C Survival Guide

Good news: Writing C code is easy!

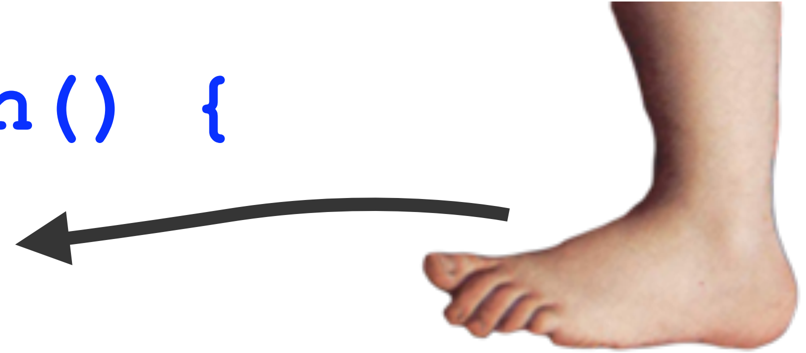
```
void* myfunction() {
    char *p;
    *p = 0;
    return (void*) &p;
}
```

Bad news: Writing BAD C code is easy!

```
void* myfunction() {
    char *p;
    *p = 0;
    return (void*) &p;
}
```

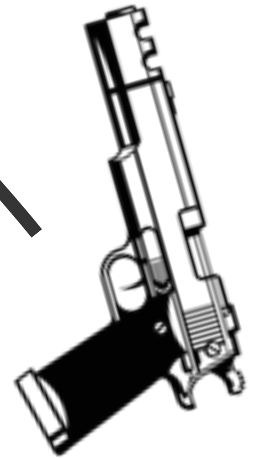
Bad news: Writing BAD C code is easy!

```
void* myfunction() {
    char *p;
    *p = 0;
    return (void*) &p;
}
```



Bad news: Writing BAD C code is easy!

```
void* myfunction() {
    char *p;
    *p = 0;
    return (void*) &p;
}
```



Copyright ©: University of Illinois CS 241 Staff

How do I write good C programs?

- Fluency in C syntax
- Stack vs. Heap
- Key skill: read code for bugs
  - Do not rely solely on compiler warnings, if any, and testing

- C is powerful - it's the System Programmer's choice language

Copyright ©: University of Illinois CS 241 Staff

The C Language Spirit

- Made by professional programmers for professional programmers
- Very flexible, very efficient and portable
  - Does not protect the programmers from themselves.
  - Rationale: programmers know what they are doing.
- UNIX and most “serious” system software (servers, compilers, etc) are written in C.
- Can do everything Java and C++ can. But complex tasks could look ugly in C.

C vs. C++

- Problem
  - Object oriented languages provided nice features to programmers, but were very, very slow
- Solution
  - The development of C++
  - C enhanced with objects
- 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

Copyright ©: University of Illinois CS 241 Staff

Key Differences between C and C++

- Input/Output
  - C does not have "iostreams"
  - C: `printf("hello world\n");`
  - C++: `cout<<"hello world"<<endl;`
- Heap memory allocation
  - C: `malloc()/free()`
    - `int *x = malloc(8 * sizeof(int));`
      `free(x);`
  - C++: `new/delete`
    - `int *x = new int[8]; delete(x);`

Copyright ©: University of Illinois CS 241 Staff

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 compilation utility
  - Google 'makefile'

Copyright ©: University of Illinois CS 241 Staff

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

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

Name

Address	Name	Value
10,000	x	Value1
10,002	y	Value2
10,008	z	Value3
10,010	p	Value4
10,012	d	Value5
…		

Memory Address

Value

Type of each variable (also determines size)

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

Copyright ©: University of Illinois CS 241 Staff

The “&” Operator: Reads “Address of”

10,000	x	Value1
10,002	y	Value2
10,008	z	Value3
10,010	p	Value4
10,012	d	Value5
...		

