

(Relatively) Speedy Introduction to C



University of Victoria
Department of Computer Science

SENG265: Software Development Methods
C Language (part 1): Slide 1

Rudiments

- History
- Features
- How do I use C tools?
- Programming style
- C data types
 - Basic data types
- Literals
- Storage class
- Scalar variable definitions



History

- 1972 : Dennis Ritchie
 - developed as a convenient way of accessing the machine instruction set
 - produced efficient machine code
- 1973 : Ritchie and Ken Thompson
 - rewrite the UNIX kernel using C
 - portability of the UNIX kernel's code was a goal
- 1977 : Ritchie and Brian Kernighan
 - “The C Programming Language”
 - K&R dialect of C
- AT&T releases PCC : Portable C Compiler
 - *de facto* “standard” starts to break down as vendors begin to “extend” their C compilers in non-portable ways



History (2)

- 1983 : James Brodie (Motorola) applies to X3 committee of ANSI to draft a C standard
 - ANSI (American National Standards Institute)
 - results in X3J11 – C Programming Language Committee
- 1985 : AT&T (Bjarne Stroustrup) and first release of C++
 - (But that's another story)
- 1987,1989
 - ANSI Standard C defined simultaneously with ISO
 - ISO (International Standards Organization)
 - Committee JTC1 SC22 WG14
- we will be focusing on a version referred to as C99
 - GNU toolchain will be our workhorse (gcc version 9)



```
/*
 * mywc.c: not-quite-so-robust version of "wordcount"
 */

#include <ctype.h>
#include <stdio.h>
#include <string.h>
#include <stdlib.h>

#define MAX_LINE_LEN 256

int main (int argc, char **argv) {
    FILE *infile;
    char line[MAX_LINE_LEN];

    int num_chars = 0;
    int num_lines = 0;
    int num_words = 0;

    char *c;

    if (argc < 2) {
        fprintf(stderr, "usage: %s filename\n", argv[0]);
        exit(1);
    }

    infile = fopen(argv[1], "r");
    if (infile == NULL) {
        fprintf(stderr, "%s: cannot open %s", argv[0], argv[1]);
        exit(1);
    }

    /* continued on next slide with same indentation */
}
```

int argc, char **argv

This is new

fprintf?

```
/* continued from previous slide */

while (fgets(line, MAX_LINE_LEN-1, infile) != NULL) {
    num_lines += 1;
    num_chars += strlen(line);
    if (strncmp(line, "", MAX_LINE_LEN) != 0) {
        num_words++;
    }
    for ( c = line; *c; c++ ) {
        /* Not quite good enough!! */
        if (isspace(*c)) {
            num_words++;
        }
    }
}
fclose(infile);

printf ("%s: %d %d %d\n", argv[1],
       num_lines, num_words, num_chars);

return 0; /* return the success code */
}
```

Features

- A "general purpose" language equally usable for applications programming and systems
 - develop a network protocol
 - develop a database management system
 - write a compiler for another language
- It is ubiquitous: where you find UNIX you usually find C
- It provides a basis for understanding other languages, most notably C++



Features

- Most C toolchains have a relatively small footprint
 - popular choice for developing **embedded systems**
 - operating systems research and development
 - good choice for systems programs that one expects to port
- Compile-time features
 - ANSI-compliant compilers provide extensive compile-time diagnostics
 - ANSI-compliant compilers provide a continuum of optimizations; from **none** to **conservative** to **aggressive**



Features

- Run-time features (i.e., “pluses”):
 - easy to adapt a C compiler’s output (executables) to the execution environment on a platform:
Windows, Mac, UNIX
- Run-time features missing (i.e., also could be considered as “efficiency pluses”!):
 - no native array access bounds checking
 - no native null-pointer checks (use a custom library for this)
 - no native checks on uninitialized variables (but some scenarios can be checked at compile-time)



How do I use C tools?

- Write an application

```
$ cat > hello.c
#include <stdio.h>
int main() {
    printf("Hello, world!\n");
}
^D
```

- Compile the source file into an object file

```
$ gcc -c hello.c
```

- Link the object file to the “Standard C Runtime Library” to produce an executable (hello)

```
$ gcc hello.o -o hello
```

```
$ gcc hello.o -o hello -lm # Version which links in math libraries
```



How do I use C?

- Much terser syntax going from source code straight to executable (assuming executable needs to link a single object file)

```
$ gcc hello.c -o hello
```

- Assuming we want to also include debugging symbols:

```
$ gcc -g hello.c -o hello
```

- Specify some warning flags:

```
$ gcc -g -std=c99 -Wall hello.c -o hello
```



How do I use C ?

- Run the executable:

```
$ ./he11o  
He11o, Wor1d!  
$
```

- Basic rules:
 - all C stand-alone programs must have at most one function named **main()**
 - keywords are always lowercase; you cannot use a keyword as an identifier
 - statements must be terminated with a semicolon

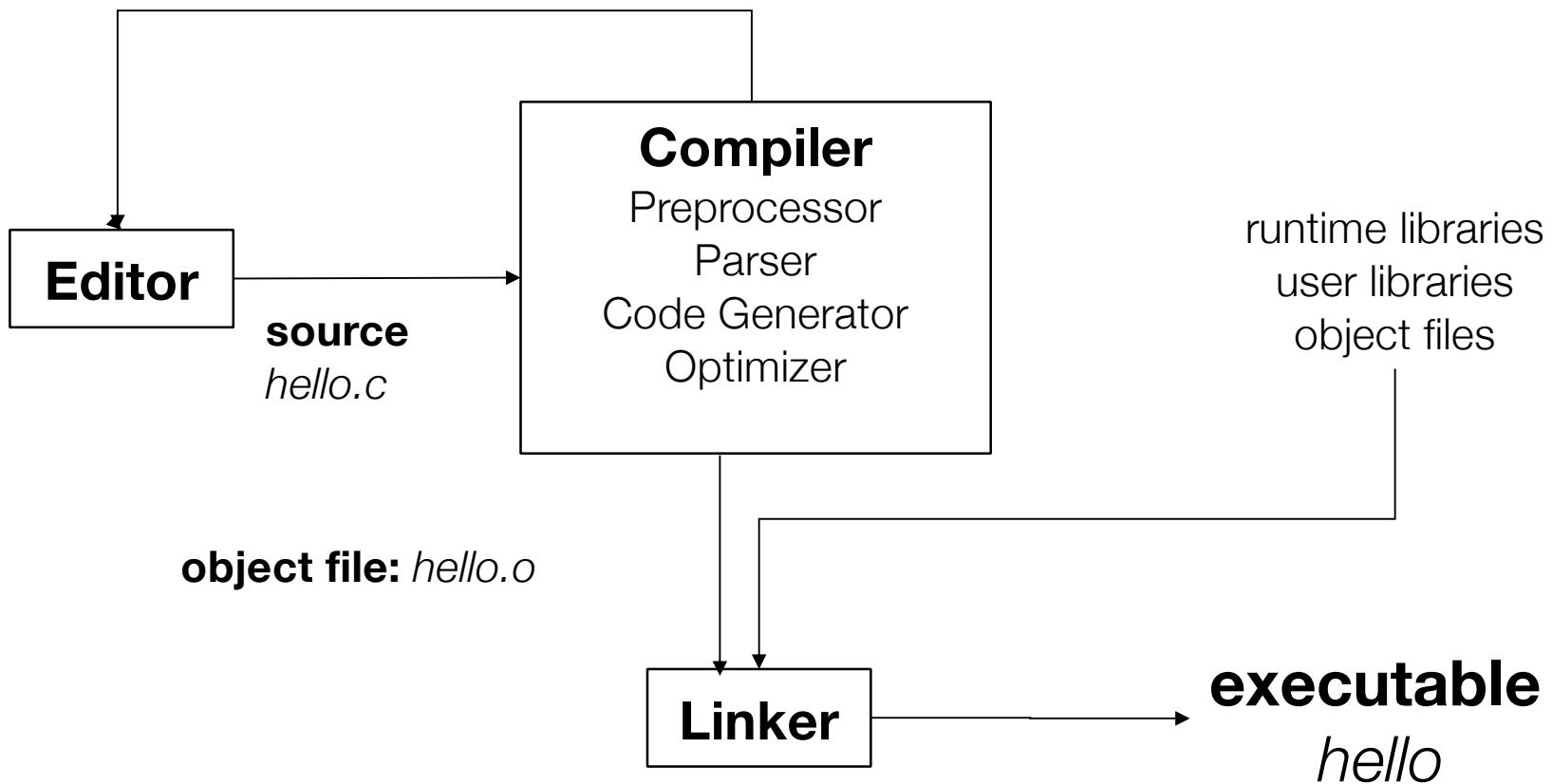


How do I use C ?

- Basic rules (continued):
 - Comments are delimited by `/* ... */`
`/* Everything between "slash star" and
"star slash" is a comment, even if it
spans several lines. Be careful not
to nest comments; some compilers are
unable to handle them. */`
 - Single line comments are not ANSI C (`//`)
- **Upcoming labs:**
 - introduce the GNU toolchain
 - aspects of the C execution model



How do I use C ?



A word about formatting style

- Any amount of white space is considered a single space
 - tabs and spaces can be used liberally
- White space improves code readability
- Commenting is important as a maintenance tool
- Use tabbing in conjunction with curly braces (**{ }**) to indicate different levels of nested functions.
- In C, type declarations must appear at the beginning of a scope
- Scope begins and ends with curly braces (**{ }**)
- Use **snake_case_variable_names** rather than **someCamelCaseVariableNames**



C data types

- These behave largely as you expect from other languages
 - Java integers, floats, etc.
- Because of the C language problem domain, lots of other additional terms
 - Modifiers, specifiers, storage “classes” (which are not like Java classes)
 - These terms can cause confusion...
 - ... but in this course you are not expected to master all of the terms



type	modifiers (precision / range)
char	signed, unsigned
int	signed, unsigned, short, long, long long
float	
double	long

int, int, int, int, float, double

type	purpose / intent
struct	(for heterogeneous aggregate types)
{arrays}	(for homogeneous aggregate types)
enum	(assign names to integral constants)
union	(tricky to use, but nice when needed)

storage	purpose / intent
static	controls variable “linkage”
extern	controls variable “linkage”
auto	(now assumed as default)
register	(rarely used as compiler does it better)

qualifier	purpose / intent
const	compiler enforces non-modification
volatile	compiler avoids optimizing access

Literals: examples

```
/* Character constants in 8-bit ASCII */
char ch = 'A';
char bell = '\a';
char formfeed = '\f';
```

```
/* But chars are also just 8-bit bytes */
char c = 65; /* Same as 'A' */
```

```
/* Integer literals */
int a = 10;
int b = 0x1CE;
int c = 0777;

unsigned int x = 0xffffU;
long int y = 2L;
```

```
/* Floating-point literals */
float x = 3.14159f;
double x = 1.25, y = 2.5E10, z = -2.5e-10;

long double x = 3.5e3L;
```

uses  , instead of " "

Danger! String literals

- String literals

```
char *s = "unable to open file\n";
```

- We will get to C strings in due course, but here is an early warning:
 - The variable “**s**” above might appear to act like a string...
 - ... but it is actually a variable holding an address to a “static string table”
 - This part of the process's memory is read-only!



