C Survival Guide

How do I write good C programs?

- Fluency in C syntax
- Stack (static) vs. Heap (dynamic) memory allocation
- Key skill: design
  - Think how to structure (functions, global/local variables, etc.) your program before you start writing your first line of code
- Key skill: debugging
  - Learn to use a debugger. Don't only rely on `printfs`!
- Key skill: defensive programming
  - Avoid assumptions about what is probably true

Why C instead of Java?

- C helps you get “under the hood”
  - One step up from assembly language
  - Many existing servers/systems written in C
- C helps you learn how to write large-scale programs
  - C is lower-level
    - C provides more opportunities to create abstractions
  - C has some flaws
    - C’s flaws motivate discussions of software engineering principles

C design Goals

- C design goals
    - Support structured programming
    - Support development of the Unix OS and Unix tools
        - As Unix became popular, so did C
- Implications for C
    - Good for systems-level programming
    - Low-level
    - Efficiency over portability
    - Efficiency over security
- Anything you can do in Java you can do in C – it just might look ugly in C!

C vs. C++

- C++ is “C with Classes”
- C is only a subset of C++
  - C++ has objects, a bigger standard library (e.g., STL), parameterized types, etc.
  - C++ is a little bit more strongly typed
- C is fortunately a subset of C++
  - Can be simpler, more direct
- C is a subset of C++
  - All syntax you use in this class is valid for C++
  - Not all C++ syntax you’ve used, however, is valid for C

Compiler

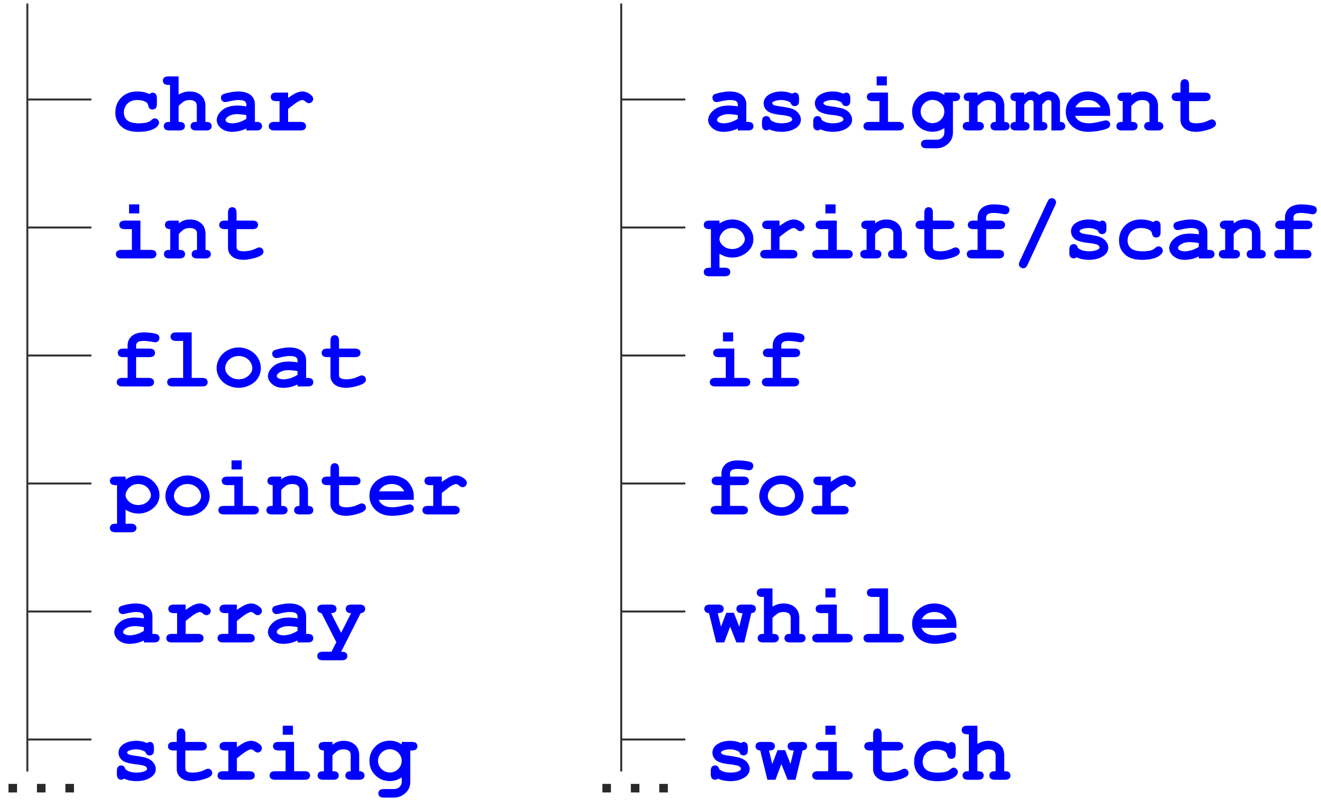
- gcc
  - Preprocessor
  - Compiler
  - Linker
  - See manual “man” for options: man gcc
- "Ansi-C" standards C89 versus C99
  - C99: Mix variable declarations and code (for int i=…)
  - C++ inline comments //a comment
- make – a utility to build executables

