetch j, assign to i  
    *py = temp; // fetch temp, assign to j  
}
```

pointers and arrays

consider an array of integers

```
int a[10];
```

a is a *constant* equal to the location in memory where the array begins



the location in memory where a begins is the same as the address of a[0]

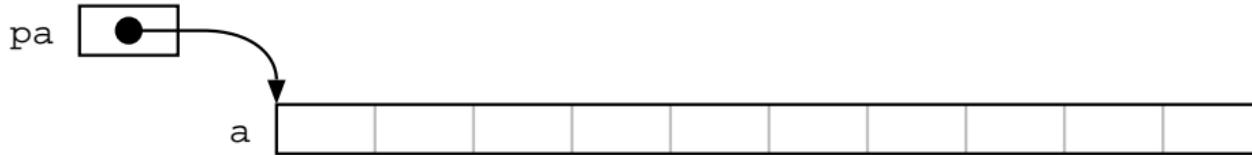
we can therefore write

```
int *pa = &a[0]; // a[0] is an 'int' ⇒ &a[0] is an 'int *'
```

pa is now a *variable* whose value is the location in memory where a begins



pointers and arrays



the assignment

```
int x = *pa;
```

will copy the value stored in `a[0]` into `x`

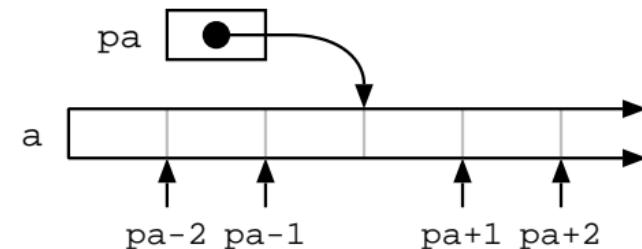
by definition `pa+1` is the address of the element `a[1]`

if `pa` points to any element in `a` then

`pa+1` points to the element after `pa`

`pa-1` points to the element before `pa`

so if `pa = &a[0]` then `*(<code>pa+1)` refers to `a[1]`
and `*(<code>pa+i)` refers to `a[i]`

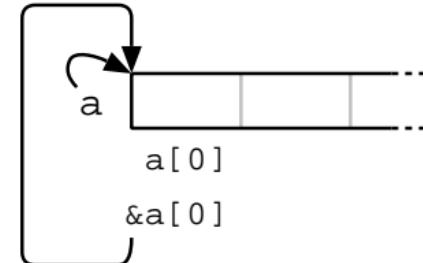


pointers and arrays

by definition, the value of an array in an expression is the address of its first element
in other words, `a == &a[0]`

the following assignments are therefore equivalent

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



it also follows that `a+i == &a[0]+i == &a[0+i] == &a[i]`

in fact, when you write `a[i]` the compiler treats it as `*(a+i)`

you can therefore also write `*(pa+i)` as `pa[i]` (or even `i[pa]` !!!)

in practice, the *only* difference between a pointer and an array is this:

- a pointer is a *variable*; you can assign to it: `pa = pa+1; *pa++ = 42;` etc.
- an array is a *constant*; you cannot modify it: `a = pa; a++;` are both **illegal**
 - except when the array is a function parameter (then it is a pointer *variable*)

pointers, arrays, and function parameters

function parameters are always variables

= local variables initialised with argument values during the function call

when declared as an array, a function parameter is really a pointer variable

consider a function to find the length of a string

```
int strlength(char s[]) {  
    int n;  
    for (n = 0; s[n] ; ++n) ;  
    return n;  
}
```

which can be called with a string constant, an array of `char`, or a pointer to `char`

```
strlength("hello");      // string constant  
strlength(array);       // char array[] = "hello";  
strlength(pchars);      // char *pchars = "hello";
```

pointers, arrays, and function parameters

the same function could be written using a pointer

```
int strlength(char *s) { // char s[] ≡ char *s
    int n;
    for (n = 0; *s != '\0'; ++s)
        ++n;
    return n;
}
```

(declaring the parameter `s` as an array does not stop it being a variable)

which can still be called with any of these arguments

```
strlength("hello");      // string constant
strlength(array);        // char array[] = "hello";
strlength(pchars);       // char *pchars = "hello";
```

(incrementing `s` has no affect on the argument because `s` is a local variable)

pointers, arrays, and function parameters

as a formal parameter declaration, `char s[]` and `char *s` are equivalent

- use whichever notation seems most appropriate for the function body

because `s` is a pointer variable, any compatible pointer is valid as an argument

e.g., you can pass a pointer to the middle of an array as the argument

the following function calls are both valid

