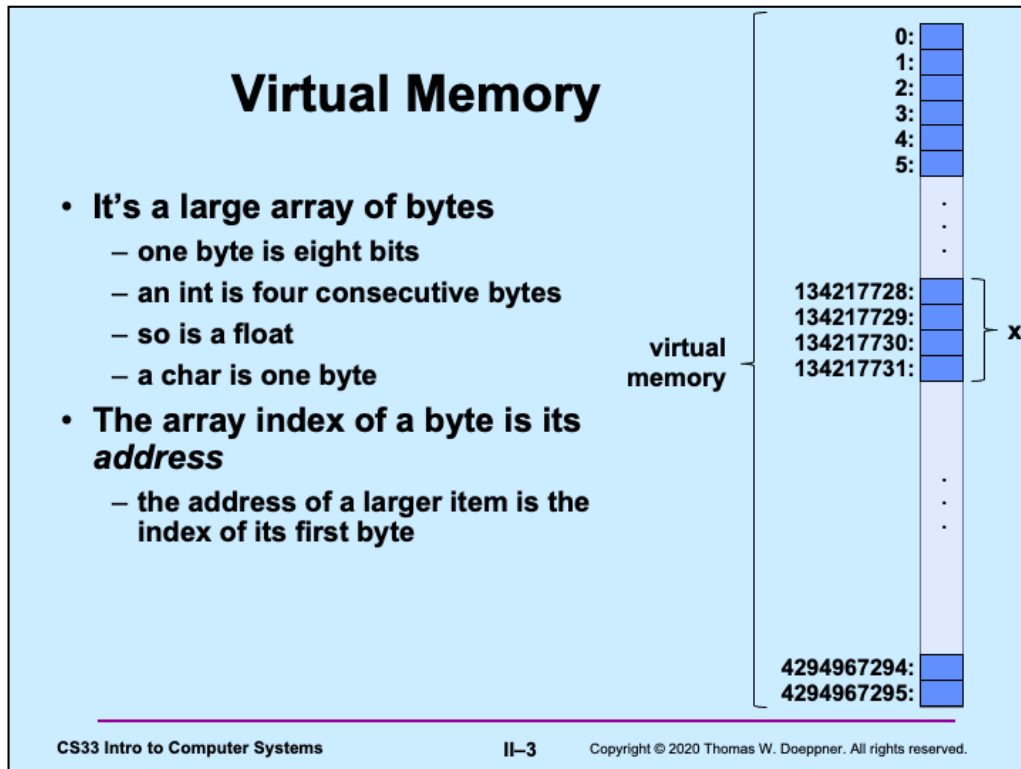# CS 33

Introduction to C
Part 2

Some of this lecture is based on material prepared by Pascal Van Hentenryck.

Memory

- **"Real" memory**
  - it's complicated
  - it involves electronics, semiconductors, physics, etc.
  - it's not terribly relevant at this point
- **"Virtual" memory**
  - the notion of memory as used by programs
  - it involves logical concepts
  - it's how you should think about memory (most of the time)

Virtual Memory

- **It's a large array of bytes**
  - one byte is eight bits
  - an int is four consecutive bytes
  - so is a float
  - a char is one byte
- **The array index of a byte is its *address***
  - the address of a larger item is the index of its first byte

virtual memory

```
0:
1:
2:
3:
4:
5:
.
.
.
134217728:
134217729:          } x
134217730:
134217731:
.
.
.
4294967294:
4294967295:
```
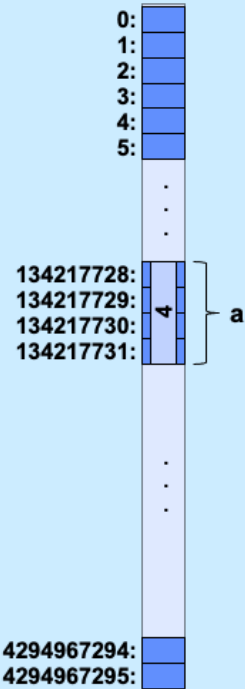
In the diagram, x is an int occupying bytes 134217728, 134217729, 134217730, and 134217731. Its address is 134217728; its size is 4 (bytes).