Programming in C

- C = Variables + Instructions

Programming in C

- C = Variables + Instructions



- char
- int
- float
- pointer
- array
- string

…

- assignment
- printf/scanf
- if
- for
- while
- switch

…

What we’ll show you

- You already know a lot of C from C++:

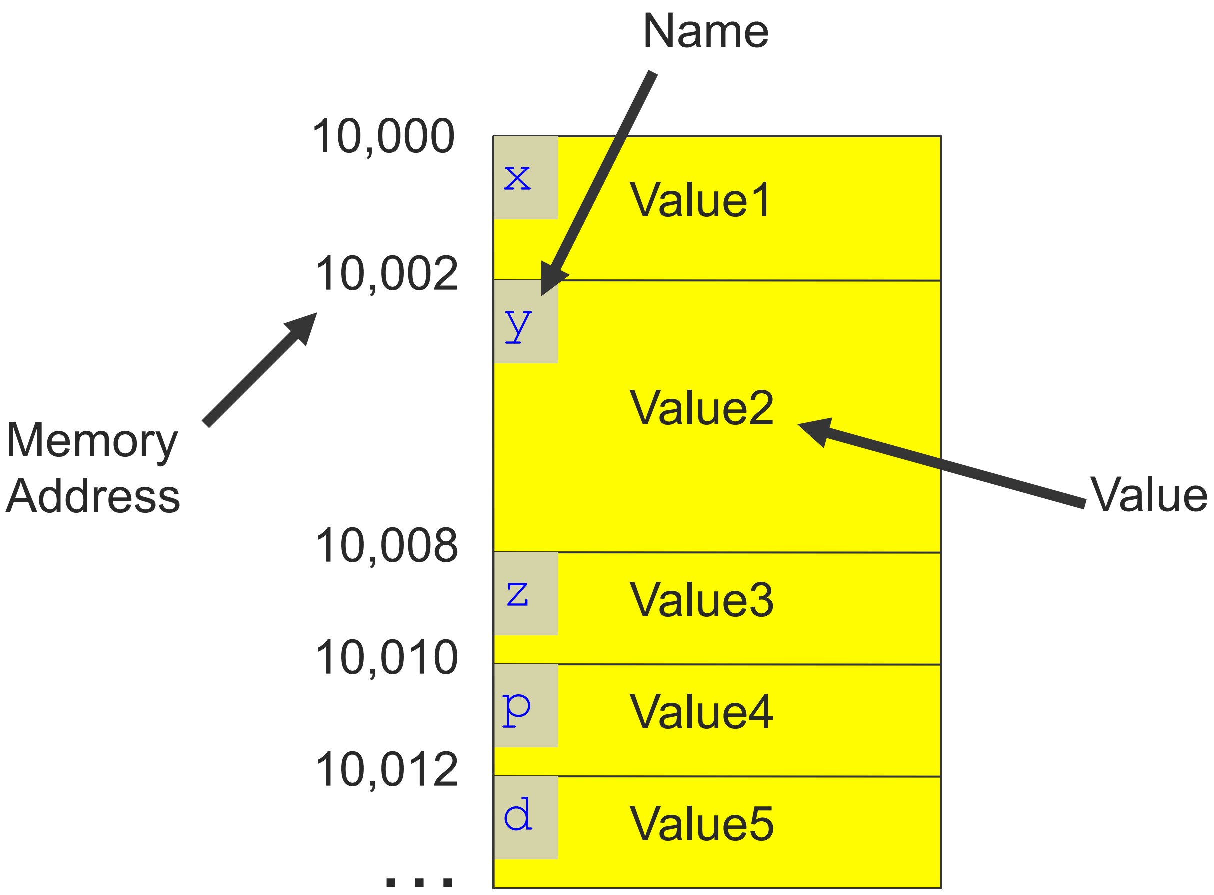
```
int my_fav_function(int x) {
    return x+1; }
```

