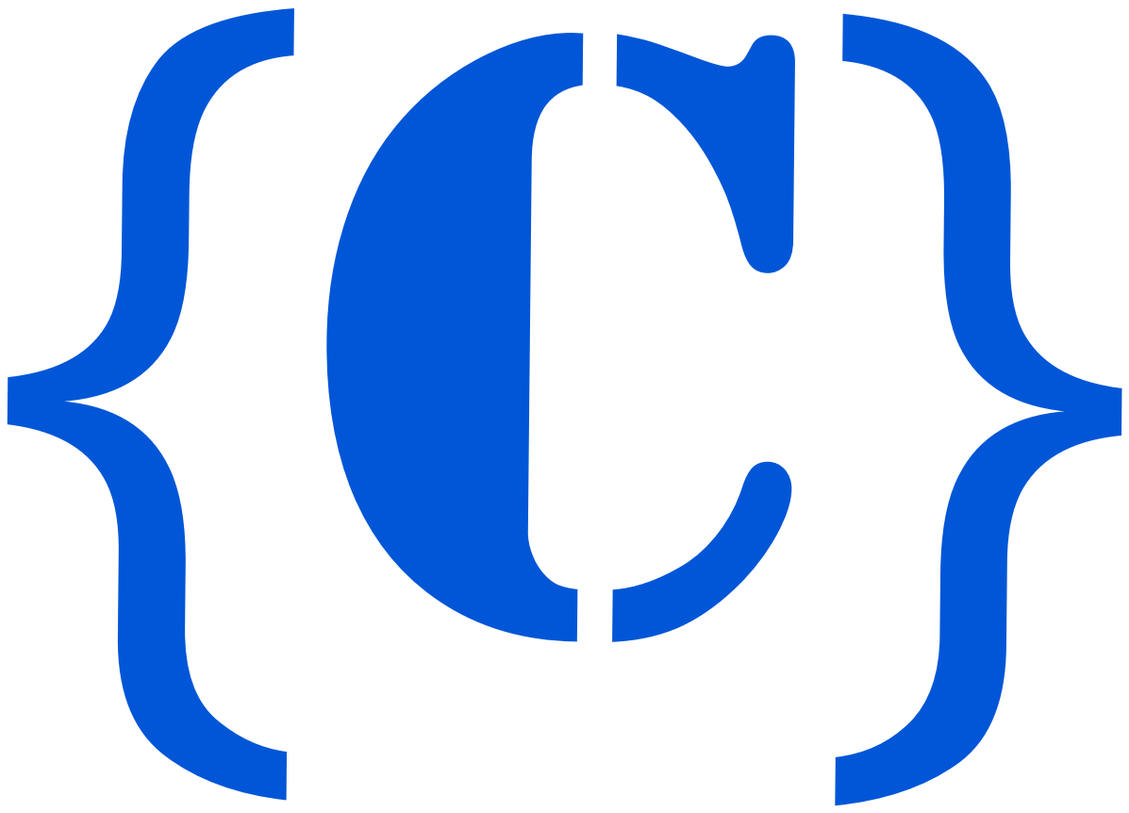
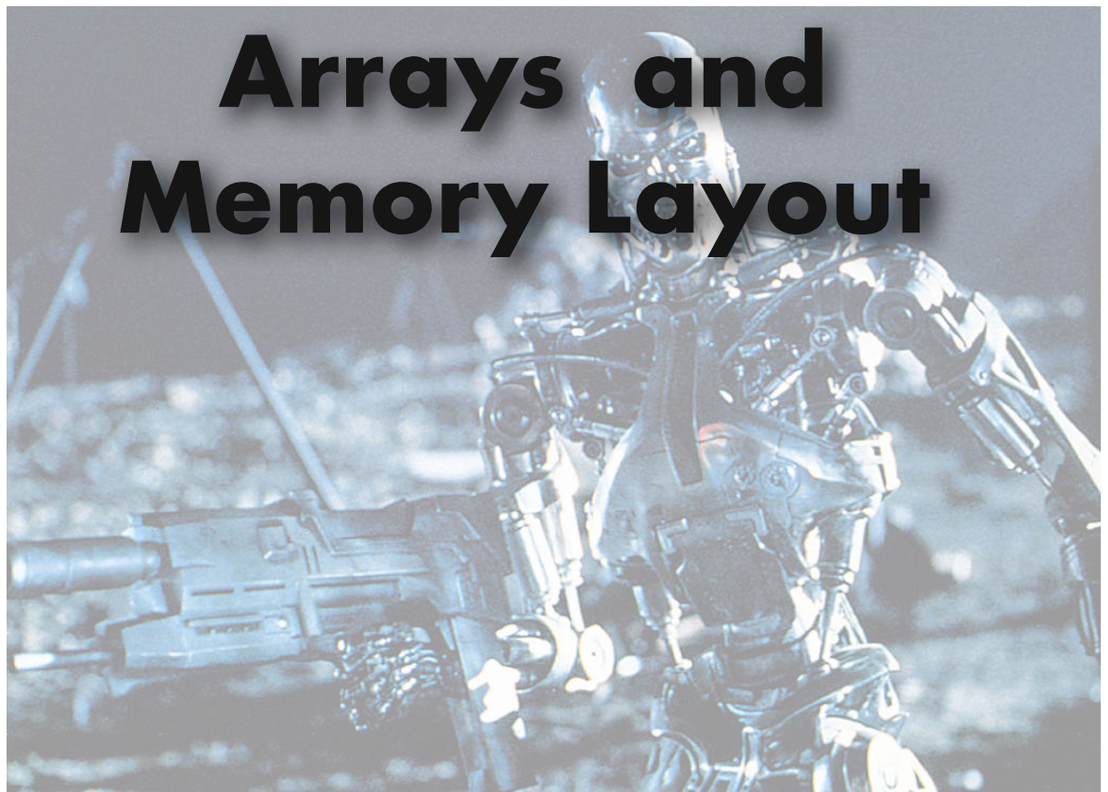