Memory addresses in C

- **In C**
  - **you can get the memory address of any variable**
  - **just use the operator &**

```
int main() {
    int a = 4;
    printf("%u\n", &a);
}
```

```
$ ./a.out
134217728
```

0:
1:
2:
3:
4:
5:

.
.
.

134217728:
134217729:
134217730:
134217731:

4 } a

.
.
.

4294967294:
4294967295:

The "%u" format code in printf means to interpret the item being printed as being unsigned. We'll explain this concept more thoroughly in an upcoming lecture. What's being printed is an address, which can't be negative.

C Pointers

- **What is a C pointer?**
    - a variable that holds an address
- **Pointers in C are "typed" (remember the promises)**
    - pointer to an int
    - pointer to a char
    - pointer to a float
    - pointer to <whatever you can define>
- **C has a syntax to declare pointer types**
    - things start to get complicated …

C Pointers

p is a pointer to an int

if you follow p, you find an int

```
int main() {
    int *p;
    int a = 4;
    p = &a;
    printf("%u\n", p);
}
```
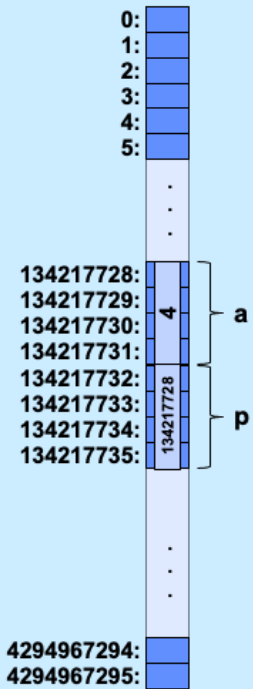
p is assigned the address of a

```
$ ./a.out
134217728
```

This slide assumes that pointers are 4 bytes long. As we'll discuss later, on sunlab machines (and most other current computers), pointers are 8 bytes long.

Some compilers might choose to order p in memory before a.

C Pointers

- **Pointers are typed**
  - the types of the objects they point to are known
  - there is one exception (discussed later)
- **Pointers are first-class citizens**
  - they can be passed to functions
  - they can be stored in arrays and other data structures
  - they can be returned by functions

Swapping

What does this do?

```
void swap(int *i, int *j) {
    int *tmp;
    tmp = j; j = i; i = tmp;
}
int main() {
    int a = 4;
    int b = 8;
    swap(&a, &b);
    printf("a:%d  b:%d\n", a, b);
}
```
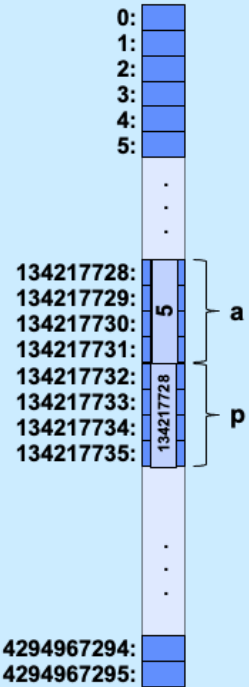
Damn!

```
$ ./a.out
a:4  b:8
```

C Pointers

- **Dereferencing pointers**
  - accessing/modifying the value pointed to by a pointer

```
int main() {
    int *p;
    int a = 4;
    p = &a;
    printf("%d\n", *p);
    *p = *p + 1;
    printf("%d\n", *p);
}
```

```
$ ./a.out
4
5
```

0:	
1:	
2:	
3:	
4:	
5:	
. . .	
134217728:	
134217729:	5 → a
134217730:	
134217731:	
134217732:	
134217733:	134217728 → p
134217734:	
134217735:	
. . .	
4294967294:	
4294967295:	

Dereferencing C Pointers

```c
int main() {
    int *p;
    int a = 4;
    p = &a;
    printf("%d\n", *p);
    *p = *p + 1;
    *p += 3;
    printf("%d\n", a);
}
```

```
$ ./a.out
4
8
```

Note that "*p" and "a" refer to the same thing after p is assigned the address of a.

"x+=y" means the same as "x = x+y". Similarly, there are −=, *=, and /= operators.