- Key concepts for this lecture:
  - Pointers
  - Memory allocation
  - Arrays
  - Strings

Theme:
how memory
really works

Pointers

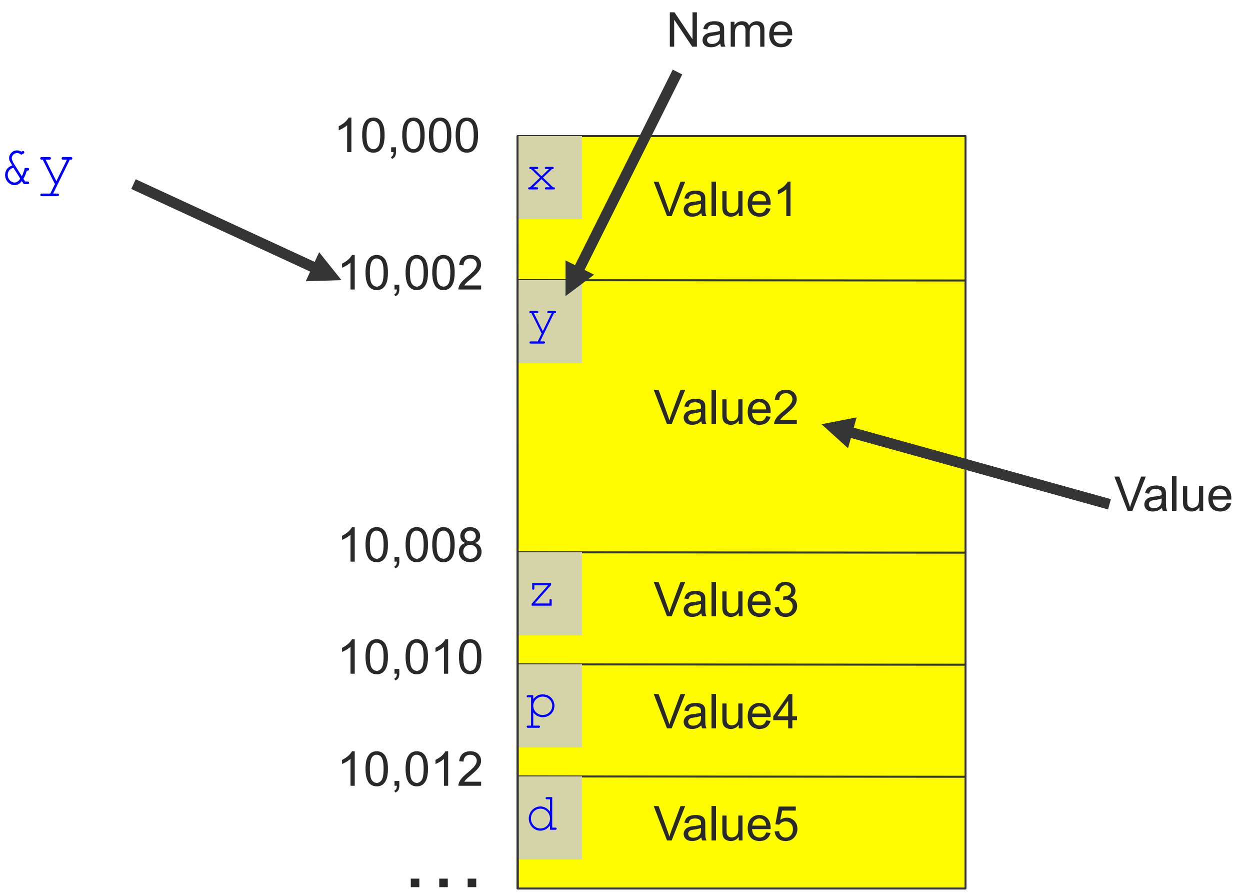
Variables