Example

```
#include <stdio.h>

int main() {
    char s[50] = "abcdefghijklmnopqrstuvwxyz";
    char *t = "zyxwvutsrqponmlkjihgfedcba";

    printf("message s is: '%s'\n", s);
    s[0] = ' ';
    s[1] = ' ';
    printf("modified message s is: '%s'\n", s);

    printf("message t is: '%s'\n", t); /* next two lines will fail */
    t[0] = ' ';
    t[1] = ' ';
    printf("modified message s is: '%s'\n", t);
}
```

```
$ ./staticstring
message s is: 'abcdefghijklmnopqrstuvwxyz'
modified message s is: ' cdefghijklmnopqrstuvwxyz'
message t is: 'zyxwvutsrqponmlkjihgfedcba'
Bus error: 10
```



Example

```
#include <stdio.h>

int main() {
    char person_A[50] = "Anakin Skywalker";
    char *person_B = "Darth Vader";

    printf("%s; %s\n", person_A, person_B);

    person_A[1] = '\0';
    printf("%s\n", person_A);

    person_B[1] = '\0';
    printf("%s\n", person_B);
}
```

```
$ ./fubar
Anakin Skywalker; Darth Vader
A
Segmentation fault (core dumped)
```



C data types

- We will not go into all of the C type system in exhaustive detail ...
- ... but rather will examine – via examples – the way data types are used in practice.
- Deep knowledge is essential for some programmers, though!
 - For example: those who write C compilers!
- Our focus in this course is getting you comfortable with using C data types in a straightforward way during your problem solving.
 - We will return to different type-system elements and their explanation when they are needed to understand the bigger picture.



Introduction to C Programming (cont)

- C arrays in a bit more detail
- Control flow
- A bit about functions
- User-defined types
 - Aggregate data type: structures (**struct**)
 - type synonyms (**typedef**)
 - (briefly) enumerations (**enum**)
- Simple I/O



C Arrays

- An array is a group of data elements of the same type, accessed using the same identifier; e.g., **X[3], X[11]**
- Arrays may be statically or dynamically allocated. Static arrays cannot grow at runtime. Dynamic arrays can grow at runtime (using standard library functions).
- Arrays may be multidimensional; e.g., **X[row][column]**
- Access to the elements of an array is accomplished using integer indices
- If an array is dimensioned to hold **size** elements, the elements are indexed from **0** up to **size-1**
- **C provides no array bounds checking**, so accessing elements beyond index **size-1**, or below index **0** can cause a segmentation fault
- Static arrays can be auto-initialized at runtime



C Arrays (2)

- syntax for a one-dimensional array declaration:
`<storage class> <type> <identifier>[<size>]`
e.g. `double vector[3];`
`char buffer[256];`
- `<size>` must be known at compile time
- **`<size>` is not a part of an array data structure.** Programmer has to manage correct access to array!
- Examples:

```
double f[3] = {0.1, 2.2, -100.51};

int freq[10] = {20,12}; /* freq[0] = 20,
                           freq[1] = 12,
                           freq[2] = 0,
                           freq[9] = 0 */
```



Control Flow

- five basic flow control statements:
 - **if-then**, **if-then-else** (conditional)
 - **switch** (multi-branch conditional)
 - **while** loops (iteration, top-tested)
 - **do-while** loops (iteration, bottom-tested)
 - **for** loops (iteration)
- Other control flow constructs:
 - “**goto**”, there are many reasons not to use this, so we won’t (use “**continue**” and “**break**” instead);
 - “**setjmp/longjmp**”, special functions provided by the standard library to implement non-local return from a function – these also won’t be used in this course



Control Flow: true & false?

- Plain-vanilla C **does not** have a boolean type
 - C99 does have `<stdbool.h>`, but it does not help you understand how truth and falsity work in C.
- However, to build conditional (boolean) expressions we can use the following operators:
 - relational operators: `>`, `<`, `>=`, `<=`
 - equality operators: `==`, `!=`
 - logical operators: `&&`, `||`, `!`
- Any expression that evaluates to zero is treated as **false**, otherwise it is treated as **true**



Control Flow: beware = vs ==

- the assignment operator ("=") and equality operator ("==") have different meanings

- legal (but possibly not what you intended):

```
int a = 20;  
if (a = 5) {  
    S;  
}
```



- One extreme approach is to write conditionals like this:

- **if (5 == a) { ... }**

- Best approach overall: use compiler to catch this
 - The "-Wall" flag works well here.



if

```
/* A bexpr is a C expression which, if it is zero, is interpreted as false.  
 * Otherwise it is interpreted as true. */  
  
/* case 1: if */  
if (bexpr) {  
    S;  
}  
  
/* case 2: if-else */  
if (bexpr) {  
    S;  
} else {  
    notS;  
}  
  
/* case 3: multiway if */  
if (bexpr1) {  
    S;  
} else if (bexpr2) {  
    T;  
} else if (bexpr3) { /* Can keep on chaining more "else if" clauses */  
    U;  
} else {  
    V;  
}
```

switch

```
switch (intexpr) {  
    case int_literal_1:  
        S1;  
        break;  
  
    case int_literal_2:  
        S2;  
        break;  
  
    /* potentially many other cases */  
  
    default:  
        Sdefault;  
        break;  
}
```

- Syntax:
 - **intexpr** is an "integer expression"
 - **intlit** is an integer literal (i.e., it must be computable at compile time)
 - **if (intexpr == intlit)** execute Sn;
 - **break** continues execution after the switch-statement's closing brace



Example: char case labels

```
#include <ctype.h>
#define TRUE 1
#define FALSE 0
...
int isvowel(int ch) {
    int res;

    switch(toupper(ch)) {
        /* Note that character literals are also considered
         * integer expressions in C!
        */
        case 'A':
        case 'E':
        case 'I':
        case 'O':
        case 'U':
            res = TRUE;
            break;
        default:
            res = FALSE;
    }
    return res;
}
```

Control Flow (while)

- `while (bexpr) {
 S;
}`
- iteration, top-tested
- keywords: **continue**, **break** have significance here
 - **continue**: start the next loop iteration by checking the while conditional
 - **break** : exit the loop immediately, resume at first instruction after the while body

```
char buf[50];
int pos = 0;

if (fgets(buf, 50, stdin) == NULL) {
    /* report an error and exit */
}

while(buf[pos] != '\0') {
    if (isvowel(buf[pos])) {
        putchar(toupper(buf[pos]));
    } else {
        putchar(buf[pos]);
    }
    pos += 1;
}
```

alycd ef ghi
choose arr[51];
faets(arri, 3, inputFile)
→ abc ef gh
word1 word2 word3

Control Flow (do while)

- **do {
 S;
} while (bexpr);**
- iteration, bottom-tested
- keywords: **continue**, **break** also have significance here

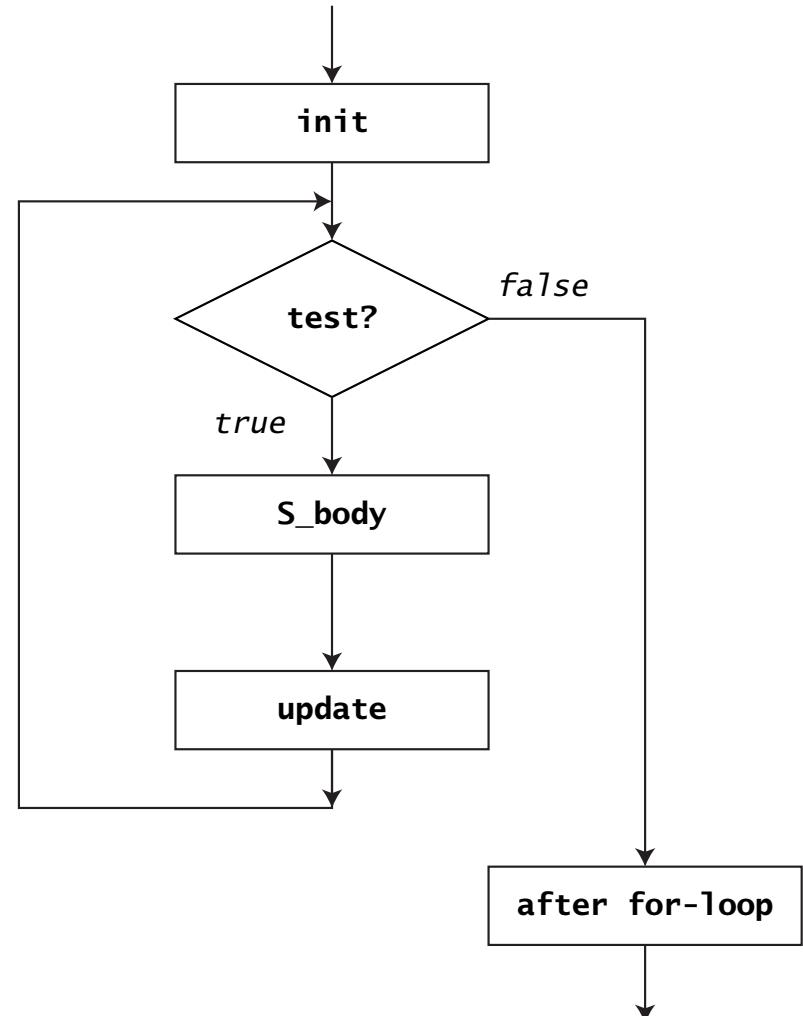
```
int ch, cnt = 0;  
  
do {  
    ch = getchar();  
    if (ch == BLANK)  
        cnt += 1;  
} while (ch != '\n');
```



Control flow (for)

```
for (init; test; update) {  
    S_body;  
}
```

1. **init** is evaluated, usually variable initialization
2. **test** is evaluated
 - a) if **test** is false, leave for-loop
 - b) if **test** is true, **S_body** is executed
 - c) after **S_body** is executed, **update** is evaluated, return to step 2
- iteration, top-tested
- keywords: **continue**, **break** have significance here



Examples

```
int i, sum;  
  
sum = 0;  
  
for (i = 1; i <= 20; i++) {  
    sum += i;  
}  
  
printf ("Sum of first 20 integers is: %d\n", sum);
```

```
int i, sum;  
  
for (i = 1, sum = 0; i <= 20; ++i, sum += i); /* Possibly works! */  
  
printf ("Sum of first 20 integers is: %d\n", sum);
```



```
while (1) {  
    /* do something */  
}
```

```
for (;;) {  
    /* do something */  
}
```

Functions

- A program is made up of one or more functions, one of which is `main()`
- Program execution always begins with `main()`
- When program control encounters a function name, the function is invoked
 - program control passes to the function
 - the function is executed
 - control is passed back to the calling function



Functions

- function syntax:

```
[<storage class>] <return type>  
    name (<parameters>) {  
        <statements>  
    }
```

- parameter syntax:

```
<type> varname , <type> varname , ...
```

- type **void**:

- if **<return type>** is **void** the function has no return value
- if **<parameters>** is **void** the function has no parameters
- e.g., **void f(void);**



Functions

- example:

```
int main(int argc, char *argv[]) {
    printf("Hello, world!\n");
    return 0;
}
```

- example:

```
double fmax(double x, double y) {
    if (x > y) {
        return x;
    } else {
        return y;
    }
}
```



