C Survival Guide

Why C?

- It's 2012.
- Why are we programming in C?
  - Why not C++? Java? C#? Perl? PHP? Python? Ruby?

Why C?

- **1)** C gives you better control of what's going on.

- C++ Code:

```
MyObject obj;
MyObject *ptr = new MyObject;
```


Why C?

- **1)** C gives you better control of what's going on.

- C++ Code:

```
MyObject obj;
MyObject *ptr = new MyObject;
```

Both implicitly invokes **MyObject::MyObject()**

Why C?

- **2)** C requires much less runtime support.
  - C++ contains a large number of libraries, simpler language constructs
  - Makes it more suitable for low-level environments
    - Kernels, embedded devices, OS drivers

Why C?

- The Android, iOS, and Linux kernels are all programmed in C.
  - Want to jailbreak your iPhone…?

Exploit Code

```
int ret, fd; struct vn_ioctl vn; struct hfs_mount_args args;

fd = open("/dev/vn0", O_RDONLY, 0);
if (fd < 0) {
    puts("Can't open /dev/vn0 special file.");
    exit(1);
}

memset(&vn, 0, sizeof(vn));
ioctl(fd, VNIOCDETACH, &vn);
vn.vn_file = "/usr/lib/exploit.hfs";
vn.vn_control = vncontrol_readwrite_io_e;
ret = ioctl(fd, VNIOCATTACH, &vn);
close(fd);
if (ret < 0) {
    puts("Can't attach vn0.");
    exit(1);
}

memset(&args, 0, sizeof(args));
args.fspec = "/dev/vn0";
args.hfs_uid = args.hfs_gid = 99;
args.hfs_mask = 0x1c5;
ret = mount("hfs", "/mnt/", MNT_RDONLY, &args);
```

Source: Stefan Esser, iOS Kernel Exploitation, August 2011, Black Hat Security Conference

C vs. Java: Design Goals

- Java design goals
  - Support object-oriented programming
  - Allow same program to run on multiple operating systems
  - Support using computer networks
  - Execute code from remote sources securely
  - Adopt the good parts of other languages
- Implications for Java
  - Good for application-level programming
  - High-level (insulates from assembly language, hardware)
  - Portability over efficiency
  - Security over efficiency

C vs. Java: 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 enhanced with objects
- C has some shortcomings compared to C++
  - C++ has objects, a bigger standard library (e.g., STL), parameterized types, etc.
  - C++ is a little bit more strongly typed
- Programming Challenge
  - 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

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

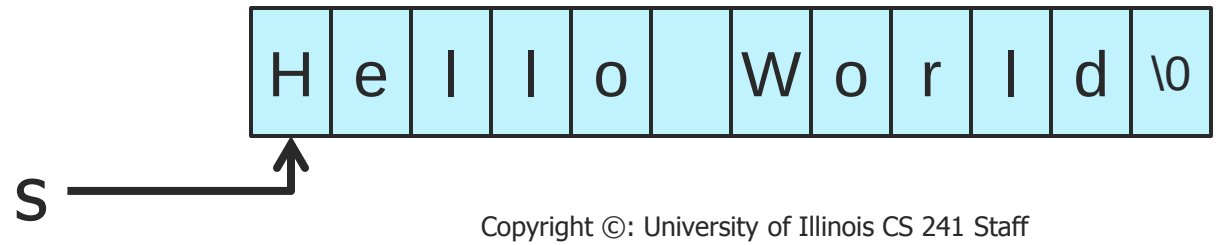
Theme:
how memory
really works

C vs. C++ Programming

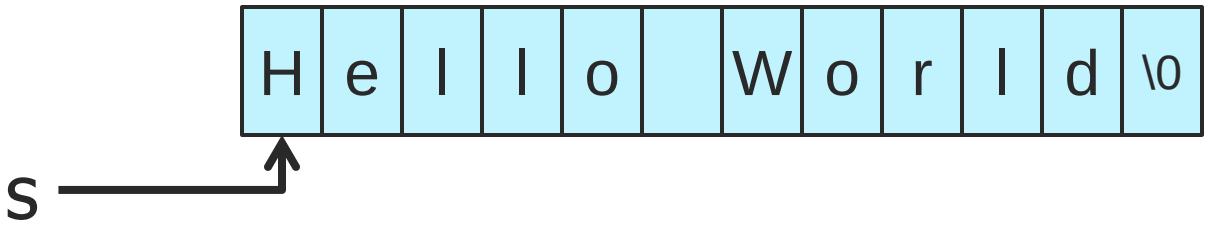
- Four key differences in programming:
  - C does not have a “string” type
    - **No** string s = “Hello”; s += “World”;
  - C does not have “iostream”
    - **No** cout<<“Hello World”<<endl;
  - C does not have new/delete:
    - **No** int *k = new int[4]; delete[] k;
  - C does not have classes