Lecture 7

Arrays and Memory Layout



Arrays


```
char a[2][3];
```

Creates a two dimensional array of characters

What is the value of a?

What is the address of a?

How is the data stored?

What is the relationship between arrays and pointers?

Can they be converted?

Experimenting…


```
char a[2][3];

printf("%p\n",   a      );  0x7fff682ba976
printf("%p\n",  &a      );  0x7fff682ba976
printf("%p\n",  &a[0]   );  0x7fff682ba976
printf("%p\n",  &a[0][0]);  0x7fff682ba976
printf("%p\n",  &a[0][1]);  0x7fff682ba977
printf("%p\n",  &a[0][2]);  0x7fff682ba978
printf("%p\n",  &a[1][0]);  0x7fff682ba979
printf("%p\n",  &a[1][1]);  0x7fff682ba97a
```

Arrays


```
char a[2][3];
```

An array variable's value is the address of the array's first element

A multi-dimensional array is stored in memory as a single array of the base type with all rows occurring consecutively

There is no padding or delimiters between rows

All rows are of the same size

Pointers and arrays

There is a strong relationship between pointers and arrays

```
int a[10];
int* p;
```

A pointer (e.g. `p`) holds an address while the name of an array (e.g. `a`) denotes an address

Thus it is possible to convert arrays to pointers

```
p = a;
```

Array operations have equivalent pointer operations

```
a[5]      ==      *( p + 5 )
```

Note that `a=p` or `a++` are compile-time errors.

Pointers to arrays


```
char a[2][3];
```

Multi-dimensional array that stores two strings of 3 characters. (Not necessarily zero-terminated)