Type of each variable
(also determines size)

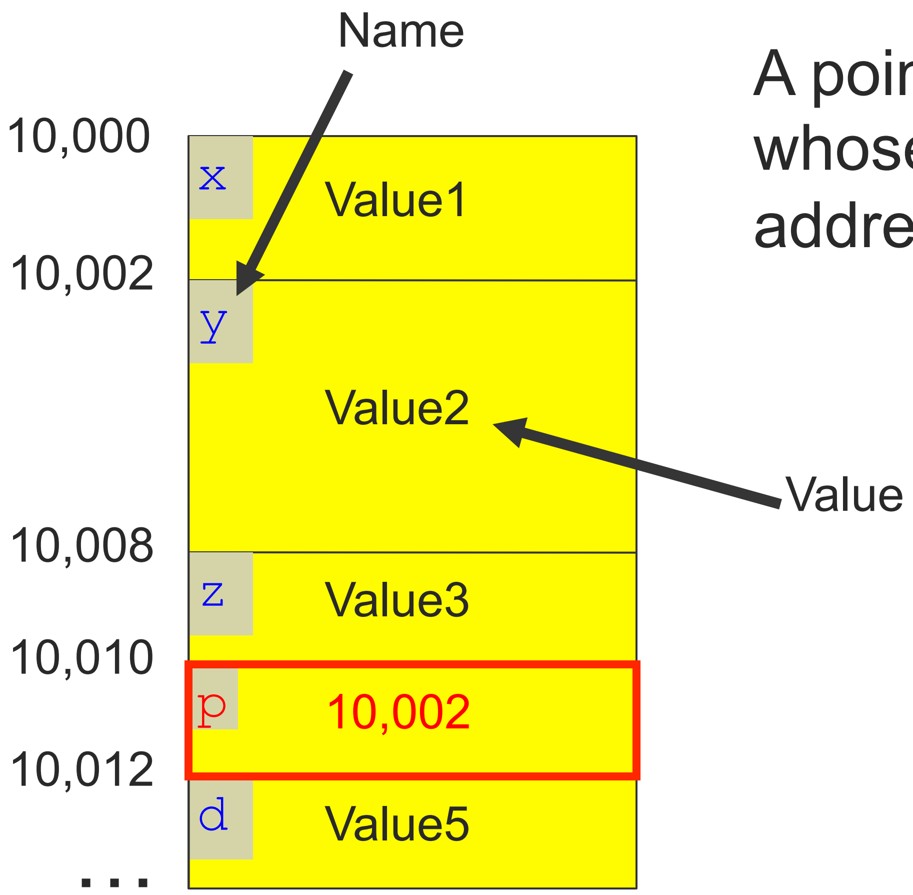
```
int       x;
double    y;
float     z;
double*   p;
int       d;
```

The "&" Operator: Reads "Address of"