C Strings

- Strings in C
  - A string is simply an array of characters that end in a NULL (0x00, '\0') character.
  - Called "C Strings"
  - Data type is a **char pointer**, or char *
  - char *s = "Hello World";
  - In memory:

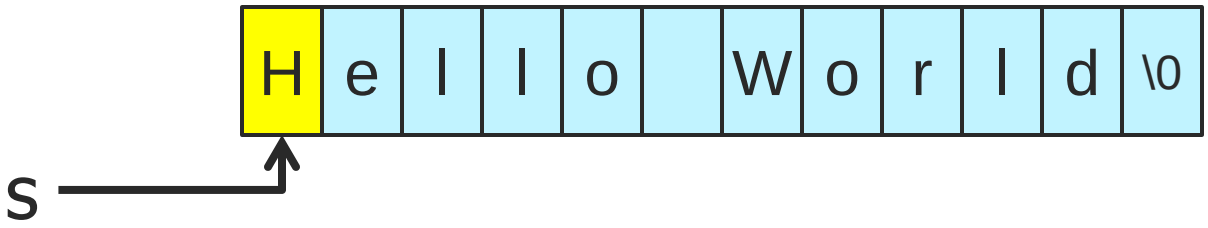


C Strings



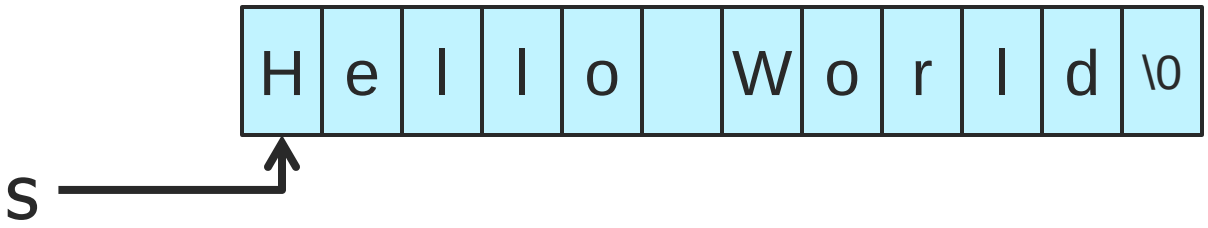
- Strings in C
  - char *s = “Hello World”;
  - s[0] ?

C Strings



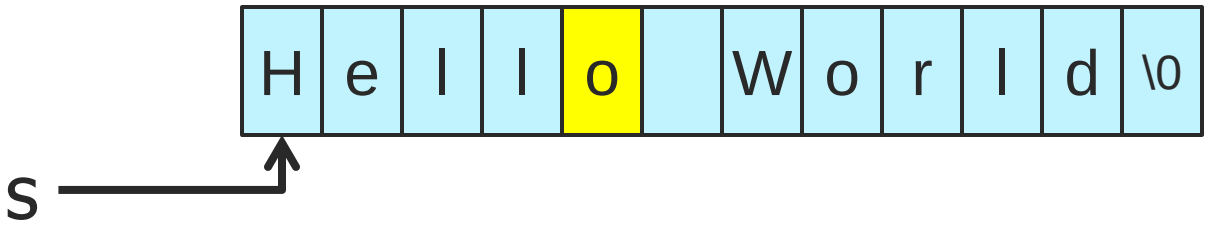
- Strings in C
  - char *s = “Hello World”;
  - s[0] ?

C Strings



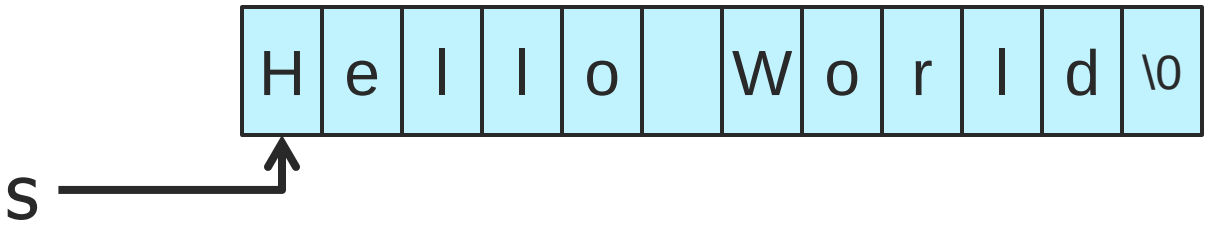
- Strings in C
  - char *s = “Hello World”;
  - s[4] ?