Parameter passing

- C implements **call-by-value** parameter passing:

```
/* Formal parameters: m, n */

int maxint(int m, int n) {
    if (m > n) {
        return m;
    } else {
        return n;
    }
}
```

```
/* ... more code ... */

void some_function() {
    int a = 5;
    int b = 10;
    int c;

    /* Actual parameters: a, b */
    c = maxint (a, b);
    printf ("maximum of %d and %d is: %d", a, b, c);
}
```

Parameter passing

- **Call-by-value semantics** copies actual parameters into formal parameters.

```
int power2( double f ) {
    if (f > sqrt(DBL_MAX)) {
        return 0; /* Some sort of error was detected... */
    } else {
        return (int) (f * f);
    }
}
```

```
/* ... a bunch of code intervenes ... */

void some_other_function() {
    double g = 4.0;
    int h = power2(g);

    printf( "%f %d \n", g, h );
}
```



Addresses and Pointers

- Remember these points:
 - a) All variables refer to data.
 - b) All data resides in memory.
 - c) Every memory location has an address.
 - d) C exposes these details for us to use in our programs
- Some language systems hide these details from us
 - Java, C#, Python
- But others keep these details visible



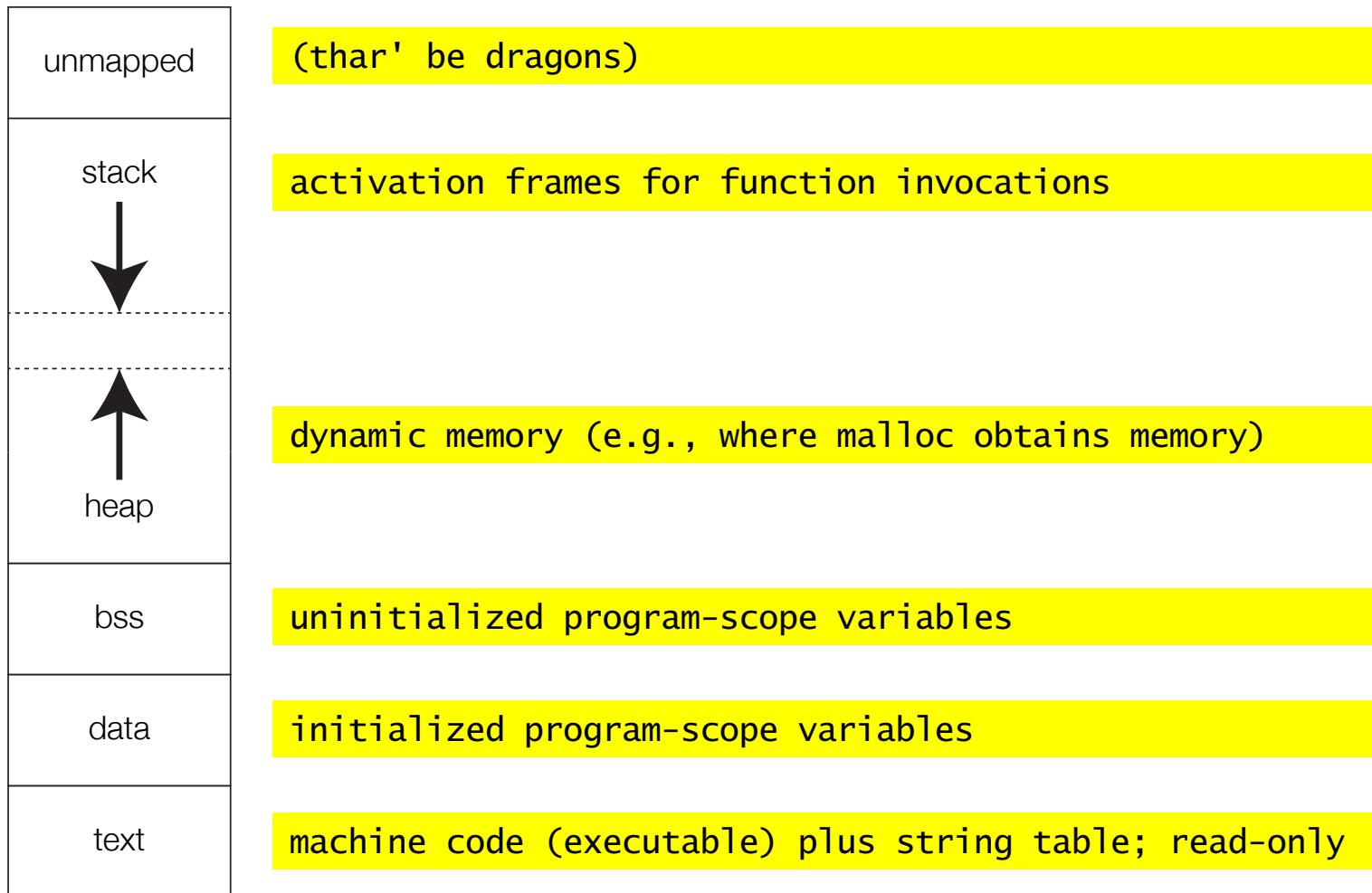
Pointer variables

- Holds the **address of a memory location** storing a value of **some data type**
 - Usually contains the address of a **named variable**
 - Sometimes an **anonymous variable** on the heap
- Often is an address **within an aggregate data type**, e.g. a location within a C string (which is stored in a character array)
- Can be used as a formal function parameter to receive the address of a variable (an ersatz "call-by-reference" mechanism)
 - Here the actual parameter (address) is copied to formal parameter (pointers)
- **To obtain an address:** use the **& symbol**
- **To use an address:** use the *** symbol** (outside of a variable declaration)



Memory model

high memory



low memory



University of Victoria
Department of Computer Science

SENG265: Software Development Methods
C Language (part 1): Slide 52

Notation (declaration)

- **Pointer variables** are declared in terms of other types (scalar and nonscalar)
- More accurate to read the simpler variable declarations **right-to-left**

```
int *a;  
double *f;
```

'a' is a variable that holds an address to an 'int'

'f' is a variable that holds an address to a 'double'



Notation (expression)

- **&x** “get the address of memory location used to store the variable x” (**referencing**)
- ***y** “read y – which contains the address to some variable – and then go to that address in order to read what is there” (**dereferencing**)
- Remember that ***** can appear in a **variable declaration**
- **However, * has a different meaning in a declaration!**

```
double f = 30.0;  
double *g = &f;  
printf("%lf %lf\n", f, *g);
```



Addresses and Pointers

- Compare the following two code fragments

```
int x = 1;
int y = x;
x = 2;
printf("y is %d\n", y); /* "y is 1" */
```

```
int x = 1;
int *y = &x;
x = 2;
printf("*y is %d\n", *y); /* "*y is 2" */
```

- In other words, **x** is a synonym for ***&x**



Notation

- A little bit trickier with arrays of pointers
`char *st[10];`
- Note: In declarations the `*` is **beside** and **logically attached** to the variable name
 - Declaration syntax is meant to remind programmer of the result of **dereferencing** the variable



Pointers

- Why do we need pointers?
- **Call-by-value** works well for passing parameters into functions, but:
 - What if we want values to be modified in the call function?
 - What if want to pass a large struct as a function argument?
- Functions can only return a single value in **return** statements.
 - What if we need multiple values changed (but don't want to write a struct for this)?
 - **Call-by-reference-like** semantics would get around the limitation of a "single return value".
 - **However, C only has call-by-value semantics!**
 - (C++ has call-by-value and call-by-reference)



Example

- swap function:

```
void swap(int a, int b) {  
    int temp = a;  
    a = b;  
    b = temp;  
}
```

```
/* ... some code here ... */  
  
void blarg() {  
    int x = 2;  
    int y = 1;  
  
    swap(x, y);  
    printf("x = %d, y = %d\n", x, y); /* x = 2, y = 1 */  
}
```



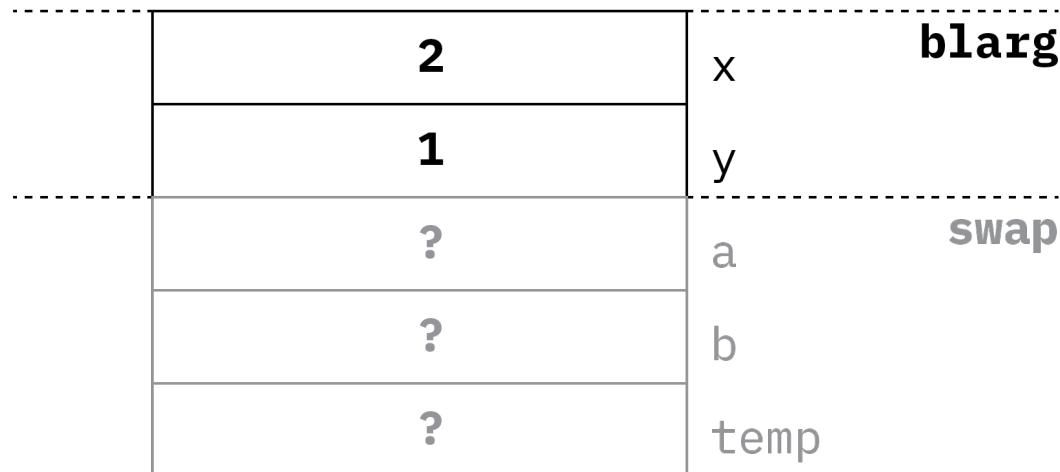
```
void swap(int a, int b) {  
    int temp = a;  
    a = b;  
    b = temp;  
}
```

```
/* ... some code here ... */  
  
void blarg() {  
    int x = 2;  
    int y = 1;  
  
    swap(x, y);  
    printf("x = %d, y = %d\n", x, y); /* x = 2, y = 1  
*/  
}
```



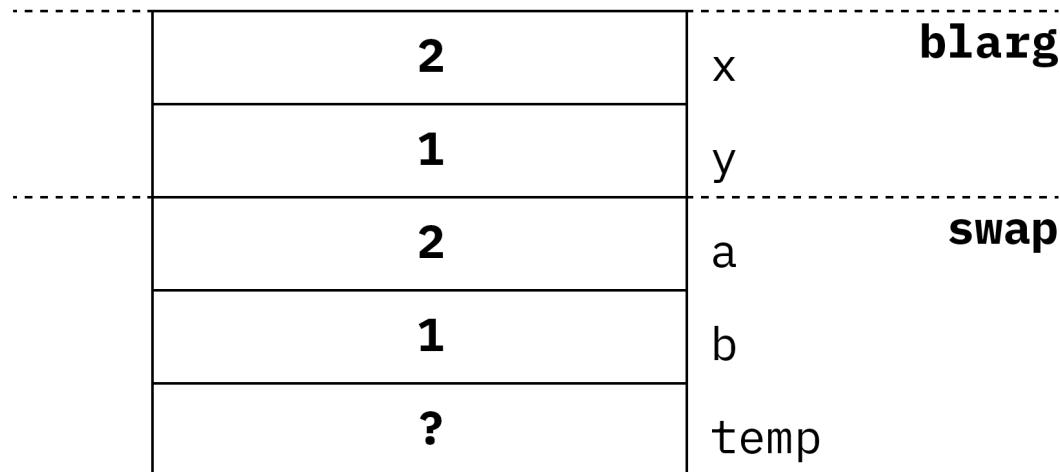
```
void swap(int a, int b) {  
    int temp = a;  
    a = b;  
    b = temp;  
}
```

```
/* ... some code here ... */  
  
void blarg() {  
    int x = 2;  
    int y = 1;  
  
    swap(x, y);  
    printf("x = %d, y = %d\n", x, y); /* x = 2, y = 1  
*/  
}
```