Swapping

```
void swap(int *i, int *j) {
    int tmp;
    tmp = *j; *j = *i; *i = tmp;
}
int main() {
    int a = 4;
    int b = 8;
    swap(&a, &b);
    printf("a:%d  b:%d\n", a, b);
}
```
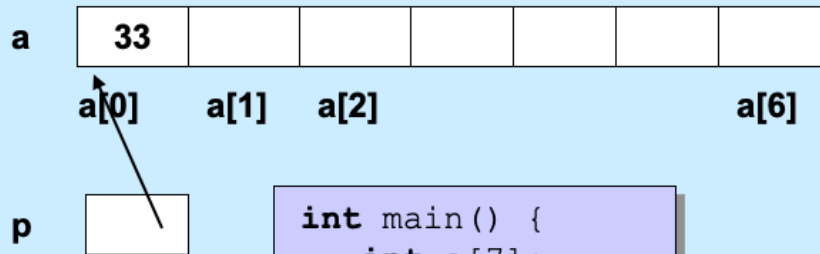
Hooray!

```
$ ./a.out
a:8  b:4
```

Quiz 1

```c
int doubleit(int *p) {
    *p = 2*(*p);
    return *p;
}
int main() {
    int a = 3;
    int b;
    b = doubleit(&a);
    printf("%d\n", a*b);
}
```

What's printed?

a) 0
b) 12
c) 18
d) 36

Pointers and Arrays

a | **33** | | | | | | |

a[0]   a[1]   a[2]   a[6]

p
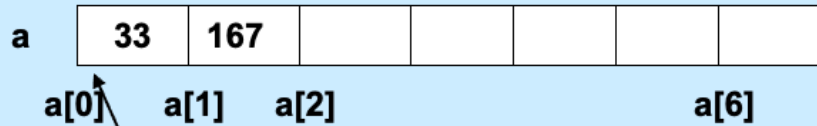
```
int main() {
    int a[7];
    int *p;
    p = &a[0];
    *p = 33;
}
```

The pointer p points to the first element of the array a. Thus a[0] and *p have identical values.

Pointer Arithmetic

Pointers can be incremented/decremented

– what this does depends on its type

a | 33 | 167 | | | | | |

a[0]   a[1]   a[2]   a[6]

p | |

```
int main() {
    int a[7];
    int *p;
    p = &a[0];
    *p = 33;
    *(p+1) = 167;
}
```

Adding one to a pointer, rather than increasing its value by one, causes it to refer to the next element. Thus if the size of what it refers to is 4 (which is the case for pointers to ints), adding one to the pointer increases its value by 4 (thus making it point to the next 4-byte value).

Pointer Arithmetic

Pointers can be incremented/decremented
– what this does depends on its type

a

a[0]   a[1]   a[2]   a[6]

p

```
int main() {
    int a[7];
    int *p;
    p = &a[0];
}
```
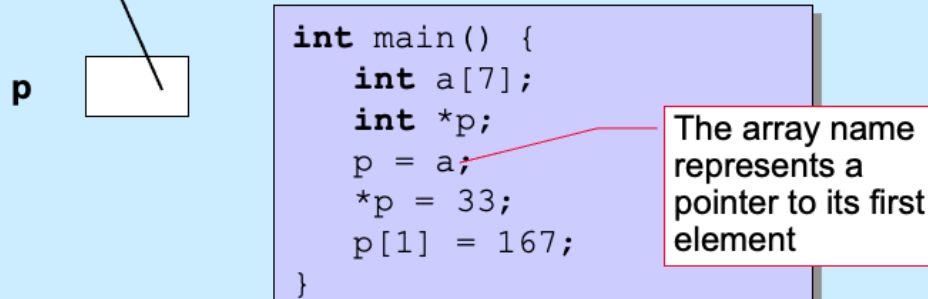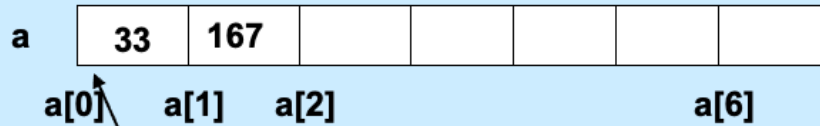
Now p and a
have the
same value
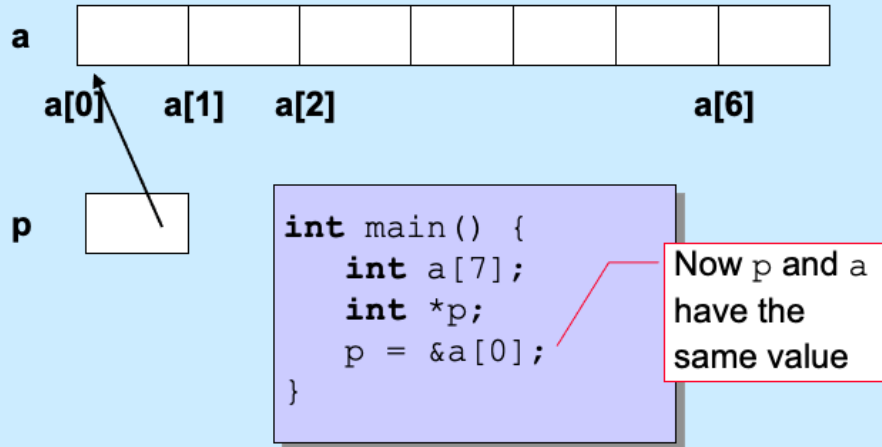
Note that setting p equal to the address of the first element of the array a is equivalent to setting p to the value of a.