C Strings



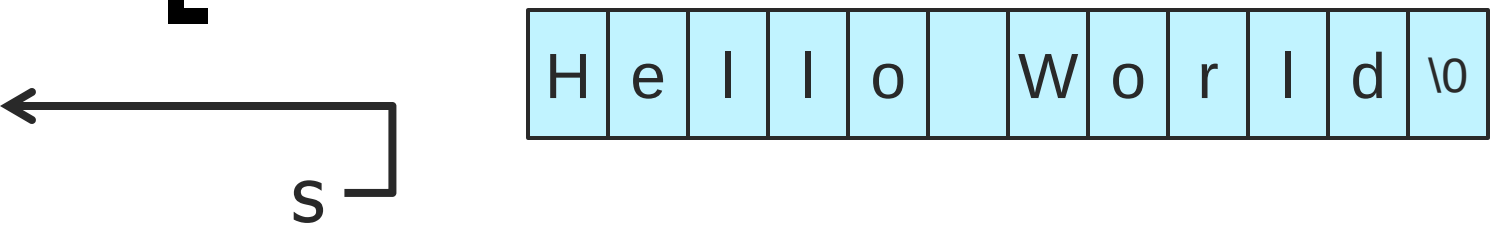
- Strings in C
  - char *s = “Hello World”;
  - s[4] ?

C Strings



- Strings in C
  - char *s = “Hello World”;
  - s = s + “Hi!!” ?

C Strings



- Strings in C
  - char *s = “Hello World”;
  - s = s + “Hi!!” ?

C Strings

- You can manipulate C Strings via library functions:
  - strcpy(): Copy a string
  - strcmp(): Compare two strings
  - strcat(): Concatenate two strings

  - **But you have to have enough memory to do it!**

A Few Differences between C and C++

- Input/Output
  - C++: `cout<<"hello world"<<endl;`
  - C: `printf("hello world\n");`

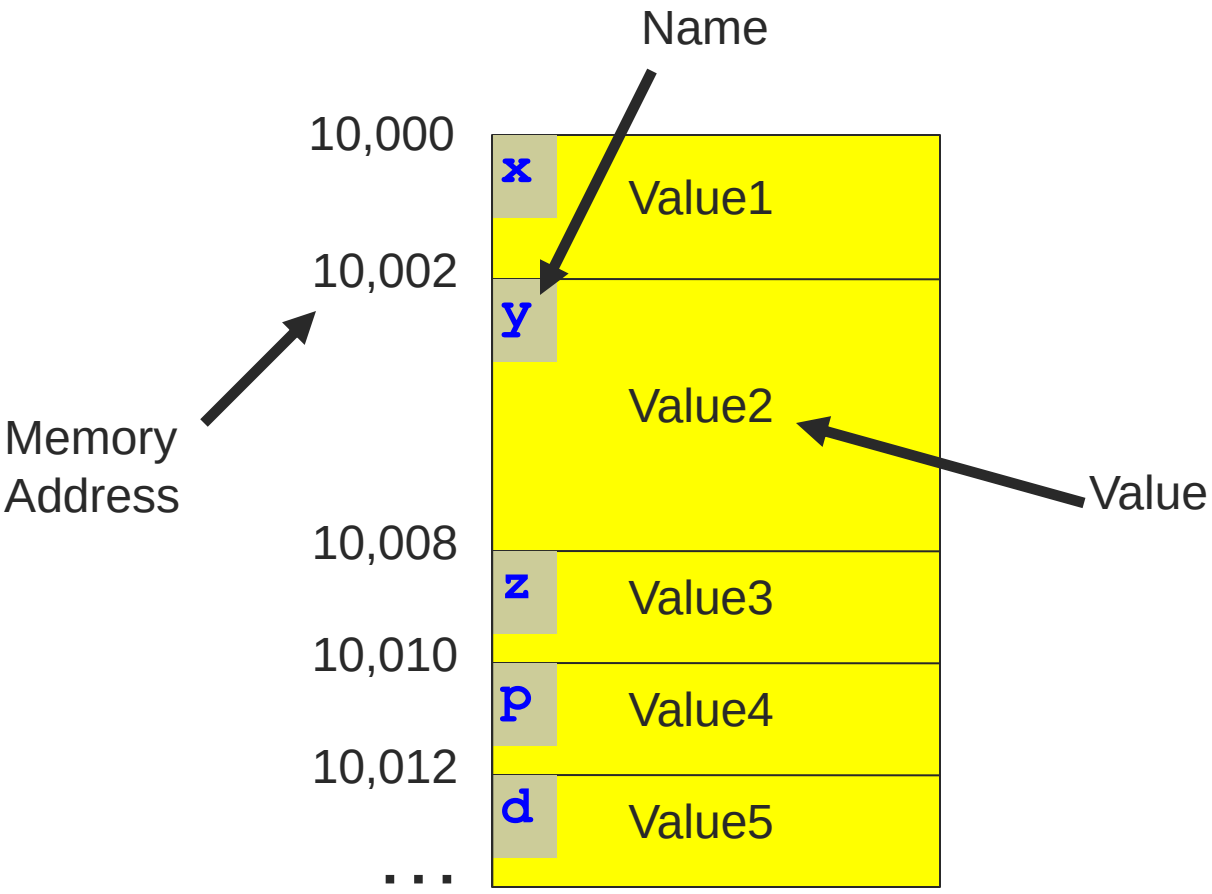
- Heap memory allocation
  - C++: **`new/delete`**
    - `int *x = new int[8]; delete[] x;`
  - C: **`malloc()/free()`**
    - `int *x = malloc(8 * sizeof(int));`
      `free(x);`