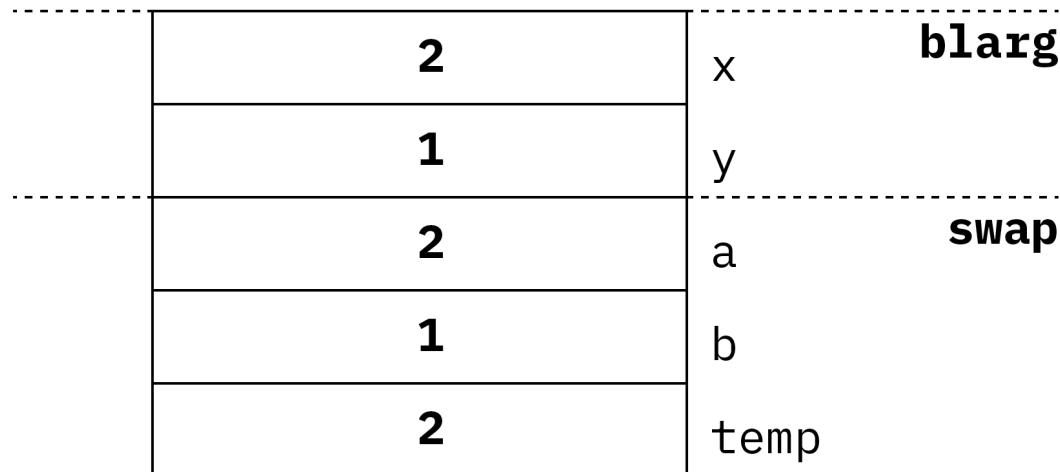
```
void swap(int a, int b) {  
    int temp = a;  
    a = b;  
    b = temp;  
}
```

```
/* ... some code here ... */  
  
void blarg() {  
    int x = 2;  
    int y = 1;  
  
    swap(x, y);  
    printf("x = %d, y = %d\n", x, y); /* x = 2, y = 1  
*/  
}
```



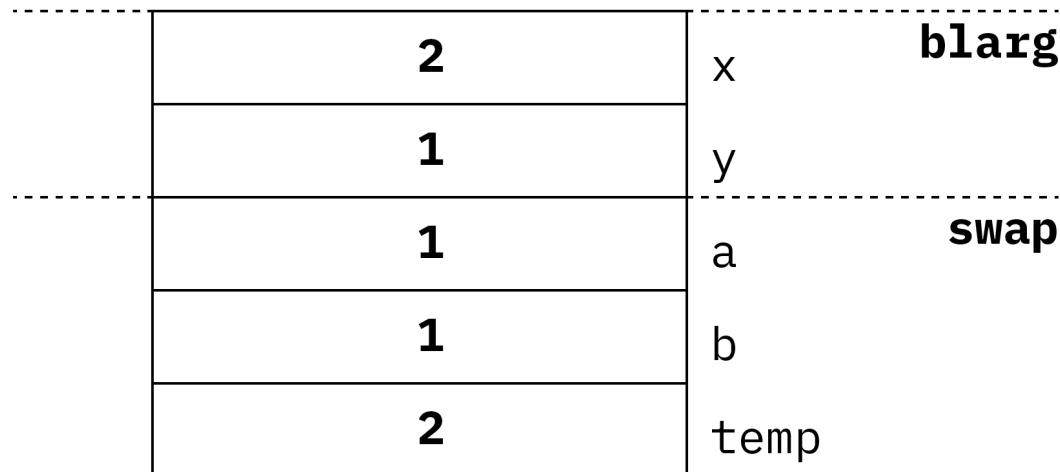
```
void swap(int a, int b) {  
    int temp = a;  
    a = b;  
    b = temp;  
}
```

```
/* ... some code here ... */  
  
void blarg() {  
    int x = 2;  
    int y = 1;  
  
    swap(x, y);  
    printf("x = %d, y = %d\n", x, y); /* x = 2, y = 1  
*/  
}
```



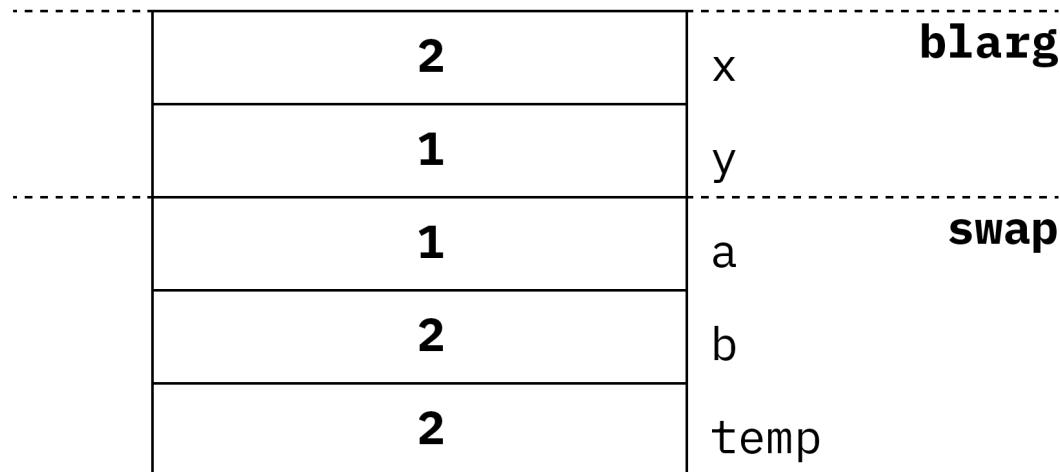
```
void swap(int a, int b) {  
    int temp = a;  
    a = b;  
    b = temp;  
}
```

```
/* ... some code here ... */  
  
void blarg() {  
    int x = 2;  
    int y = 1;  
  
    swap(x, y);  
    printf("x = %d, y = %d\n", x, y); /* x = 2, y = 1  
*/  
}
```



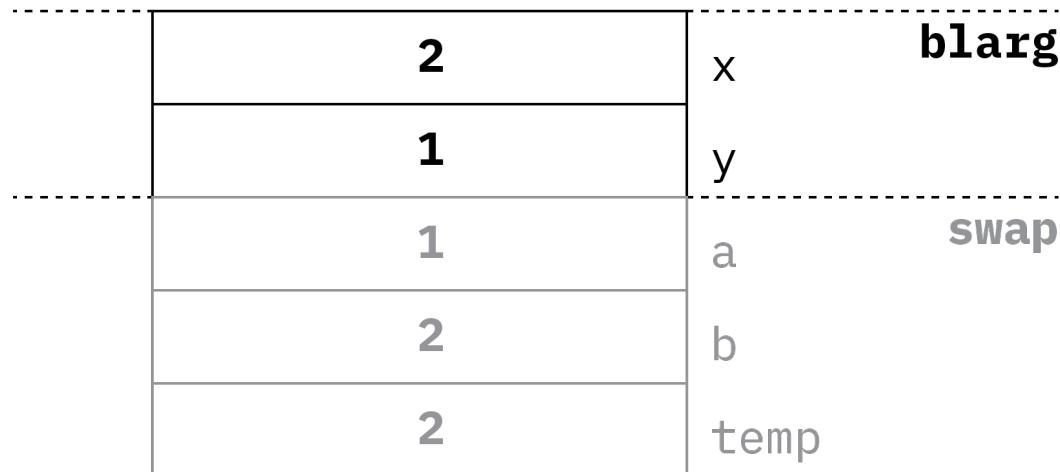
```
void swap(int a, int b) {  
    int temp = a;  
    a = b;  
    b = temp;  
}
```

```
/* ... some code here ... */  
  
void blarg() {  
    int x = 2;  
    int y = 1;  
  
    swap(x, y);  
    printf("x = %d, y = %d\n", x, y); /* x = 2, y = 1  
*/  
}
```



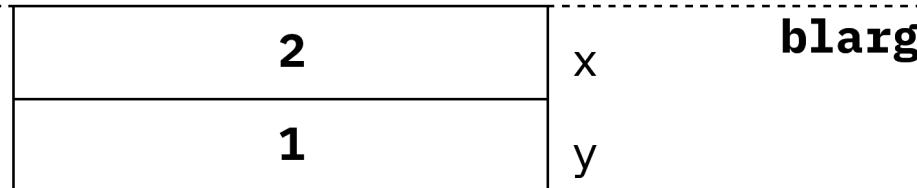
```
void swap(int a, int b) {  
    int temp = a;  
    a = b;  
    b = temp;  
}
```

```
/* ... some code here ... */  
  
void blarg() {  
    int x = 2;  
    int y = 1;  
  
    swap(x, y);  
    printf("x = %d, y = %d\n", x, y); /* x = 2, y = 1  
*/  
}
```



```
void swap(int a, int b) {  
    int temp = a;  
    a = b;  
    b = temp;  
}
```

```
/* ... some code here ... */  
  
void blarg() {  
    int x = 2;  
    int y = 1;  
  
    swap(x, y);  
    printf("x = %d, y = %d\n", x, y); /* x = 2, y = 1  
*/  
}
```



Example

- swap function:

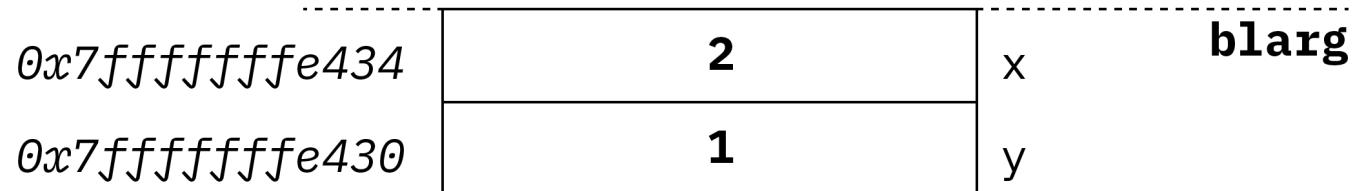
```
void swap(int *a, int *b) {  
    int temp = *a;  
    *a = *b;  
    *b = temp;  
}
```

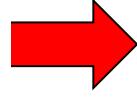
```
/* ... some code here ... */  
  
void blarg() {  
    int x = 2;  
    int y = 1;  
  
    swap(&x, &y);  
    printf("x = %d, y = %d\n", x, y);  
}
```



```
void swap(int *a, int *b) {  
    int temp = *a;  
    *a = *b;  
    *b = temp;  
}
```