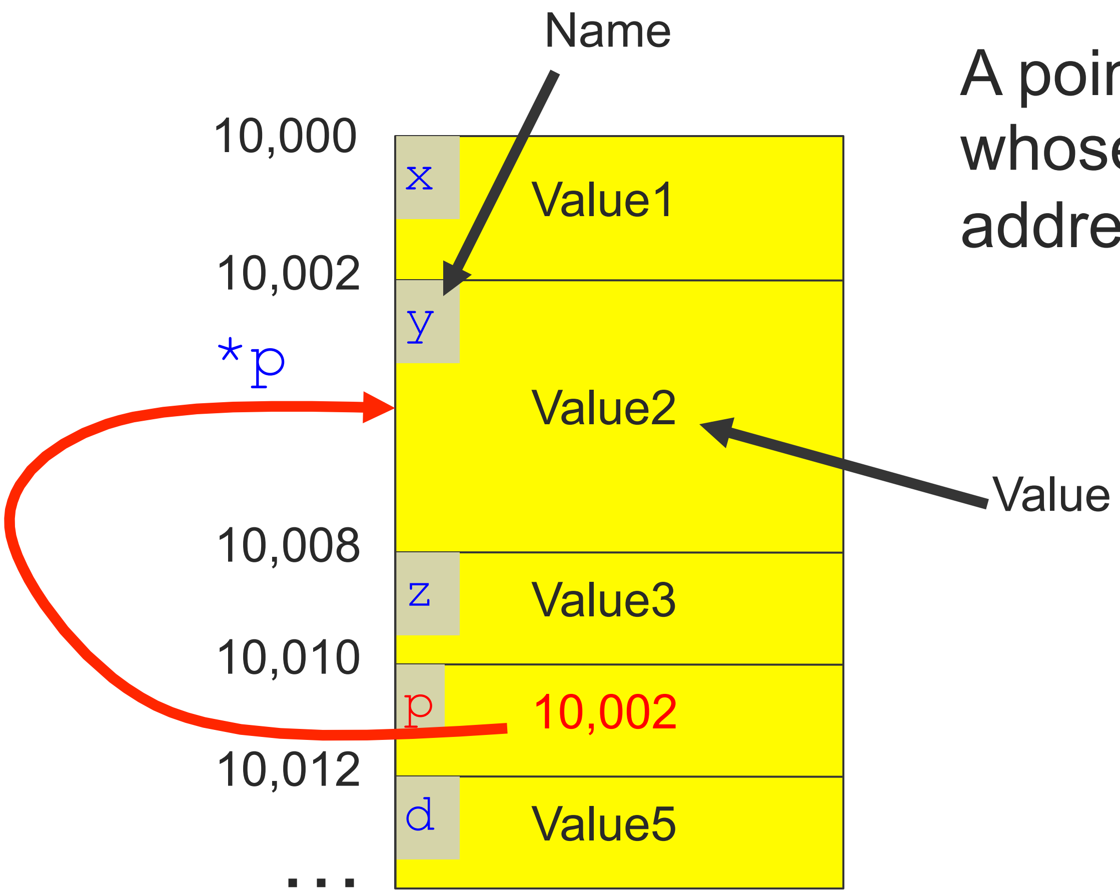
Copyright ©: University of Illinois CS 241 Staff

Pointers



A pointer is a variable whose value is the address of another

The "*" Operator Reads "Variable pointed to by"



A pointer is a variable whose value is the address of another

Cardinal Rule: Must Initialize Pointers before Using them

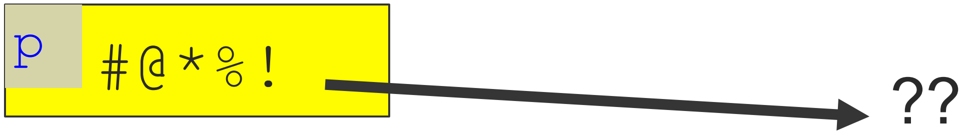
```
int *p;
*p = 10;
```

GOOD or BAD?