Pointers

Variables

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

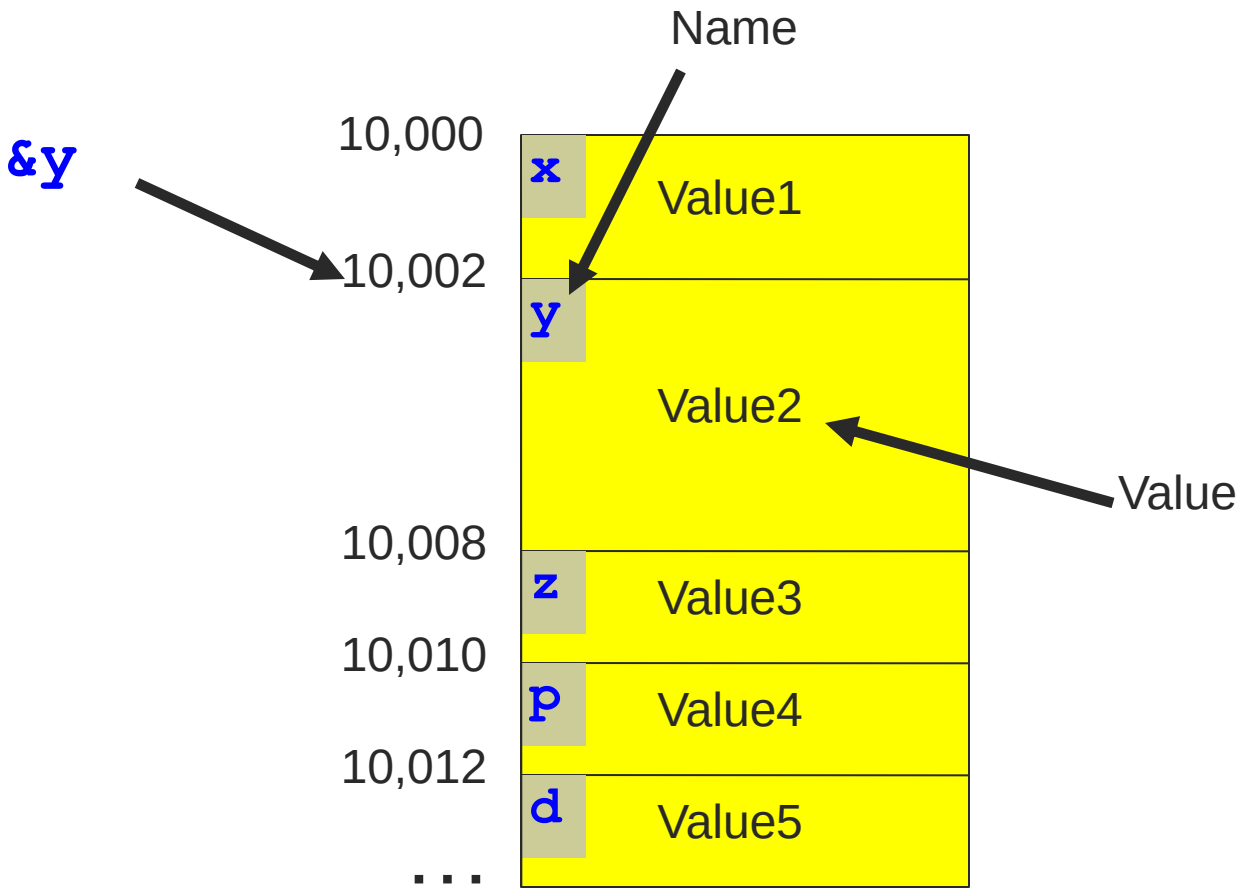

Variables



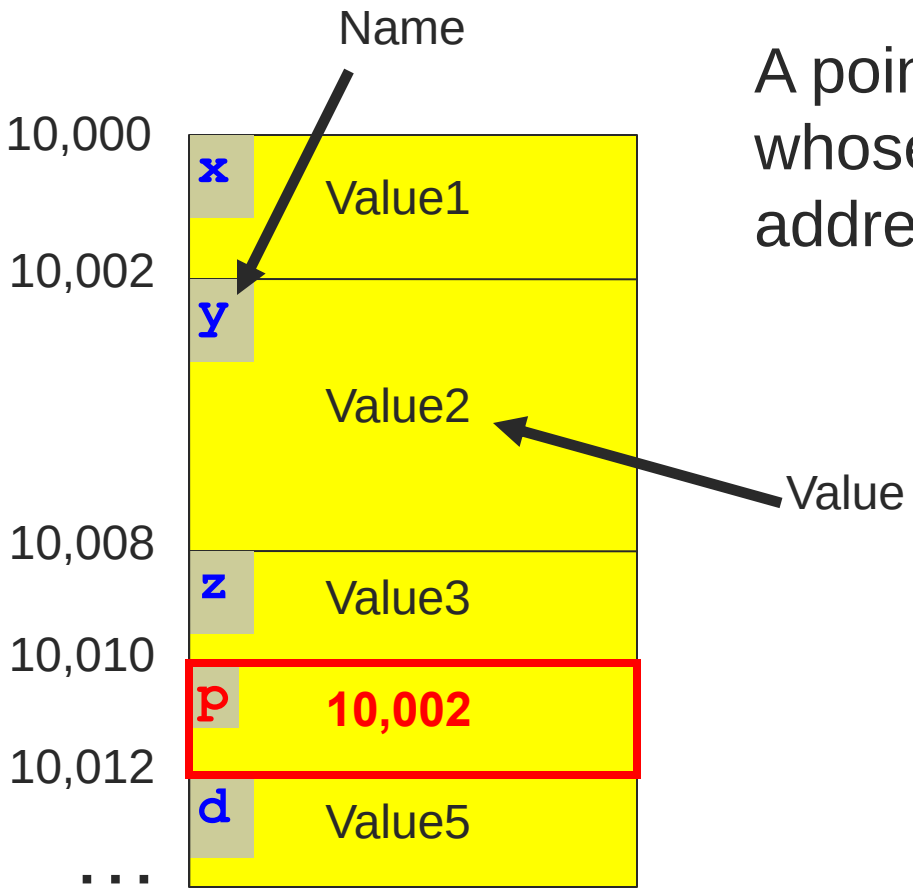
Type of each variable (also determines size)

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


The "&" Operator: Reads "Address of"