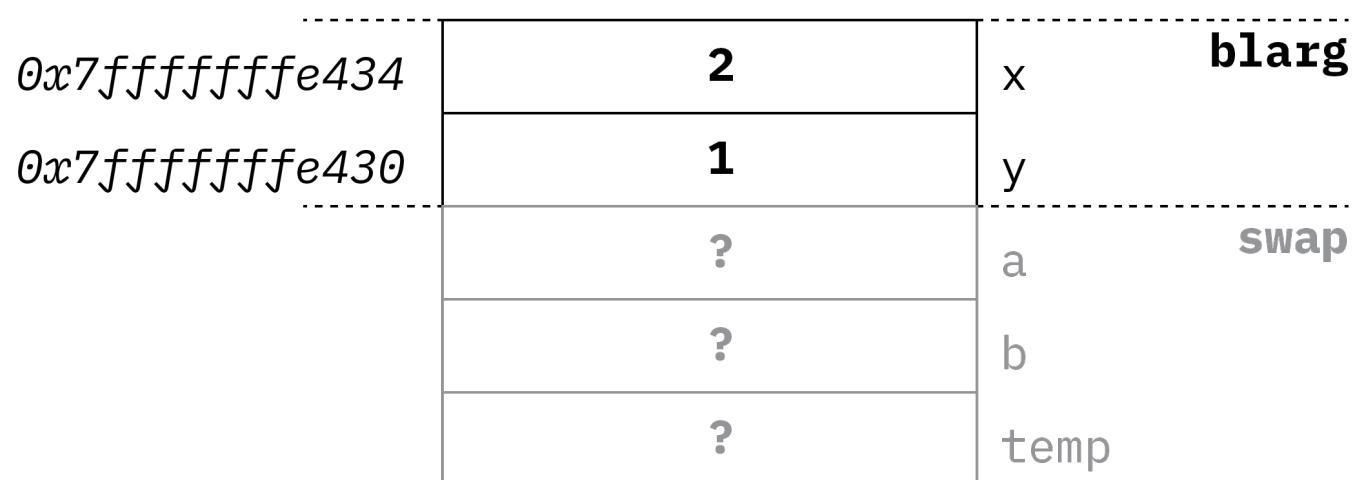
```
/* ... some code here ... */  
  
void blarg() {  
    int x = 2;  
    int y = 1;  
  
    swap(&x, &y);  
    printf("x = %d, y = %d\n", x, y);  
}
```





```
void swap(int *a, int *b) {  
    int temp = *a;  
    *a = *b;  
    *b = temp;  
}
```

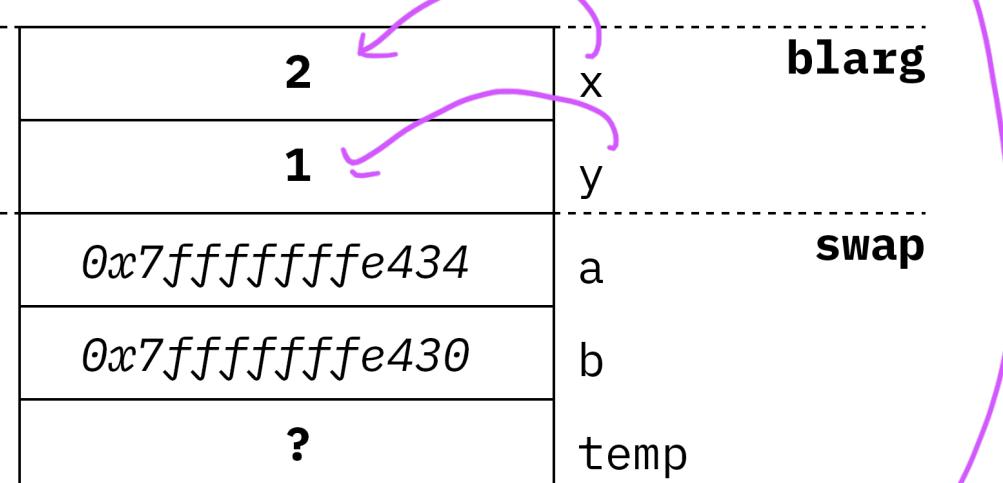
```
/* ... some code here ... */  
  
void blarg() {  
    int x = 2;  
    int y = 1;  
  
    swap(&x, &y);  
    printf("x = %d, y = %d\n", x, y);  
}
```



```
→ id swap(int *a, int *b) {  
    int temp = *a;  
    *a = *b;  
    *b = temp;  
}
```

```
/* ... some code here ... */  
  
void blarg() {  
    int x = 2;  
    int y = 1;  
  
    swap(&x, &y);  
    printf("x = %d, y = %d\n", x, y);  
}
```

This is **fantastic!** ↗
↗ memory location of ↗
↗ memory loc. of ↗



dereferenced so int?

```
void swap(int *a, int *b) {  
    int temp = *a;  
    *a = *b;  
    *b = temp;  
}
```

```
/* ... some code here ... */  
  
void blarg() {  
    int x = 2;  
    int y = 1;  
  
    swap(&x, &y);  
    printf("x = %d, y = %d\n", x, y);  
}
```

ptr b
ptr a

0x7fffffff434
0x7fffffff430

2	x	blarg
1	y	
0x7fffffff434	a	swap
0x7fffffff430	b	
2	temp	

```
void swap(int *a, int *b) {  
    int temp = *a;  
    *a = *b;  
    *b = temp;  
}
```

/* ... some code here ... */

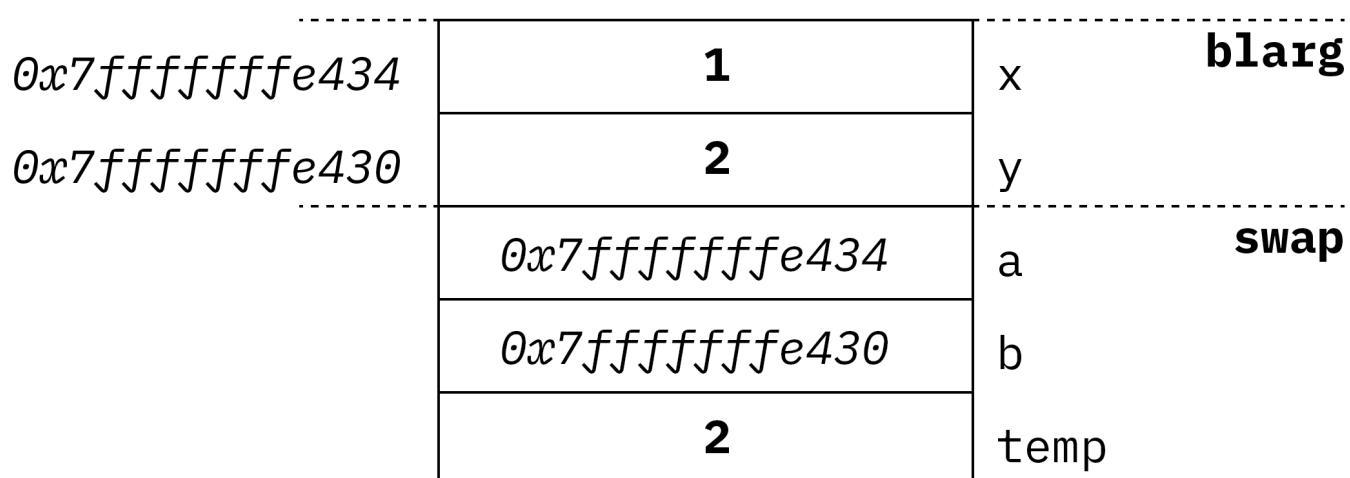
```
void blarg() {  
    int x = 2;  
    int y = 1;  
  
    swap(&x, &y);  
    printf("x = %d, y = %d\n", x, y);  
}
```

0x7fffffff434	1	x	blarg
0x7fffffff430	1	y	
	0x7fffffff434	a	swap
	0x7fffffff430	b	
	2	temp	

```
void swap(int *a, int *b) {  
    int temp = *a;  
    *a = *b;  
    *b = temp;  
}
```

/* ... some code here ... */

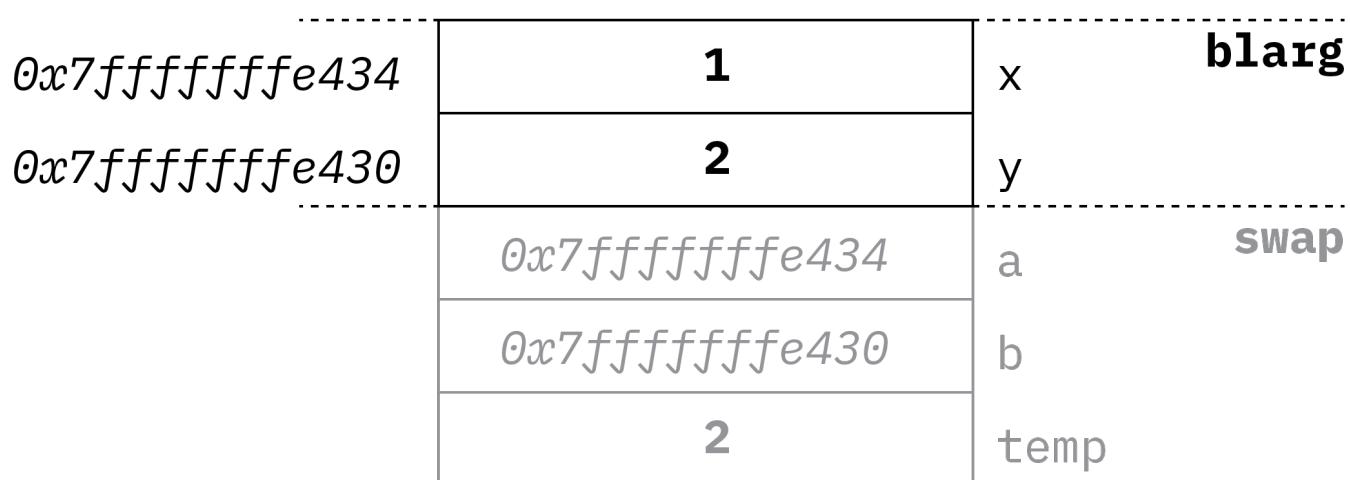
```
void blarg() {  
    int x = 2;  
    int y = 1;  
  
    swap(&x, &y);  
    printf("x = %d, y = %d\n", x, y);  
}
```



```
void swap(int *a, int *b) {  
    int temp = *a;  
    *a = *b;  
    *b = temp;  
}
```

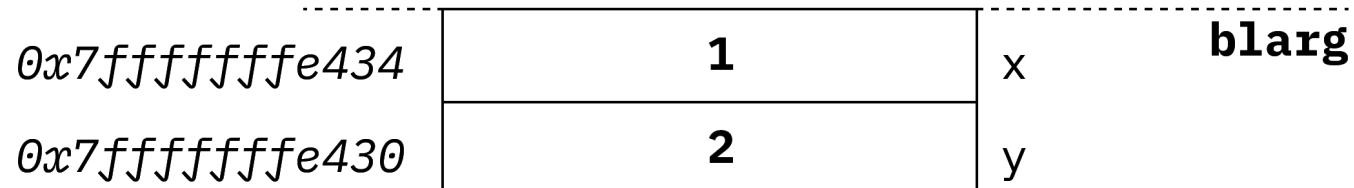
/* ... some code here ... */

```
void blarg() {  
    int x = 2;  
    int y = 1;  
  
    swap(&x, &y);  
    printf("x = %d, y = %d\n", x, y);  
}
```



```
void swap(int *a, int *b) {  
    int temp = *a;  
    *a = *b;  
    *b = temp;  
}
```

```
/* ... some code here ... */  
  
void blarg() {  
    int x = 2;  
    int y = 1;  
  
    swap(&x, &y);  
    printf("x = %d, y = %d\n", x, y);  
}
```