Cardinal Rule: Must Initialize Pointers before Using them

```
int *p;
*p = 10;
```

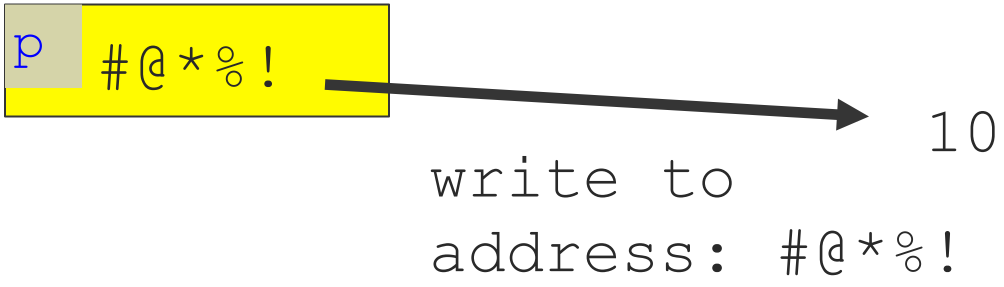
BAD!



Pointing somewhere random

Cardinal Rule: Must Initialize Pointers before Using them

```
int *p;
*p = 10;
```



p

#@*%!

10

write to
address: #@*%!

How to initialize pointers

- Set equal to address of some piece of memory
- …or NULL for "pointing nowhere"

- OK, where do we get memory?

Memory allocation

Memory allocation

- Two ways to dynamically allocate memory
- Stack
  - Named variables in functions
    - Allocated for you when you call a function
    - Deallocated for you when function returns
- Heap
  - Memory on demand
    - You are responsible for all allocation and deallocation

Allocating and deallocating heap memory

- Dynamically allocating memory
  - Programmer explicitly requests space in memory
  - Space is allocated dynamically on the heap
  - using “malloc” in C
- Dynamically deallocating memory
  - Must reclaim or recycle memory that is never used again
  - To avoid (eventually) running out of memory
  - using “free” in C

Manual Deallocation

- Programmer deallocates memory (C and C++)
  - Manually determines which objects can't be accessed
  - And then explicitly returns those resources to the heap
  - e.g., using "free" in C or "delete" in C++
- Advantages
  - Lower overhead
  - No unexpected "pauses"
  - More efficient use of memory
- Disadvantages
  - More complex for the programmer
  - Subtle memory-related bugs

Manual deallocation can lead to bugs

- Dangling pointers
  - Programmer frees memory … but still has a pointer to it
  - Dereferencing pointer reads or writes nonsense values

```
int main(void) {
     char *p;
     p = malloc(10);
     …
     free(p);
     …
     printf(“%c\n”,*p);
}
```

Manual deallocation can lead to bugs

- Dangling pointers
  - Programmer frees memory … but still has a pointer to it
  - Dereferencing pointer reads or writes nonsense values

```
int main(void) {
      char *p;
      p = malloc(10);
      …
      free(p);
      …
      printf(“%c\n”,*p);
}
```

May print nonsense character

Manual deallocation can lead to bugs

- Memory leak
  - Programmer neglects to free unused region of memory
  - So, the space can never be allocated again
  - Eventually may consume all of the available memory

```
void f(void) {
     char *s;
     s = malloc(50);
}
int main(void) {
     while (1) f();
}
```

Manual deallocation can lead to bugs

- Memory leak
  - Programmer neglects to free unused region of memory
  - So, the space can never be allocated again
  - Eventually may consume all of the available memory

```
void f(void) {
      char *s;
      s = malloc(50);
}
int main(void) {
      while (1) f();
}
```

Eventually, malloc() returns NULL

Manual deallocation can lead to bugs

- Double free
  - Programmer mistakenly frees a region more than once
  - Corruption of the heap or destruction of a different object

```
int main(void) {
     char *p, *q;
     p = malloc(10);
     …
     free(p)
     q = malloc(10);
     free(p)
}
```

Heap memory allocation

- C++:
  - `new` and `delete` allocate memory for a whole object
- C:
  - `malloc` and `free` deal with **unstructured blocks of bytes**

```
void* malloc(size_t size);
void free(void* ptr);
```

Example

```
int* p;

p = (int*) malloc(sizeof(int));

*p = 5;

free(p);
```

How many bytes do you want?