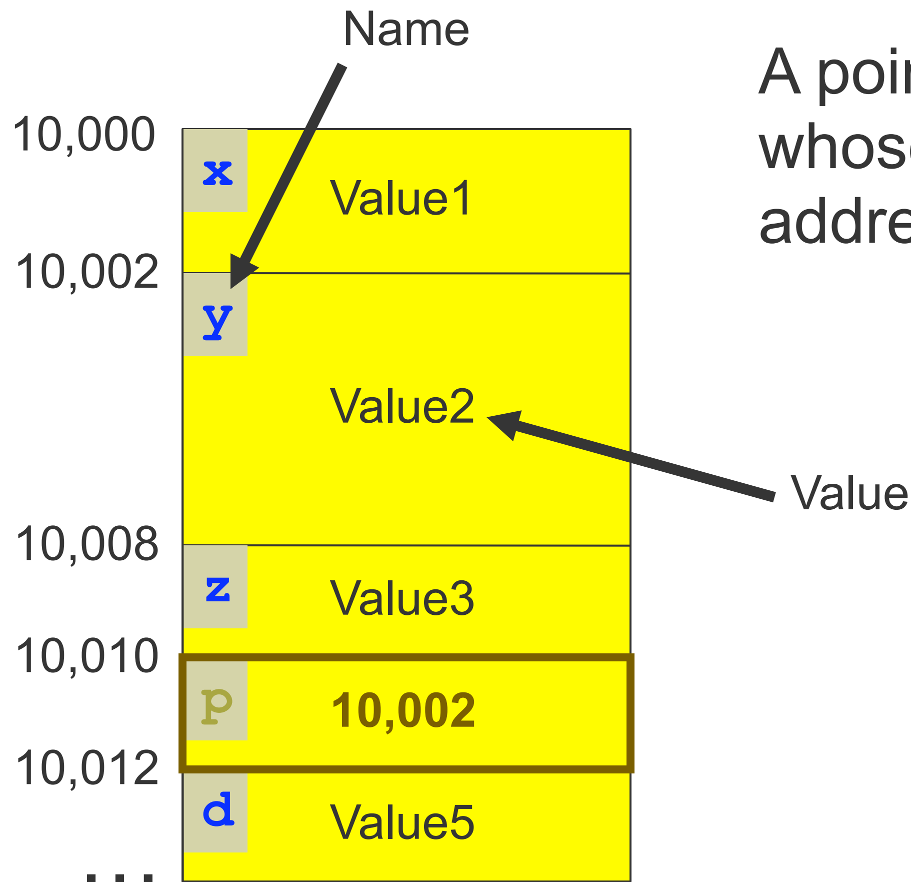
&y → 10,002

Name → y

Value → Value2

Copyright ©: University of Illinois CS 241 Staff

Pointers



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

Copyright ©: University of Illinois CS 241 Staff

The “*” Operator Reads “Variable pointed to by”

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

Name

Address	Name	Value
10,000	x	Value1
10,002	y	Value2
10,008	z	Value3
10,010	p	**10,002**
10,012	d	Value5
…		

*p

Value

Copyright ©: University of Illinois CS 241 Staff

What is the Output?

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

Copyright ©: University of Illinois CS 241 Staff

What is the Output?

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

p	#@*%!

q	#@%$!

x	@*%^

What is the Output?

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

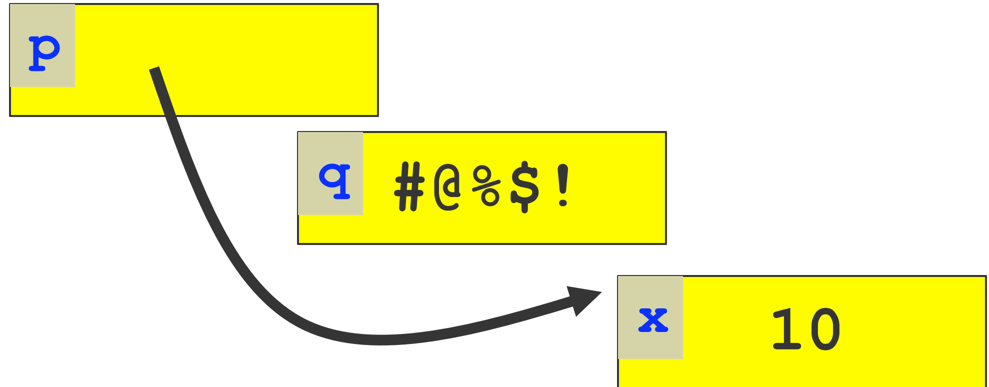
p	#@*%!

q	#@%$!

x	10

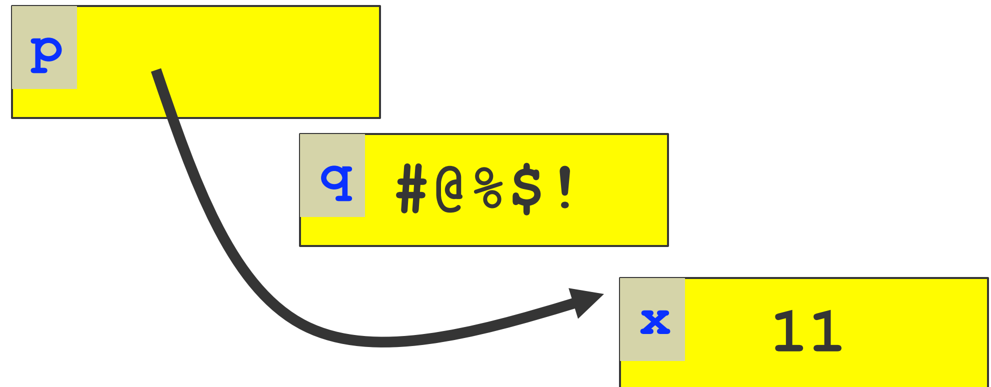
What is the Output?