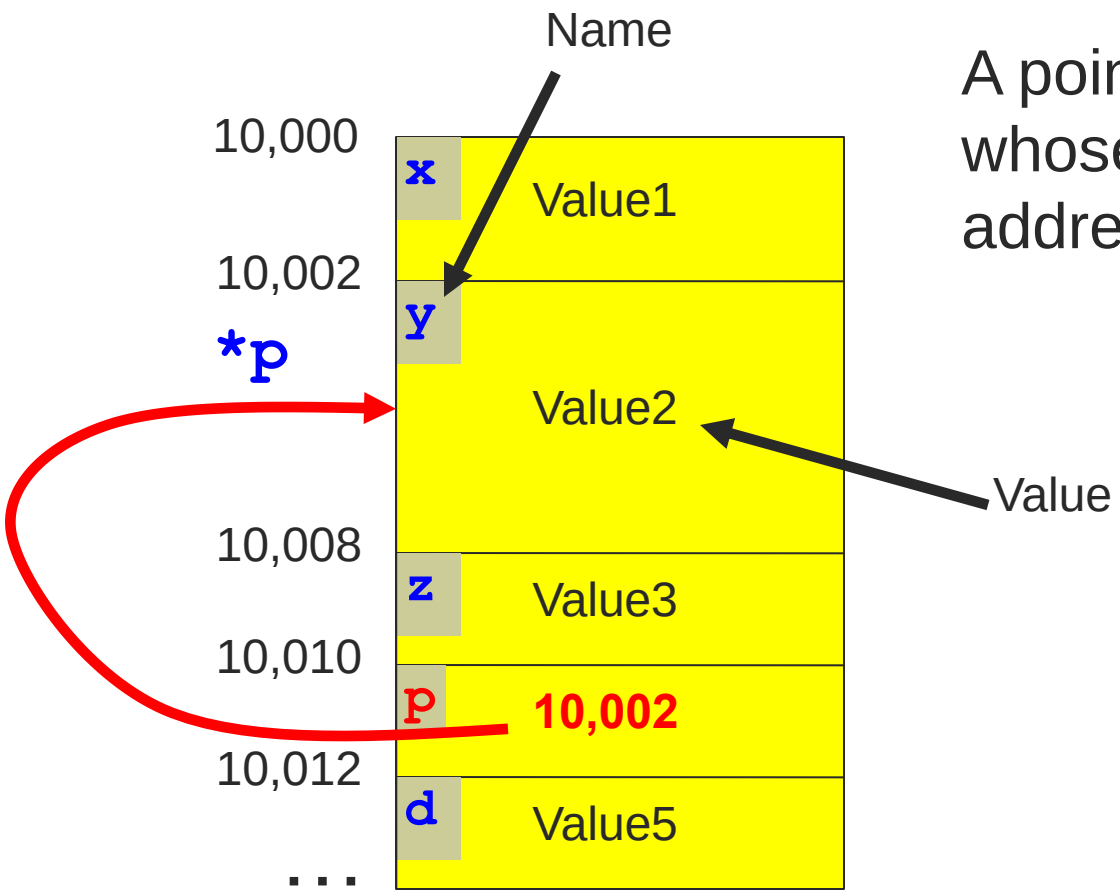


Pointers



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

What is the Output?

```
main() {
   int *p, q, x;
   x=10;
   p=&x;
   *p=x+1;
   q=x;
   printf ("Q = %d\n", q);
}
```


Cardinal Rule: Must Initialize Pointers before Using them

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

GOOD or BAD?

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
  - E.g., using “malloc” in C, “new” in Java
- Dynamically *deallocating* memory
  - Must reclaim or recycle memory that is never used again
  - To avoid (eventually) running out of memory
- “Garbage”
  - Allocated blocks in heap that will not be used again
  - Can be reclaimed for later use by the program

Option #1: Garbage Collection

- Run-time system does garbage collection (Java)
  - Automatically determines which objects can't be accessed
  - And then reclaims the resources used by these objects

```
Object x = new Foo() ;
Object y = new Bar() ;
x = new Quux() ;

if (x.check_something()) {
     x.do_something(y) ;
}

System.exit(0) ;
```

Object Foo() is never used again!

Challenges of Garbage Collection

- Detecting the garbage is not always easy
  - `long char z = x ;`
  - `x = new Quux();`
  - Run-time system cannot collect *all* the garbage
- Detecting the garbage introduces overhead
  - Keeping track of references to object (e.g., counters)
  - Scanning through accessible objects to identify garbage
  - Sometimes walking through a large amount of memory
- Cleaning the garbage leads to bursty delays
  - E.g., periodic scans of the objects to hunt for garbage
  - Leads to unpredictable “freezes” of the running program
  - Very problematic for real-time applications
    - … though good run-time systems avoid long freezes

Option #2: Manual Deallocation

- *Programmer* deallocates the 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
  - Can lead to security vulnerabilities in code

Manual deallocation can lead to bugs

- Dangling pointers
  - Programmer frees a region of 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();
}
```

Eventually, malloc() returns NULL

Manual deallocation can lead to bugs

- Double free
  - Programmer mistakenly frees a region more than once
  - Leading to corruption of the heap data structure
  - … or premature destruction of a different object

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

Might free space allocated by **q**!

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));
```

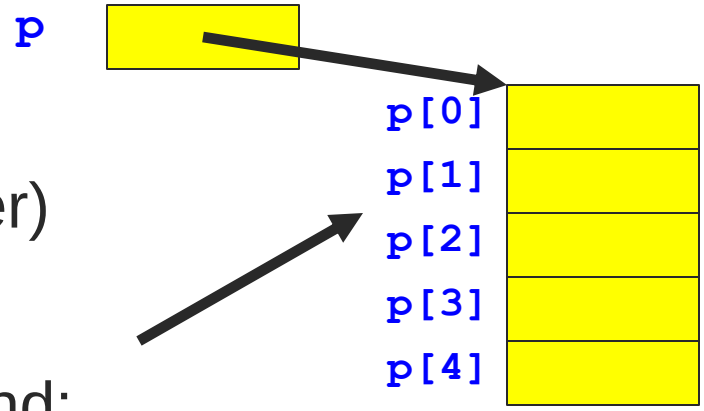
Is there a difference?