Pointer Arithmetic

Pointers can be incremented/decremented
  – what this does depends on its type

a

33	167					

a[0]   a[1]   a[2]   a[6]

p

```
int main() {
    int a[7];
    int *p;
    p = a;
    *p = 33;
    p[1] = 167;
}
```

The array name represents a pointer to its first element

A pointer to the first element of an array can be used as if it were the array itself. Thus, in this example, there's little difference between how one uses "p" and "a".

An array's name represents a pointer to its first element.

Pointers and Arrays

`p = &a[0];` can also be written as `p = a;`

`a[i];` really is `*(a+i)`

- **This makes sense, yet is weird ...**
  - `p` **is of type** `int *`
    - **it can be assigned to**
      `int *q;`
      `p = q;`
  - `a` **sort of behaves like an** `int *`
    - **but it can't be assigned to**
      ~~`a = q;`~~

Pointers and Arrays

- **An array name represents a pointer to the first element of the array**
- **Just like a literal represents its associated value**
    - **in:**

        ```
        x = y + 2;
        ```
        - » **"2" is a *literal* that represents the value 2**
    - **can't do**

        ```
        2 = x + y;
        ```

Literals and Functions

```
int func(int x) {
    x = x + 4;
    return x * 2;
}

int main() {
    result = func(2);
    printf("%d\n", result);
    return 0;
}
```

initialized with a copy of the argument

As we've already discussed, arguments to functions are passed by value – this means that the function receives a copy of the argument.

Arrays and Functions

```
int func(int *a, int nelements) {
    // sizeof(a) == sizeof(int *)
    int i;
    for (i=0; i<nelements-1; i++)
        a[i+1] += a[i];
    return a[nelements-1];
}

int main() {
    int array[50] = ... ;
    // sizeof(array) == 50*sizeof(int)
    printf("result = %d\n", func(array, 50));
    return 0;
}
```

initialized with a copy of the argument

Note that the argument to *func* is not the entire array, but the pointer to its first element. Thus *a* is initialized by copying into it this pointer.

Note that when "array" is used as an identifier in the program, it refers to address of the beginning of the array. However, "sizeof" is special. It's not a function in the normal sense – it's not something that's called when the program is running. Instead, it's replaced with information known to the compiler when the program is compiled. While there is nothing stored with "array" that indicates its size is 50*sizeof(int), the compiler is aware of how much memory it allocated for array and, thus, at compile time, it replaces "sizeof(array)" with 200 (assuming sizeof(int) is 4).

Equivalently ...

```
int func(int a[], int nelements) {
    // sizeof(a) == sizeof(int *)
    ...

}
```

No need for array size, since all that's used is pointer to first element

```
int main() {
    int array[50] = ... ;
    // sizeof(array) == 50*sizeof(int)
    printf("result = %d\n", func(array, 50));
    return 0;
}
```

Note that one could include the size of the array ("**int** proc(**int** a[50], **int** nelements)"), but the size would be ignored, since it's not relevant: arrays don't know how big they are. Thus the *nelements* argument is very important.