Cast to the right type

I’m hungry. More bytes plz.

```
int* p = (int*) malloc(10 * sizeof(int));
```

- Now I have space for 10 integers, laid out contiguously in memory. What would be a good name for that...?

Arrays

- Contiguous block of memory
  - Fits one or more elements of some type
- Two ways to allocate
  - named variable

```
int x[10];
```

  - dynamic

```
int* x = (int*)
   malloc(10*sizeof(int));
```

Is there a difference?

Arrays

- Contiguous block of memory
  - Fits one or more elements of some type
- Two ways to allocate
  - named variable

```
int x[10];
```

  - dynamic

```
int* x = (int*)
   malloc(10*sizeof(int));
```

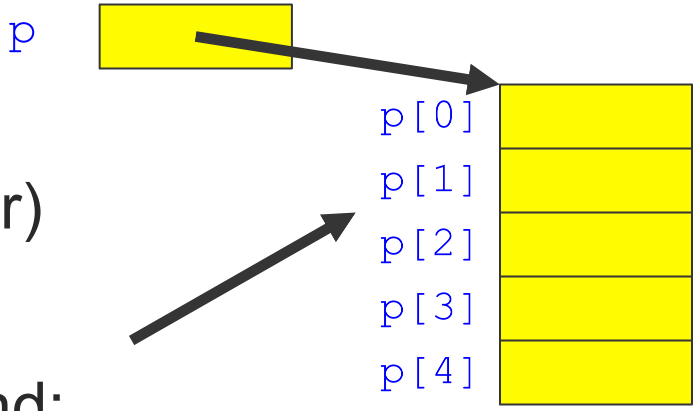
Is there a difference?

One is on the stack, one is on the heap

Arrays

```
int p[5];
```

Name of array (is a pointer)



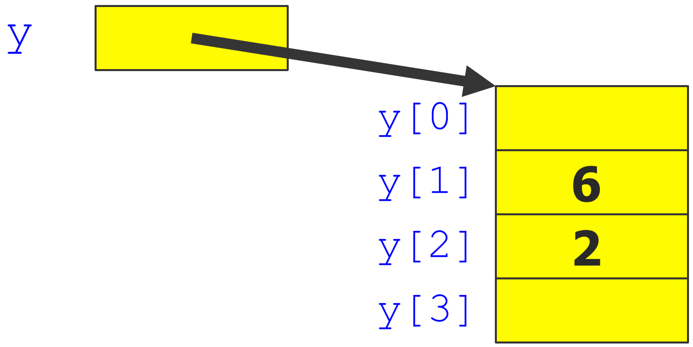
Shorthand:

```
*(p+1) is called p[1]
*(p+2) is called p[2]
```

etc..

Example

```
int y[4];
y[1]=6;
y[2]=2;
```



Array Name as Pointer

- What’s the difference between the examples?

- Example 1:

```
int z[8];
int *q;
q=z;
```

- Example 2:

```
int z[8];
int *q;
q=&z[0];
```

Array Name as Pointer

- What’s the difference between the examples?

- Example 1:

```
int z[8];
int *q;
q=z;
```

- Example 2:

```
int z[8];
int *q;
q=&z[0];
```

NOTHING!!