Invalid pointers

- C **does not implicitly check the validity of a pointer!**
 - The address could be to a region of memory holding complete and total garbage...
 - ... but C will dereference the (garbage) address if told to do so.
- It is your responsibility to ensure that a pointer contains a valid memory address
 - avoiding **dangling pointers**
 - avoid dereferencing a pointer when you're not sure of "where it has been"
- Example, what happens?
`int *x = NULL;
printf("%d\n", *x);`
– sometimes the runtime system reports use of null pointer
– **NULL** is defined in both “stdio.h” and “stdlib.h”



Pointers and arrays

- Recall that arrays are an aggregate data type where each data element has the same "type":

```
int grades[10];  
struct date_record info[50];  
char buffer[100];
```

y = memory location

- All elements in an array occupy contiguous memory locations
- To get the address of any data element, we can

~~use &:~~

~~5th element of "grades": &grades[4]~~

~~1st element of info: &info[0]~~

~~Last element of "buffer": &buffer[99]~~



Pointers and arrays

- An important array location is usually that of the first element
- In C, an array variable name without the subscript represents the first element; recall that each element is a character

```
char buffer[100];
char *cursor; → defined variable

cursor = &buffer[0]; /* these two lines ... */
cursor = buffer;    /* ... have the same effect. */
```



Pointers and arrays (3)

- Can use pointer variables and array names (sometimes) interchangeably to access array elements:

```
int X[4];
int *p = &X[0];
p = X; /* okay */
p++; /* okay */
X = p; /* illegal */
X++; /* illegal */
X[1] ~ *(p + 1);
X[n] ~ *(p + n);
```

X is an array
of a pointer to an array -

- Declarations: the following function declarations are equivalent:

```
extern double func(double X[]);
extern double func(double *X);
```

- Format #1 is often preferred as it does conveys more information



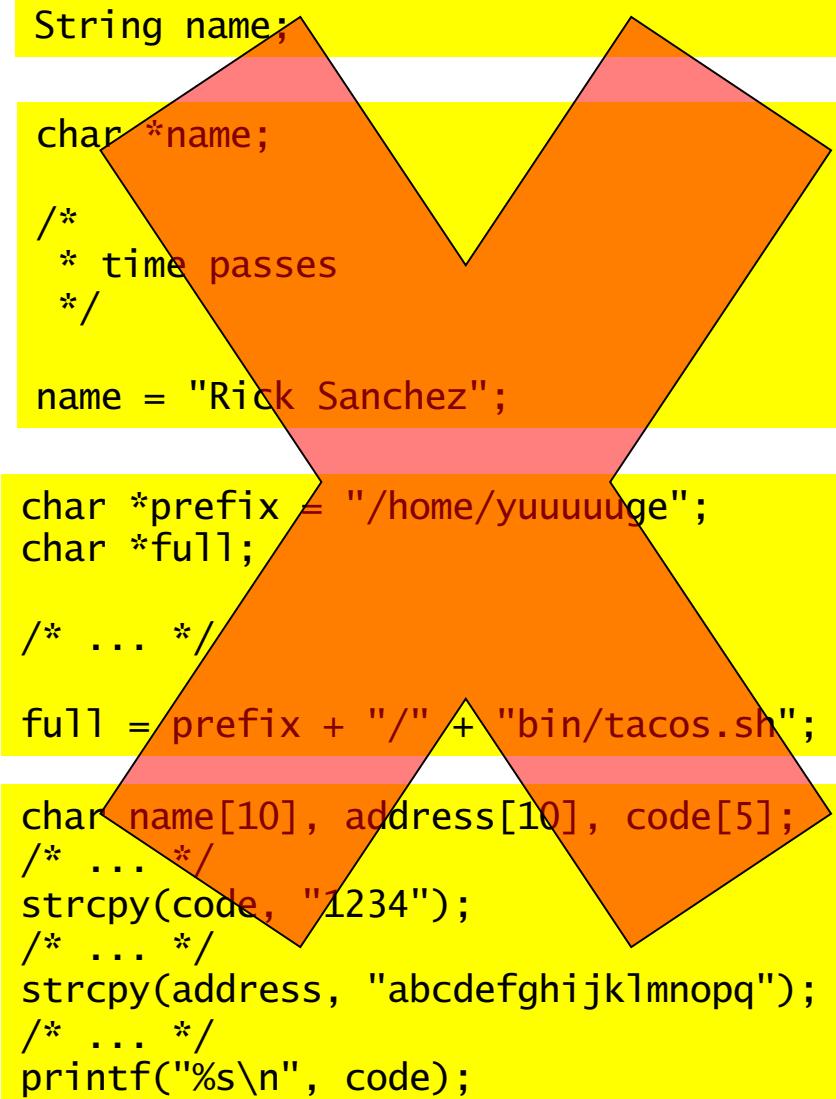
Call-by-value: caution!

- Call-by-value parameter passing semantics is straightforward to understand for:
 - scalar types (e.g., int, float, char, etc.)
 - structs
- It is a bit trickier with arrays
 - Call-by-value is still used with arrays...
 - ... but what is copied (actual parameter to formal parameter) is the **address of the array's first element!**
 - This will make more sense in 15 slides.
 - Just be aware the C **does not copy** the value each element in the array from the actual parameter to the formal parameter...
- Java implements **call-by-value** for primitive types and **call-by-sharing** for object parameters.



What is a "string"?

- "Strings" as a datatype known in Java **do not exist** in C
- Memory for strings is **not automatically allocated** on assignment.
- Concatenation via the "+" operator **is not possible**.
- The boundaries between strings **are not enforced** by the C runtime.



```
String name;  
char *name;  
/*  
 * time passes  
 */  
name = "Rick Sanchez";
```

```
char *prefix = "/home/yuuuuuge";  
char *full;  
/* ... */  
full = prefix + "/" + "bin/tacos.sh";
```

```
char name[10], address[10], code[5];  
/* ... */  
strcpy(code, "1234");  
/* ... */  
strcpy(address, "abcdefghijklmnopq");  
/* ... */  
printf("%s\n", code);
```



Strings are character arrays

- A C string is stored in a **character array**
- The start of a string is an address to a char
 - The start of the string need not be identical with the start of an array!
- The end of a string is indicated with a **null character** ('\0')
- The size of a string **need not necessarily be the same size** as the character array in which it is stored.
- C strings are often manipulated using special functions
 - `strncpy()`
 - `strcmp()`
 - `strncat()`
 - `strtok()`
- C strings are sometimes accessed **char** by **char**
- C strings are difficult to use at first
 - But you always have access to their underlying representation
- Mourn, and move on.



Example

```
char words[20];
char *pw;

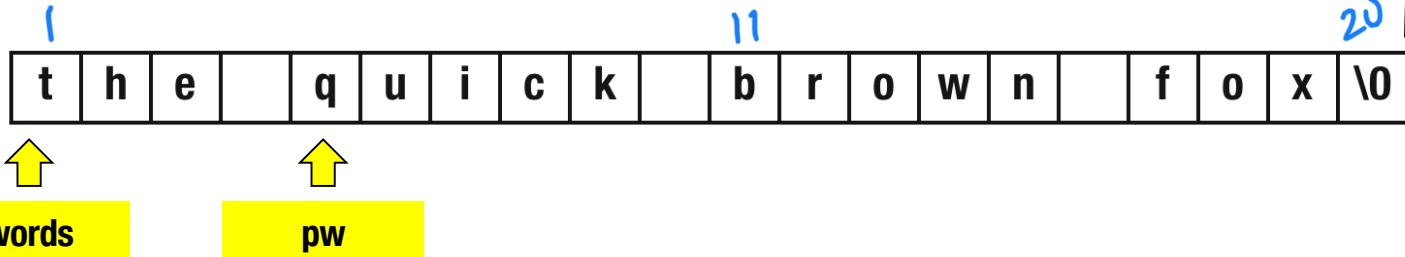
/* ... */
strncpy(words, "the quick brown fox", 20);
pw = &words[0]; /* That's the same as writing "pw = words;" */
pw += 4;

printf ("%s\n%s\n", words, pw);
printf ("%x\n%x\n", words, pw);
```

the quick brown fox
quick brown fox
~~bffff9a8~~
~~bffff9ac~~

some address-here...
some address-here...
some address-here...

null character



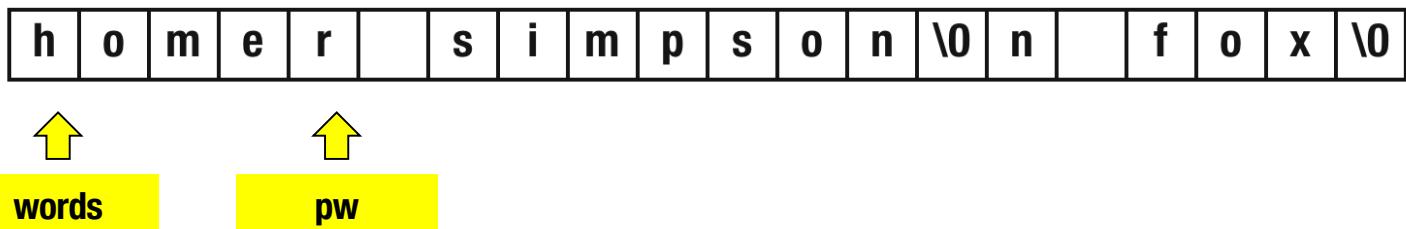
Example

```
/* ... continued from previous slide ... */

strncpy(words, "homer simpson", 20);

printf ("%s\n%s\n", words, pw);
printf ("%x\n%x\n", words, pw);
```

```
homer simpson
r simpson
bfffff9a8
bfffff9ac
```



Always be aware of array-ness!

- Always be aware that C strings are, underneath, really just C char arrays
- To store a string in your program:
 - **You must have enough room in some character array** for all the string's characters **plus** one extra character for the null
 - Therefore correct program behavior often boils down to declaring (and later in the course, allocating) char arrays which have correct sizes for your purposes
- Must be scrupulous about specifying "maximum" sizes
 - Note the third parameter of "strncpy"
- Also use "strncat" to append a string to an already existing string



Example

```
char words[20];
char first[10];
char second[10];

strncpy(first, "aaaaa", 10);
strncpy(second, "bbbb", 10);

strncpy(words, first, 20);
strncat(words, " ", 2);
strncat(words, second, 10);

printf("%s\n", words);
```

aaaaa bbbb



Example with serious problems

```
#include <stdio.h>

int main() {
    char s[50] = "abcdefghijklmnoprstuvwxyz";
    char *t = "zyxwvutsrqponmlkjihgfedcba";

    printf("message s is: '%s'\n", s);
    s[0] = ' ';
    s[1] = ' ';
    printf("modified message s is: '%s'\n", s);

    printf("message t is: '%s'\n", t); /* next two lines will fail */
    t[0] = ' ';
    t[1] = ' ';
    printf("modified message s is: '%s'\n", t);
}
```

This is okay

```
$ ./staticstring
message s is: 'abcdefghijklmnoprstuvwxyz'
modified message s is: ' cdefghijklmnoprstuvwxyz'
message t is: 'zyxwvutsrqponmlkjihgfedcba'
Bus error: 10
```



Strings

- In C, we can manipulate pointers in many ways
- This can help us when working with strings

char *cp = buffer same as **cp = &buffer[0]**

cp + n same as **&buffer[n]**

***(cp + n)** same as **buffer[n]**

cp++ same as **cp = cp + 1**

***cp++** same as ***cp, cp++**



C string functions

string.h: C string functions

- **strncpy(char *dest, const char *src, int length):**
 - copies the contents of string **src** to the array pointed to by **dest**. **src** and **dest** should not overlap.
- **strcmp(const char *s1, const char *s2, int length):**
 - compares the two strings **s1** and **s2**, returning a negative, zero, or positive integer if **s1** is lexicographically <, ==, > **s2**.
- **strlen(const char *s):**
 - compute the length of string **s** (not counting the terminal null character ('\0')).