```
int array[100];  
f(&array[5]);  
f(array+5);
```

regardless of whether `f` is defined as `f(int a[])` `{...}` or as `f(int *a)` `{...}`

address arithmetic

consider an array a and two pointers, p and q

```
int a[100], *p = a+10, *q = a + 15;
```

an integer can be added to any of the pointers:

- adding an integer i to a produces $a+i == \&a[i]$
- adding an integer i to p produces the address of $p+i == a+10+i == \&a[10+i]$
- adding an integer i to q produces the address of $q+i == a+15+i == \&a[15+i]$

note that: $p+5 == a+10+5 == a+15 == q$

this leads to a natural definition of pointer subtraction: $p+5 == q \Rightarrow q-p == 5$

subtracting two pointers gives *the number of elements between them*

- but the pointers *must* point into the *same* array
- pointer subtraction is *undefined* for pointers into different arrays

relational operators also work between pointers into the *same* array: $a < p \&& p < q$

character pointers and functions

a string constant is an array of characters (with terminating '\0' character)

in an expression, the value of a string constant is the address of its first element

the type of a string constant is therefore 'pointer to character' or 'char *'

```
extern int printf(char *format, ...);  
printf("hello, world\n");  
char *message = "hello, world\n";  
printf(message);  
  
char bye[] = "goodbye\n";  
message = bye;  
printf(message);
```

modifying character pointers and arrays

character arrays and character pointers have an important difference

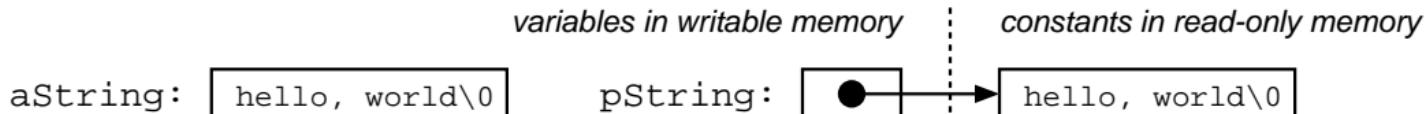
```
char aString[] = "hello, world"; // an array  
char *pString = "hello, world"; // a pointer
```

aString is an array of `char` large enough for the string, including terminating null

- individual characters within aString can be modified
- aString itself cannot be modified, it is a *constant*
- aString will always point to the storage reserved for the array

pString is a pointer variable initialised with the address of a string **constant**

- individual characters in the string *constant* that it points to cannot be modified
- pString can subsequently be modified to point to a different location in memory
- if that memory is an array, then the contents can be modified via pString



useful idiomatic statements involving pointers

find the end of a string pointed to by `char *s`

```
while (*s) ++s;
```

copy a string from `char *t` to `char *s`

```
while ((*s++ = *t++)) ;
```

push an integer value onto a stack using a stack pointer `int *sp`

```
*sp++ = value;
```

pop an integer value off a stack using a stack pointer `int *sp`

```
value = *--sp;
```

next week: more arrays and pointers

arrays of strings

sorting lines of text from the input

modified `qsort()` that sorts strings

`day_of_year()` and `month_day()`

`month_name()` and initialising array of strings

`echo` program

find command line string in text lines

`sort` using function pointers

 to parameterise behaviour

many complex declarations

`dcl` converts declarations to English

`undcl` converts English to declaration

5.6 Pointer Arrays; Pointers to Pointers

5.7 Multi-dimensional Arrays

5.8 Initialization of Pointer Arrays

5.9 Pointers vs. Multi-dimensional Arrays

5.10 Command-line Arguments

5.11 Pointers to Functions

5.12 Complicated Declarations

assignment

please download assignment from
MS Team “Information Processing 2”, “General” channel

exercises

using pointers to strings

pointer arithmetic

copying strings with pointers

combining string operations

longest of two strings

swapping the contents of two strings

reversing a string using pointers

reversing a sub-string using pointers

finding words and reversing sub-strings

encoding and decoding encrypted messages

reversing the order of lines in a file

12:40 10 test
12:50 5 last week: program structure, multiple files
12:55 5 this week: memory addresses, arrays, pointers
13:00 5 pointers and addresses
13:05 5 computer memory organisation
13:10 5 addresses and pointers
13:15 5 accessing a value using its address
13:20 5 pointer declarations
13:25 5 using pointers in expressions
13:30 5 pointers are values too
13:35 5 pointers and function arguments
13:40 5 pointers and arrays
13:45 5 pointers, array, and function parameters
13:50 5 pointer (address) arithmetic
13:55 5 character pointers and functions
14:00 5 modifying character arrays and pointers
14:05 5 useful idiomatic pointer expressions and statements

14:10 10 *break*

14:20 90 exercises
15:50 00 end