`z` (the array name) is a pointer to the beginning of the array, which is `&z[0]`

Questions

- What’s the difference between

```
int* q;
int q[5];
```

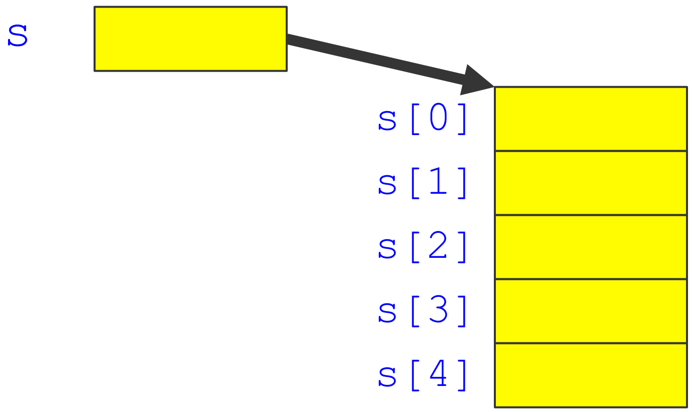
- What’s wrong with

```
int ptr[2];
ptr[1] = 1;
ptr[2] = 2;
```

Strings

Strings (Null-terminated Arrays of Char)

- Strings are arrays that contain the string characters followed by a "Null" character `'\0'` to indicate end of string.
  - Do not forget to leave room for the null character
- Example
  - `char s[5];`



Conventions

- Strings
  - “`string`”
  - “`c`”
- Characters
  - ‘`c`’
  - ‘`X`’

String Operations

- strcpy
- strlen
- strcat
- strcmp

strcpy, strlen

- What’s wrong with

```
char str[5];
strcpy (str, "Hello");
```

Constants: binary/decimal/hexadecimal

- What is the difference between these assignments?
  - i =   42;
  - i =   0x2a;
  - i = 0b101010;

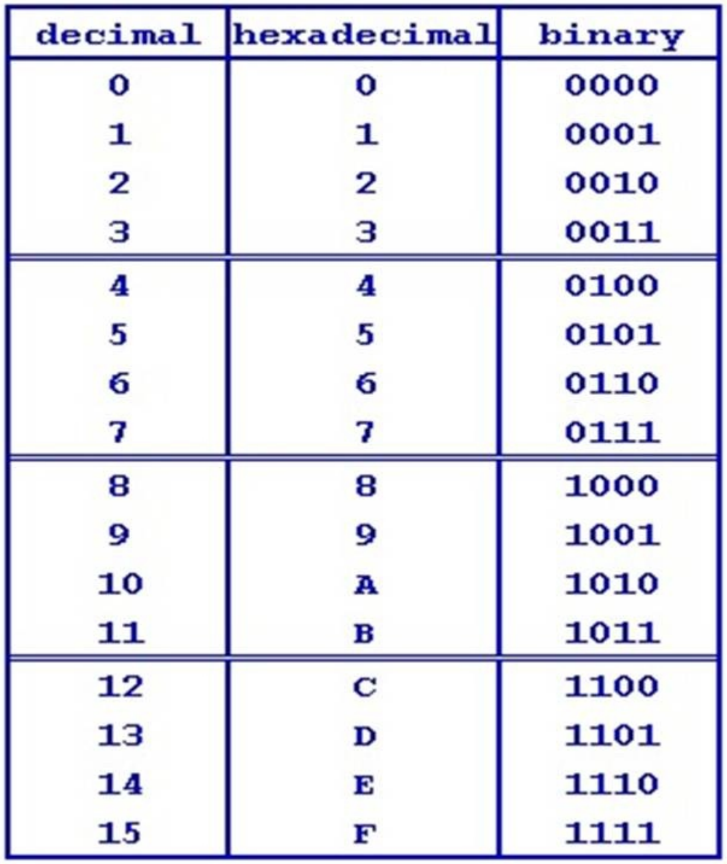
Constants: binary/decimal/hexadecimal

- What is the difference between these assignments?
  - `i =       42;`
  - `i =     0x2a;`
  - `i = 0b101010;`

  - These assignments are identical!

Constants: binary/decimal/hexadecimal

- You should be able to convert between binary and hexadecimal quickly.



decimal	hexadecimal	binary
0	0	0000
1	1	0001
2	2	0010
3	3	0011
4	4	0100
5	5	0101
6	6	0110
7	7	0111
8	8	1000
9	9	1001
10	A	1010
11	B	1011
12	C	1100
13	D	1101
14	E	1110
15	F	1111