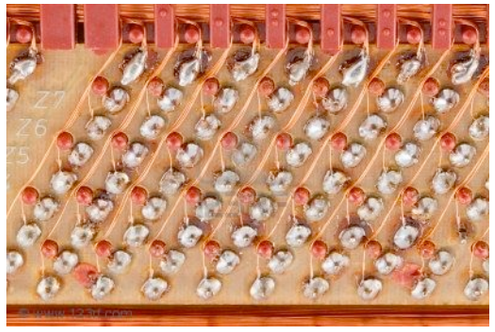
```
char a[2][3]={“ah”,”oh”};
```

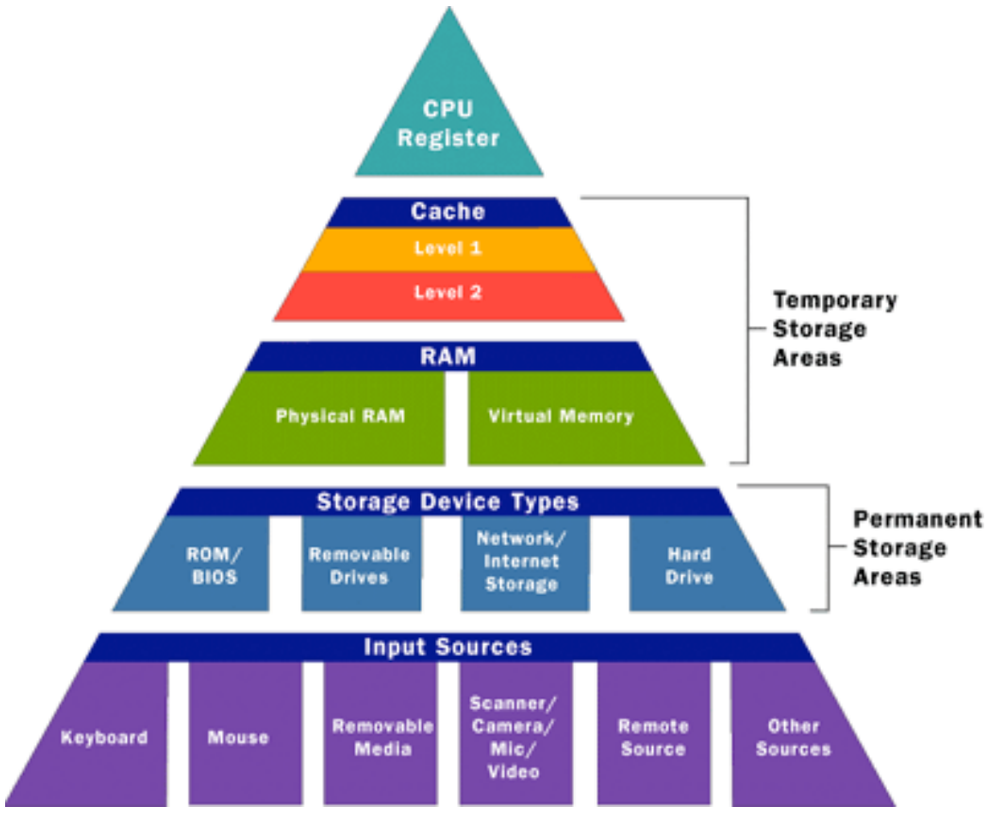
Array initialized with 2 zero-terminated strings.

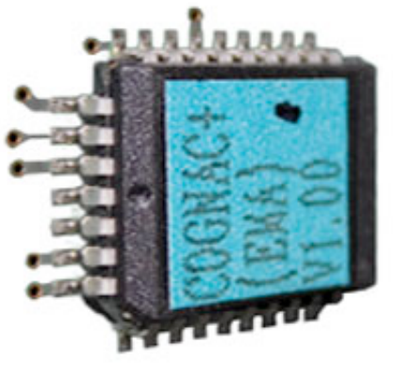
```
char *p = &a[1];

while( *p != ‘\0’ ) p++;
```

Iterate over the second string

Memory: on the hardware side







@ http://computer.howstuffworks.com/computer-memory.htm/printable

Memory: on the software side

Each computer programming languages offers *a different abstraction*

The goal is to make programming easier and improve portability of the source code by hiding irrelevant hardware oddities

Each language offers a memory API – a set of operations for manipulating memory

Consider the differences between C and Java

Memory: the C Story

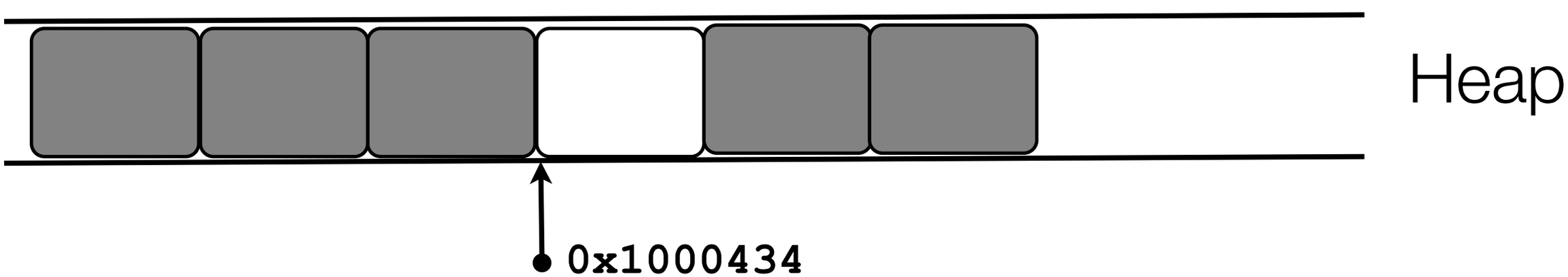
C offers a story both simpler and more complex than Java