Parameter passing

Passing arrays to a function

```
int average(int a[], int size) {
    int i; int sum;
    for(i=0,sum=0; i<size; i++)
        sum += a[i];
    return sum/s;
}
int main() {
    int a[100];
    …
    printf("%d\n",average(a,100));
}
```

- **Note that I need to pass the size of the array**
- **This array has no idea how big it is**

Swapping

Write a function to swap two entries of an array

```
void swap(int a[], int i, int j) {
    int tmp;
    tmp = a[j];
    a[j] = a[i];
    a[i] = tmp;
}
```

Selection Sort

```
void selectsort(int array[], int length){
  int i, j, min;
  for (i = 0; i < length; ++i){
    /* find the index of the smallest item from i onward */
    min = i;
    for (j = i; j < length; ++j)
      if (array[j] < array[min])
        min = j;
    /* swap the smallest item with the i-th item */
    swap(array, i, min);
  }
  /* at the end of each iteration, the first i slots have the i
     smallest items */
}
```

Note that C uses the same syntax as Java does for conditional (if) statements. In addition to relational operators such as "==", "!=", "<", ">", "<=", and ">=", there are the conditional operators "&&" and "||" ("logical and" and "logical or", respectively).

Quiz 2

```
int func(int a[], int nelements) {
    int b[5] = {0, 1, 2, 3, 4};
    a = b;
    return a[1];
}

int main() {
    int array[50];
    array[1] = 0;
    printf("result = %d\n",
        func(array, 50));
    return 0;
}
```

This program prints:

a) 0
b) 1
c) 2
d) nothing: it doesn't
   compile because of a
   syntax error

Note how we initialize the contents of array *b* in *func*.

Quiz 3

```c
int func(int a[], int nelements) {
    int b[5] = {0, 1, 2, 3, 4};
    a = b;
    return a[1];
}

int main() {
    int array[5] = {4, 3, 2, 1, 0};
    func(array, 5);
    printf("%d\n", array[1]);
    return 0;
}
```

This program prints:

a) 0
b) 1
c) 2
d) 3

The Preprocessor

`#include`

- **calls the preprocessor to include a file**

What do you include?

- **your own *header* file:**
  `#include "fact.h"`
  - **look in the current directory**
- **standard *header* file:**
  `#include <assert.h>`
  `#include <stdio.h>` — Contains declaration of *printf* (and other things)
  - **look in a standard place**

The preprocessor modifies the source code before the code is compiled. Thus its output is what is passed to gcc's compiler.

Note that one must include *stdio.h* if using *printf* (and some other functions) in a program.

On most Unix systems (including Linux, but not OS X), the standard place for header files is the directory /usr/include.

Function Declarations

fact.h

main.c

```
float fact(int i);
```

```
#include "fact.h"
int main() {
    printf("%f\n", fact(5));
    return 0;
}
```

It's convenient to package the declaration of functions (and other useful stuff) in header files, such as *fact.h,* so the programmer need simply to include them, rather than reproduce their contents.

The source code for the *fact* function would be in some other file, perhaps as part of a library (a concept we discuss later).

#define

```
#define SIZE 100
int main() {
    int i;
    int a[SIZE];
}
```

#define
- defines a substitution
- applied to the program by the preprocessor

#define

```
#define forever for(;;)
int main() {
    int i;
    forever {
        printf("hello world\n");
    }
}
```

The #define directive can be used for pretty much anything, such as segments of code as shown in the slide. (It's not its concern as to whether the code segments are useful!)

assert

```
#include <assert.h>
float fact(int i) {
   int k; float res;
   assert(i >= 0);
   for(res=1,k=1; k<=i; k++)
      res = res * k;
   return res;
}
int main() {
   printf("%f\n", fact(-1));
```

assert
- verify that the assertion holds
- abort if not

```
$ ./fact
main.c:4: failed assertion 'i >= 0'
Abort
```

The assert statement is actually implemented as a macro (using #define). One can "turn off" asserts by defining (using #define) NDEBUG. For example,

#include <assert.h>

...

#define NDEBUG

...

assert(i>=0);

In this case, the assert will not be executed, since NDEBUG is defined. Note that one also can define items such as NDEBUG on the command line for gcc using the –D flag. For example,

gcc –o prog prog.c –DNDEBUG

Has the same effect as having "#define NDEBUG" as the first line of prog.c.