Do not use **strcpy()**!

- **strcpy()** takes only two parameters:
 - destination char array
 - source char array
- If the string in the source array is longer than the size of the destination array:
 - then **strcpy()** will write over the end of destination array...
 - ... and this is what happens in a buffer overflow attack
- What kind of bad things can happen?
 - Overwrite data in the activation frame
 - Cause function to return to a different location
 - read: https://en.wikipedia.org/wiki/Buffer_overflow



Extracting words from an array

- Common problem to be solved:
 - An input line consists of individual words
 - Words are separated by "whitespace" (space character, tabs, etc.)
 - Want to get a list of the individual words
- This is called "tokenization"
 - From the word "token" used by compiler writers
 - Once streams of tokens are extracted from text, compiler operates on tokens and not the text
- We ourselves can use tokenize functionality available in the C runtime library.



tokenize.c: global elements

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>

/*
 * Compile-time constants
 */
#define MAX_WORD_LEN 20
#define MAX_WORDS 100
#define MAX_LINE_LEN 100
#define MAX_LINES 10

/*
 * Global variables
 */
int num_words = 0;
int num_lines = 0;
char lines[MAX_LINES][MAX_LINE_LEN];
char words[MAX_WORDS][MAX_WORD_LEN];

void dump_words (void);
void tokenize_line (char *);
```

The program will store lines of text.

It will also store words.

Size of global arrays is determined by the run-time constants.

The constants are not stored with the array!

Function prototypes...



tokenize.c: easy stuff

```
void dump_words ()  
{  
    int i = 0;  
  
    for (i=0; i<num_words; i++) {  
        printf("%5d : %s\n", i, words[i]);  
    }  
  
    return;  
}
```



tokenize.c: easy stuff

```
int main(int argc, char *argv[])
{
    int i;

    if (argc == 1) {
        exit(0);
    }

    for (i=0; i < argc-1; i++) {
        strncpy(lines[i], argv[i+1], MAX_LINE_LEN);
        tokenize_line (lines[i]);
    }

    dump_words();

    printf("first line: \"%s\"\n", lines[0]);

    exit(0);
}
```



tokenize.c: hard stuff

```
void tokenize_line (char *input_line)
{
    char *t;

    t = strtok (input_line, " ");
    while (t && num_words < MAX_WORDS) {
        strncpy (words[num_words], t, MAX_WORD_LEN);
        num_words++;
        t = strtok (NULL, " ");
    }

    /* Question: What would now be the output from
     * this statement:
     *
     * printf("%s\n", input_line);
     */
    return;
}
```

Note difference in the two calls to "strtok"

Second one uses "NULL" as the first parameter.

Why do we use a "while" to structure the loop?
Or could it be converted into a "for" loop?

Array-ness of char arrays...

```
#include <string.h>
#include <stdlib.h>
#include <stdio.h>

#define MAX_WORDS 5
#define MAX_WORD_LEN 7

char w[MAX_WORDS][MAX_WORD_LEN] ;

...
```

	0	1	2	3	4	5	6
0							
1							
2							
3							
4							



Array-ness of char arrays...

```
...
int main(int argc, char *argv[]) {
    int i;

    for (i = 0; i < MAX_WORDS; i++) {
        w[i][0] = '\0'; /* same as strncpy(w[i], "", 1); */
        /* ... but avoid w[i] = ""; */
    }
}
...
```

	0	1	2	3	4	5	6
0	/						
1	/\						
2		/\					
3			/\				
4				/\			

	0	1	2	3	4	5	6
0	/						
1	/\						
2		/\					
3			/\				
4				/\			



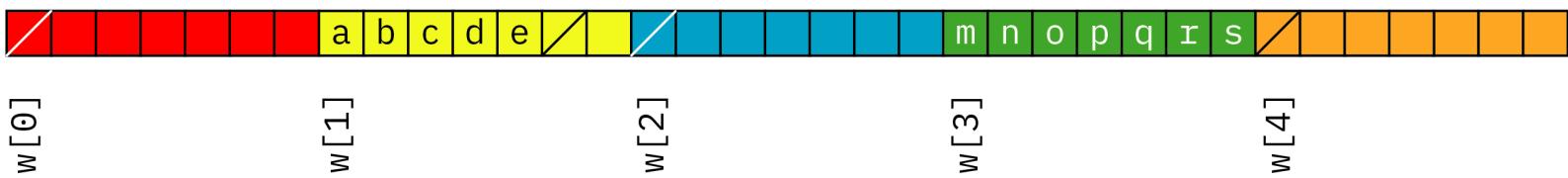
Array-ness of char arrays...

```
...
strncpy(w[1], "abcde", MAX_WORD_LEN);
strncpy(w[3], "mnopqrstuvwxyz", MAX_WORD_LEN);
...
```

	0	1	2	3	4	5	6
0	/						
1	a	b	c	d	e	/	
2	/						
3	m	n	o	p	q	r	s
4	/						

Two-dimensional arrays must be laid down onto main memory (which is just a massive 1D array of bytes).

C uses “row-major” order

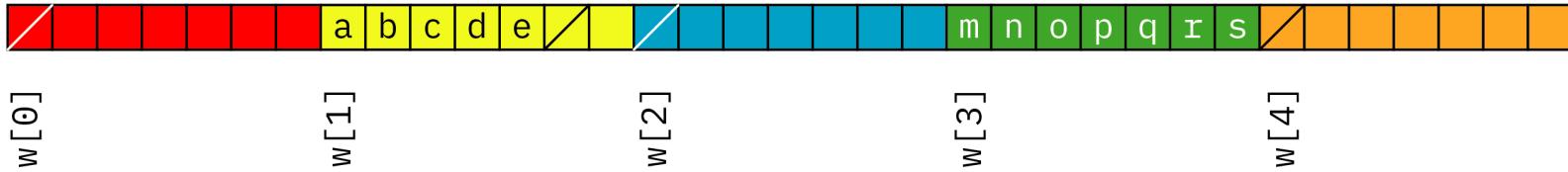


Array-ness of char arrays...

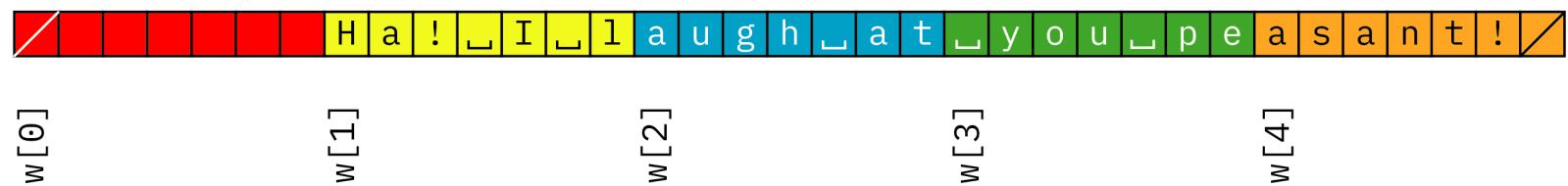
```
...  
strcpy(w[1], "Ha! I laugh at you peasant!");  
...
```

Before strcpy()

strcpy doesn't respect array dimensions.



After strcpy()



File input and output

I/O

- C, like most languages, provides facilities for reading and writing files
- files are accessed as **streams** using **FILE** objects
- the **fopen()** function is used to open a file; it returns a pointer to info about the file being opened

```
FILE *data = fopen("input.txt", "r");
```
- streams **FILE *stdin**, **FILE *stdout**, and **FILE *stderr** are automatically opened by the O/S when a program starts



File I/O

- open modes (text): "r" for reading, "w" for writing, and "a" for appending
- open modes (binary): "rb" for reading, "wb" for writing, and "ab" for appending
- the **fclose()** function is used to close a file and flush any associated buffers
- use **fgetc()** to read a single character from an open file (file was opened in "r" mode)
- similarly, **fputc()** will output a single character to the open file (file was opened in "w" mode)



File I/O

```
/* charbychar.c
 * Echo the contents of file specified as the first argument,
 * char by char. */

#include <stdio.h>
#include <stdlib.h>

int main(int argc, char *argv[]) {
    int ch, num_char;

    if (argc < 2) {
        fprintf(stderr, "You must provide a filename\n");
        exit(1);
    }

    FILE *data_fp = fopen(argv[1], "r");

    if (data_fp == NULL) {
        fprintf(stderr, "unable to open %s\n", argv[1]);
        exit(1);
    }

    num_char = 0;
    while ((ch = fgetc(data_fp)) != EOF) {
        num_char++;
        printf("%c", ch);
    }
    fclose(data_fp);

    fprintf(stdout, "Number of characters: %d\n", num_char);
    return 0;
}
```

cheekng file result
every time is important

```
/* linebyline.c
 * Echo the contents of file specified as the first argument, line by line. */

#include <stdio.h>
#include <stdlib.h>
#define BUflen 100

int main(int argc, char *argv[]) {
    char buffer[BUflen];
    int num_lines;

    if (argc < 2) {
        fprintf(stderr, "You must provide a filename\n");
        exit(1);
    }