Memory is a sequence of bytes, read/written by providing an address

Addresses are values manipulated using arithmetic & logic operations

Memory can be allocated:

- Statically
- Dynamically on the stack
- Dynamically on the heap



Memory layout

The OS creates a process by assigning memory and other resources

C exposes the layout as the programmer can take the address of any element (with &)

Stack:

- *keeps track of where each active subroutine should return control when it finishes executing; stores local variables*

Heap:

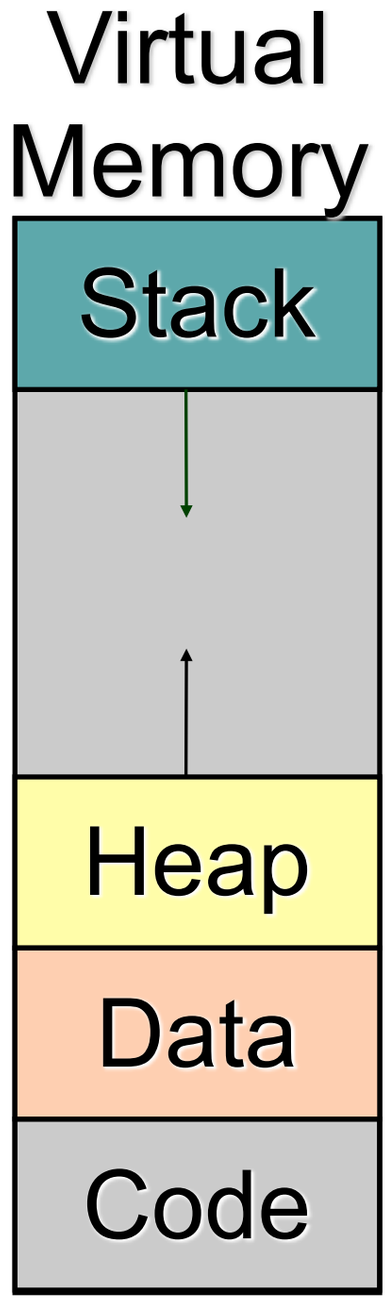
- *dynamic memory for variables that are created with malloc, calloc, realloc and disposed of with free*

Data:

- *global and static variables*

Code:

- *instructions to be executed*



Static and Stack allocation

Static allocation with the keyword `static`

Stack allocation automatic by the compiler for local variables

`printf` can display the address of any identifier

```
#include <stdio.h>

static int sx;
static int sa[100];
static int sy;

int main() {
int lx;
static int sz;

printf("%p\n", &sx);    0x1029e0004
printf("%p\n", &sa);    0x1029e0010
printf("%p\n", &sy);    0x1029e01a0
printf("%p\n", &lx);    0x30964404c
printf("%p\n", &sz);    0x1029e0000
printf("%p\n", &main); 0x1029dbf20
```

Static and Stack allocation

Any value can be turned into a pointer

Arithmetics on pointers allowed

Nothing prevents a program from writing all over memory

```
static int sx;
static int sa[100];
static int sy;

int main() {
  for(p= (int*)0x100001084;
       p <= (int*)0x100001230;
       p++)
 {
    *p = 42;
 }
 printf("%i\n",sx);
 printf("%i\n",sa[0]);
 printf("%i\n",sa[1]);
```

```
42
42
42
```

Sizeof

In C, programmers must know the size of data structures

The compiler provides a way to determine the size of data using the sizeof function; it has no runtime effect

```
struct {
 int i; char c; float cv;
} C;
```

```
int x[10];
printf("%i\n", (int) sizeof(char));      1
printf("%i\n", (int) sizeof(int));       4
printf("%i\n", (int) sizeof(int*));      8
printf("%i\n", (int) sizeof(double));    8
printf("%i\n", (int) sizeof(double*));   8
printf("%i\n", (int) sizeof(x));        80
printf("%i\n", (int) sizeof(C));        12
```