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



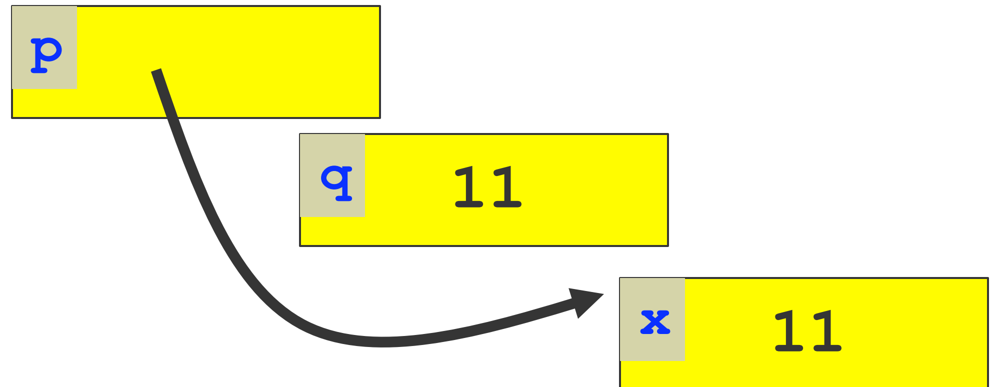
What is the Output?

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



What is the Output?

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



Copyright ©: University of Illinois CS 241 Staff

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!

Copyright ©: University of Illinois CS 241 Staff

Cardinal Rule: Must Initialize Pointers before Using them

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

Copyright ©: University of Illinois CS 241 Staff

How to Initialize Pointers

- **Use existing memory:** Set pointer equal to location of known variable

```
int *p;
int x;
…
p=&x;
```

- **Allocate new memory** -- how?

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

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);
```

Example

```
int* p;

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

*p = 5;

free(p);
```

Copyright ©: University of Illinois CS 241 Staff

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

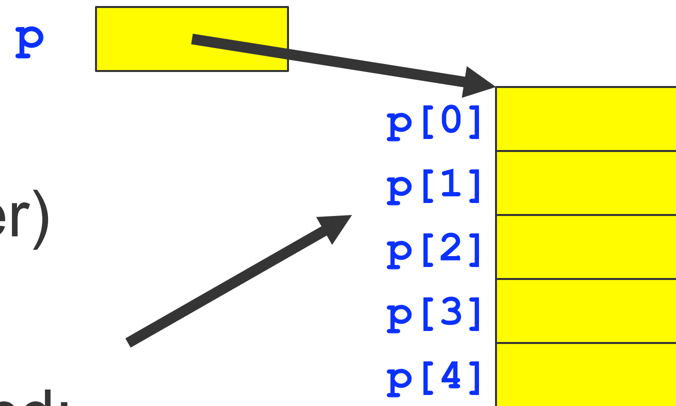
Arrays

- Contiguous block of memory to fit one or more elements of some type
- Two ways to allocate:
  - named variable: `int x[10];`
  - dynamically:
    ```
    int* x = (int*) malloc(10*sizeof(int));
    ```

Arrays

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

Name of array (is a pointer)



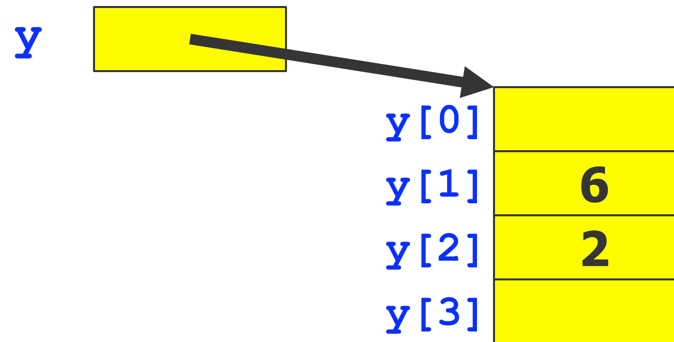
Shorthand:

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

etc..

Example

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



Copyright ©: University of Illinois CS 241 Staff

Array Name as Pointer

What's the difference between the examples below

- Example 1:

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

- Example 2:

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

Copyright ©: University of Illinois CS 241 Staff

Array Name as Pointer

What's the difference between the examples below

- Example 1:

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

- Example 2:

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

NOTHING!!

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

Copyright ©: University of Illinois CS 241 Staff

Questions

- What's the difference between

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

- What's wrong with

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

Copyright ©: University of Illinois CS 241 Staff

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];
```

Copyright ©: University of Illinois CS 241 Staff

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];
```

48	113	1

Copyright ©: University of Illinois CS 241 Staff

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];
```