One is on the stack, one is on the heap

Arrays

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

Name of array (is a pointer)



Shorthand:

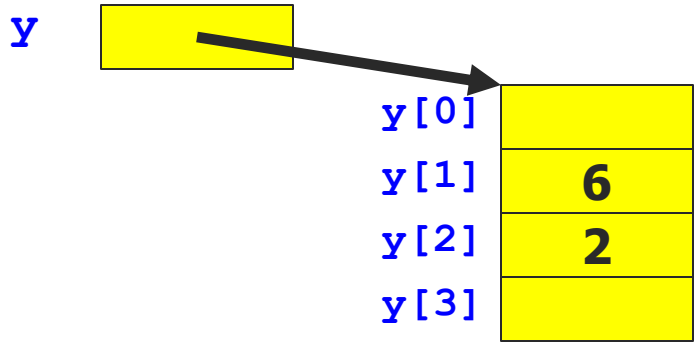
`*(p+1) is called p[1]`

`*(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];
```


Questions

- What's the difference between

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

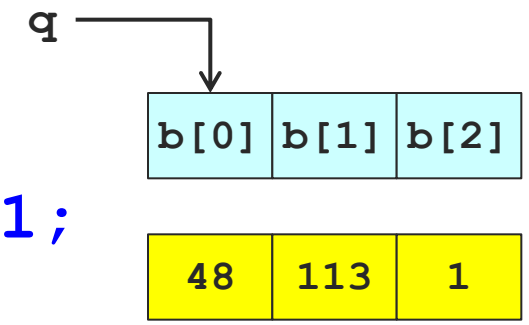
- What's wrong with

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


Questions

- What is the value of `b[2]` at the end?

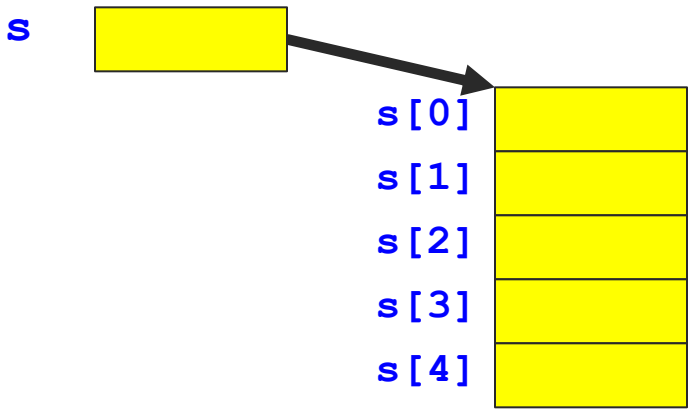
```
int b[3];
int* q;

b[0]=48; b[1]=113; b[2]=1;

q=b;

*(q+1)=2;

b[2]=*b;

b[2]=b[2]+b[1];
```



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

- `strcpy(ptr1, ptr2);`
  - `ptr1` and `ptr2` are pointers to char
- `value = strlen(ptr);`
  - `value` is an integer
  - `ptr` is a pointer to char

```
int len;
char str[15];
strcpy (str, "Hello,
   world!");
len = strlen(str);
```


strcpy, strlen

- What's wrong with

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


strncpy

- `strncpy(ptr1, ptr2, num);`
  - `ptr1` and `ptr2` are pointers to char
  - `num` is the number of characters to be copied

```
int len;
char str1[15],
   str2[15];
strcpy (str1,
   "Hello, world!");
strncpy (str2, str1,
   5);
```


strncpy

- `strncpy(ptr1, ptr2, num);`
  - `ptr1` and `ptr2` are pointers to char
  - `num` is the number of characters to be copied

```
int len;
char str1[15],
   str2[15];
strcpy (str1,
   "Hello, world!");
strncpy (str2, str1,
   5);
```

Caution: strncpy blindly copies the characters. It does not voluntarily append the string-terminating null character.

strcat

- `strcat(ptr1, ptr2);`
  - `ptr1` and `ptr2` are pointers to char

- Concatenates the two null terminated strings yielding one string (pointed to by `ptr1`).

```
char S[25] = "world!";
char D[25] = "Hello, ";
strcat(D, S);
```


strcat

- `strcat(ptr1, ptr2);`
  - `ptr1` and `ptr2` are pointers to char

- Concatenates the two null terminated strings yielding one string (pointed to by `ptr1`).
  - Find the end of the destination string
  - Append the source string to the end of the destination string
  - Add a NULL to new destination string

`strcat` Example

- What's wrong with

```
char S[25] = "world!";
strcat("Hello, ", S);
```


`strcat` Example

- What's wrong with

```
char *s = malloc(11 * sizeof(char));
        /* Allocate enough memory for an
           array of 11 characters, enough
           to store a 10-char long string. */
strcat(s, "Hello");
strcat(s, "World");
```


strcat

- `strcat(ptr1, ptr2);`
  - `ptr1` and `ptr2` are pointers to char
- Compare to Java and C++
  - `string s = s + " World!";`
- What would you get in C?
  - If you did `char* ptr0 = ptr1+ptr2;`
  - You would get the sum of two memory locations!

strcmp

- `diff = strcmp(ptr1, ptr2);`
  - `diff` is an integer
  - `ptr1` and `ptr2` are pointers to char
- Returns
  - zero if strings are identical
  - < 0 if `ptr1` is less than `ptr2` (earlier in a dictionary)
  - > 0 if `ptr1` is greater than `ptr2` (later in a dictionary)

```
int diff;
char s1[25] = "pat";
char s2[25] = "pet";
diff = strcmp(s1, s2);
```


Can we make this work?!