    FILE *data_fp = fopen(argv[1], "r");

    if (data_fp == NULL) {
        fprintf(stderr, "unable to open %s\n", argv[1]);
        exit(1);
    }

    num_lines = 0;
    while (fgets(buffer, sizeof(char) * BUflen, data_fp)) {
        num_lines++;
        printf("%d: %s", num_lines, buffer);
    }
    fclose(data_fp); /* There is a little bug in the loop! */

    return 0;
}
```

I/O functions

- **FILE *fopen(char *filename, char *mode)**
 - open file corresponding to filename
 - mode can be "r" or "rw" or "rw+" (depending on flavour of Unix)
 - if an error occurs when opening file, function returns 0 (NULL)
- **char *fgets(char *buf, int n, FILE *stream)**
 - read at most **n-1** characters from **stream** and copy to location **buf**; input terminates when newline encountered or n-1 characters input. Appends a null character to end of buffer.
 - returns **NULL** if error or end-of-file encountered
 - set **stream** to **stdin** to accept input from standard input
- **int scanf(char *format, [...])**
 - read formatted data from standard input
 - returns **EOF** when end-of-file encountered, otherwise it returns the number of fields successfully converted
 - format specifiers encoded in **format** (variable # of arguments)



I/O functions

- **standard output (stdout)**
- **int printf(char *format, [...])**
 - print formatted output to standard output
 - returns the number of characters printed
 - the format specifiers are encoded in the string **format**
 - takes a variable number of arguments
- Examples:
 - `printf("My name is %s\n", name); /* char array */`
 - `printf("My name is %s and my age is %d\n", name, age); /* name is a char array, age is an int */`
 - `printf("The temperature today is %f\n", temp_celsius); /* temp_celsius is a float */`
 - `printf("%d/%d/%d", year, month, day); /* year, month and day are ints; there is no newline */`



I/O functions

- **int fprintf(FILE *stream, char *format, [...])**
 - like printf, but output goes to (already opened) stream
- **int fputc(int c, FILE *stream)**
 - outputs a single character (indicated by ASCII code in c) to (already opened) stream
 - note that the character is stored in an integer
 - idea here is the character is a number from 0 to 255
 - (if you pass a char as the first parameter, the function will still work)
- **int fclose(FILE *stream)**
 - closes the stream (i.e., flushes all OS buffers such that output to file is completed)
 - dissociates the actual file from the stream variable
 - returns 0 if file closed successfully.



Structures

- Some languages refer to these as **records**
- Aggregate data type
 - Multiple variable declarations inside a single structure
 - **Variables can be of different types**
- Structure itself becomes a **new data type**
- Example:

```
struct day_of_year {  
    int month;  
    int day;  
    int year;  
    float rating; /* 0.0: sucked; 1.0: great! */  
}; /* this new type is named "struct day_of_year" */
```

- Note: No methods or functions can be associated with such a datatype!



Structures

- structures are used to create new aggregate types
- declarations come in the following forms:
 1. `struct { int x; int y; } id;`
 - `id` is a variable (anonymous `struct`)
 2. `struct point { int x; int y; };`
 - `struct point` is a new type
 3. `struct point { int x; int y; } x, y, z[10];`
 - `struct point` is a new type; `x,y,z[]` are variables
 4. `typedef struct point { int x; int y; } Point;`
 - `struct point` is a new type, `Point` is a synonym



Structures

- To access members of a structure we employ the **member operator** (“.”) denoted by, **x.y**, and reads: “Get the value of member y from structure x”.

```
struct day_of_year today;
today.day = 45; /* not a real date! */
today.month = 10;
today.year = 2014;
today.rating = -1.0; /* bad day, off the scale */
```

- arrays of **struct** can be defined:

```
struct day_of_year calendar[365];
calendar[180].day = 27;
calendar[180].month = 9;
calendar[180].year = 2013;
calendar[180].rating = 1.0; /* Was someone's birthday */
```



Example

```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>

#define MAX_NAME_LEN 20

struct body_stats_t {
    int code;
    char name[MAX_NAME_LEN];
    float weight, height;
};

int main(void) {
    struct body_stats_t family[4];

    family[0].code = 10; family[0].weight = 220; family[0].height = 190;
    strncpy(family[0].name, "Michael", MAX_NAME_LEN-1);

    family[1].code = 21; family[1].weight = 140; family[1].height = 150;
    strncpy(family[1].name, "Susanne", MAX_NAME_LEN-1);

    printf("Name of member %d is %s\n", 0, family[0].name);
    printf("Name of member %d is %s\n", 1, family[1].name);

    exit(0);
}
```

Type definitions (`typedef`)

- C allows a programmer to create their own names for data types
 - the new name is a synonym for an already defined type
 - Syntax: **`typedef datatype synonym;`**
- examples:

```
typedef unsigned long int ulong;
typedef unsigned char byte;
ulong x, y, z[10];
byte a, b[33];
```



Example

```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>

#define MAX_NAME_LEN 20

typedef struct body_stats_t {
    int code;
    char name[MAX_NAME_LEN];
    float weight, length;
} Body_stats;

void print_stats(Body_stats p) {
    printf("Member with code %d is named %s\n", p.code, p.name);
}

int main(void) {
    Body_stats family[4];

    family[0].code = 10; family[0].weight = 220; family[0].length = 190;
    strncpy(family[0].name, "Michael", MAX_NAME_LEN-1);

    family[1].code = 21; family[1].weight = 140; family[1].length = 150;
    strncpy(family[1].name, "Susanne", MAX_NAME_LEN-1);

    print_stats(family[0]);
    print_stats(family[1]);

    exit(0);
}
```

Problem!

```
/*
 * stat_stuff.c
 */
#include <stdlib.h>
#include <stdio.h>
#include <string.h>

#define MAX_NAME_LEN 20

typedef struct body_stats_t {
    int code;
    char name[MAX_NAME_LEN];
    float weight, length;
} Body_stats;

int main(void) {
    Body_stats family[4];

    family[0].code = 10; family[0].weight = 220; family[0].length = 190;
    strncpy(family[0].name, "Michael", MAX_NAME_LEN-1);

    family[1].code = 21; family[1].weight = 140; family[1].length = 150;
    strncpy(family[1].name, "Sus Compiler will encounter a "use" of
print_stats before the function is even
is defined!

    print_stats(family[0]);
    print_stats(family[1]);

    exit(0);
}

void print_stats(Body_stats p) {
    printf("Member with code %d is named %s\n", p.code, p.name);
}
```

(Compiler output)

```
podatus:c_examples zastre$ gcc stat_stuff.c -o stat_stuff -ansi -Wall

stat_stuff.c: In function ‘main’:
stat_stuff.c:22: warning: implicit declaration of function
‘print_stats’
stat_stuff.c: At top level:
stat_stuff.c:28: warning: conflicting types for ‘print_stats’
stat_stuff.c:22: warning: previous implicit declaration of
‘print_stats’ was here
```

On the next few slides we'll learn how to fix this.



Function prototypes

- A **function declaration** provides a **prototype** for a function.
- Such a declaration includes: **optional storage class, function return type, function name, and function parameters**
- A **function definition** is the implementation of a function; includes: function declaration, and the function body. Definitions are allocated storage.
- A function's **declaration** should be “seen” by the compiler before it is used (i.e., before the function is called)
 - Why? **Type checking** (of course)!
- ANSI compliant C compilers may refuse to compile your source code if you use a function for which you have not provided a declaration. The compiler will indicate the name of the undeclared function.



Function prototypes (2)

- General syntax:
[<storage class>] <return type> name <parameters>;
- Parameters: types are necessary, but names are optional; names are recommended (improves code readability)
- A prototype looks like a function but without the function body...
- Examples:

```
int isvowel(int ch);
extern double fmax(double x, double y);
static void error_message(char *m);
```



Example (w/ prototypes)

```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>

#define MAX_NAME_LEN 20

typedef struct body_stats_t {
    int code;
    char name[MAX_NAME_LEN];
    float weight, length;
} Body_stats;

void print_stats(Body_stats);

int main(void) {
    Body_stats family[4];

    family[0].code = 10; family[0].weight = 220; family[0].length = 190;
    strncpy(family[0].name, "Michael", MAX_NAME_LEN-1);

    family[1].code = 21; family[1].weight = 140; family[1].length = 150;
    strncpy(family[1].name, "Susanne", MAX_NAME_LEN-1);

    print_stats(family[0]);
    print_stats(family[1]);

    exit(0);
}

void print_stats(Body_stats p) {
    printf("Member with code %d is named %s\n", p.code, p.name);
}
```

Prototype appears at start of C program.

Compiler reaches this point and knows what types of parameters are accepted by print_stats.

Body of print_stats seen here and compiled.

Function pointers

- In your travels you may see code that looks a bit like the following:
 - **foo = (*fp)(x, y)**
 - The function call is actually performed to whatever function is stored at the address in variable "fp"
- Strictly speaking:
 - A function is not a variable...
 - ... yet we can assign the address of functions into pointers, pass them to functions, return them from functions, etc.
- A function name used as a reference without an argument is just the function's address
- Example: qsort's use of a function pointer



Function pointers (qsort example)

0 1 2 3 4 5 6

3	14	15	9	26	58	53
---	----	----	---	----	----	----

```
#define MAX_NUMBERS 7  
  
int numbers[MAX_NUMBERS] = {3, 14, 15, 9, 26, 58, 53};
```

0xd44

0xd48

0xd4c

0xd50

0xd54

0xd58

0xd5c

	3		14		15		9		26		58		53
--	---	--	----	--	----	--	---	--	----	--	----	--	----

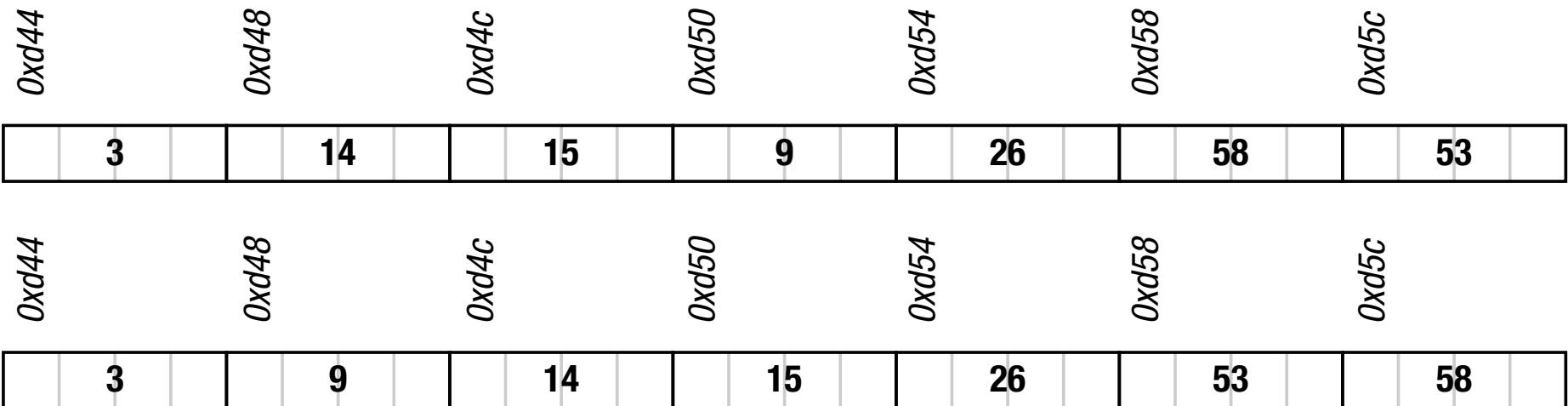


Function pointers (qsort example)

```
int main() {
    int i;
    int numbers[MAX_NUMBERS] = {3, 14, 15, 9, 26, 58, 53};

    qsort(numbers, MAX_NUMBERS, sizeof(int), compare_int);

    for (i = 0; i < MAX_NUMBERS; i++) {
        printf("%d\n", numbers[i]);
    }
}
```



Function pointers (qsort example)

```
int compare_int(const void *a, const void *b) {  
    int ia = *(int *)a;  
    int ib = *(int *)b;  
  
    return ia - ib;  
}
```

- qsort requires:
 - parameter 1: address of memory block to be sorted
 - parameter 2: number of elements in block to be sorted
 - parameter 3: size of each element
 - parameter 4: point to function that returns the sort order of two given elements in the block