48	113	1

48	2	1

Copyright ©: University of Illinois CS 241 Staff

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;
```

48	113	1

```
q=b;
```

```
*(q+1)=2;
```

48	2	1

```
b[2]=*b;
```

48	2	48

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

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;
```

48	113	1

```
q=b;
```

```
*(q+1)=2;
```

48	2	1

```
b[2]=*b;
```

48	2	48

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

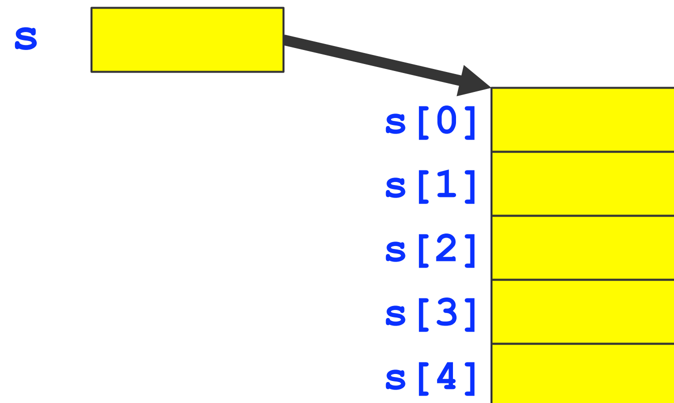
48	2	50

Copyright ©: University of Illinois CS 241 Staff

- 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];`



Copyright ©: University of Illinois CS 241 Staff

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);
```

Copyright ©: University of Illinois CS 241 Staff

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.

Copyright ©: University of Illinois CS 241 Staff

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);
```

Copyright ©: University of Illinois CS 241 Staff

`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

Copyright ©: University of Illinois CS 241 Staff

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");
```

Copyright ©: University of Illinois CS 241 Staff

strcat

Copyright ©: University of Illinois CS 241 Staff

strcat

- `strcat(ptr1, ptr2);`

strcat

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

Copyright ©: University of Illinois CS 241 Staff

`strcat`

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

Copyright ©: University of Illinois CS 241 Staff

strcat

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

Copyright ©: University of Illinois CS 241 Staff

strcat

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

- Compare to Java
  - `string s = s + " World!";`

strcat

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

- Compare to Java
  - `string s = s + " World!";`

Copyright ©: University of Illinois CS 241 Staff

strcat

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

- Compare to Java
  - `string s = s + " World!";`

- What would you get in C?

strcat

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

Copyright ©: University of Illinois CS 241 Staff

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);
```

Copyright ©: University of Illinois CS 241 Staff

Can we make this work?!

```
int x;

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

Can we make this work?!

```
int x;

 (char*)&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",     );
```

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",        );
```

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",      );
```

Perfectly legal and perfectly horrible!

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",      );
```

Can we make this work?!

```
int x;

char* s = &x;

strcpy(s, “fun”);


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

Perfectly legal and perfectly horrible!

Other operations

Increment & decrement

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

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

int z = ++x;
```

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

Increment & decrement

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

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

int z = ++x;
```

11

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

Increment & decrement

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

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

int z = ++x;
```

11

13

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

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?

Copyright ©: University of Illinois CS 241 Staff

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;
```

Hint: ++ increments pointer p not variable *p

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

Copyright ©: University of Illinois CS 241 Staff

Logic: Relational (Condition) Operators

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

Copyright ©: University of Illinois CS 241 Staff

Logic Example

```
if (a == b)
   printf (“Equal.”);
else
   printf (“Not Equal.”);
```

- Question: what will happen if I replaced the above with:

```
if (a = b)
   printf (“Equal.”);
else
   printf (“Not Equal.”);
```

Copyright ©: University of Illinois CS 241 Staff

Logic Example

```
if (a == b)
   printf ("Equal.");
else
   printf ("Not Equal.");
```

- Question: what will happen if I replaced the above with:

```
if (a = b)
   printf ("Equal.");
else
   printf ("Not Equal.");
```

Perfectly LEGAL C statement!
(syntactically speaking)
It copies the value in **b** into **a**.
The statement will be interpreted as **TRUE** if **b** is non-zero.

Copyright ©: University of Illinois CS 241 Staff

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

- How much is `y` at the end?

```
int y, x, *p;

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

BAD!!
Dereferencing an uninitialized pointer will likely segfault or overwrite something!

Segfault = unauthorized memory access

Copyright ©: University of Illinois CS 241 Staff

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
```

Copyright ©: University of Illinois CS 241 Staff

Review: Debugging

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

Copyright ©: University of Illinois CS 241 Staff

Review: Debugging

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

Copyright ©: University of Illinois CS 241 Staff

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

Copyright ©: University of Illinois CS 241 Staff