```
int x;
```

```
printf("This class is %s.\n", &x);
```


Can we make this work?!

```
int x;
```

```
printf("This class is %s.\n",   );
```


Can we make this work?!

```
int x;

 (char*)&x
```

```
printf("This class is %s.\n", &x);
```


Can we make this work?!

```
int x;

((char*)&x)[0] = 'f';



printf("This class is %s.\n", &x);
```


Can we make this work?!

```
int x;

((char*)&x)[0] = 'f';
((char*)&x)[1] = 'u';
((char*)&x)[2] = 'n';



printf("This class is %s.\n", &x);
```


Can we make this work?!

```
int x;

((char*)&x)[0] = 'f';
((char*)&x)[1] = 'u';
((char*)&x)[2] = 'n';
((char*)&x)[3] = '\0';


printf("This class is %s.\n", &x);
```

Perfectly legal and perfectly horrible!

Can we make this work?!

```
int x;

char* s = &x;
strcpy(s, "fun");
```

Perfectly legal and perfectly horrible!

```
printf("This class is %s.\n", &x);
```


Other operations

Increment & decrement

- **`x++`**: yield old value, add one
- **`++x`**: add one, yield new value

```
int x = 10;
x++;
int y = x++;
```

11

```
int z = ++x;
```

13

- **`--x`** and **`x--`** are similar (subtract one)

Math: Increment and Decrement Operators

- Example 1:

```
int x, y, z, w;
y=10; w=2;
x=++y;
z=--w;
```

- Example 2:

```
int x, y, z, w;
y=10; w=2;
x=y++;
z=w--;
```

What are **x** and **y** at the end of each example?

Math: Increment and Decrement Operators on Pointers

- Example 1:

```
int a[2];
int number1, number2, *p;
a[0]=1; a[1]=10;
p=a;
number1 = *p++;
number2 = *p;
```

- What will **`number1`** and **`number2`** be at the end?

Math: Increment and Decrement Operators on Pointers

- Example

```
int a[2];
int number1, number2, *p;
a[0]=1; a[1]=10;
p=a;
number1 = *p++;
number2 = *p;
```

Hint: ++ increments pointer p not variable *p

- What will **`number1`** and **`number2`** be at the end?

Logic: Relational (Condition) Operators

`==`	equal to
`!=`	not equal to
`>`	greater than
`<`	less than
`>=`	greater than or equal to
`<=`	less than or equal to

Review

Review

- `int p1;`
  What does `&p1` mean?

Review

- How much is `y` at the end?

```
int y, x, *p;

x = 20;
*p = 10;
y = x + *p;
```


Review

- What are the differences between **x** and **y?**

```
char* f() {
   char *x;
   static char*y;
   return y;
}
```


Review: Debugging

```
if(strcmp("a","a"))
  printf("same!");
```


Review: Debugging

```
int i = 4;
int *iptr;
iptr = &i;
*iptr = 5;//now i=5
```


Review: Debugging

```
char *p;
p=(char*)malloc(99);
strcpy("Hello",p);
printf("%s World",p);
free(p);
```


Review: Debugging

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


Operator	Description	Associativity
() [] . -> ++ --	Parentheses (function call) Brackets (array subscript) Member selection via object name Member selection via pointer Postfix increment/decrement	left-to-right
++ -- + - ! ~ (type) * & sizeof	Prefix increment/decrement Unary plus/minus Logical negation/bitwise complement Cast (change type) Dereference Address Determine size in bytes	right-to-left
* / %	Multiplication/division/modulus	left-to-right
+ -	Addition/subtraction	left-to-right
<< >>	Bitwise shift left, Bitwise shift right	left-to-right
< <= > >=	Relational less than/less than or equal to Relational greater than/greater than or equal to	left-to-right
== !=	Relational is equal to/is not equal to	left-to-right
&	Bitwise AND	left-to-right
^	Bitwise exclusive OR	left-to-right
|	Bitwise inclusive OR	left-to-right
&&	Logical AND	left-to-right
||	Logical OR	left-to-right
?:	Ternary conditional	right-to-left
= += -= *= /= %= &= ^= |= <<= >>=	Assignment Addition/subtraction assignment Multiplication/division assignment Modulus/bitwise AND assignment Bitwise exclusive/inclusive OR assignment Bitwise shift left/right assignment	right-to-left
,	Comma (separate expressions)	left-to-right