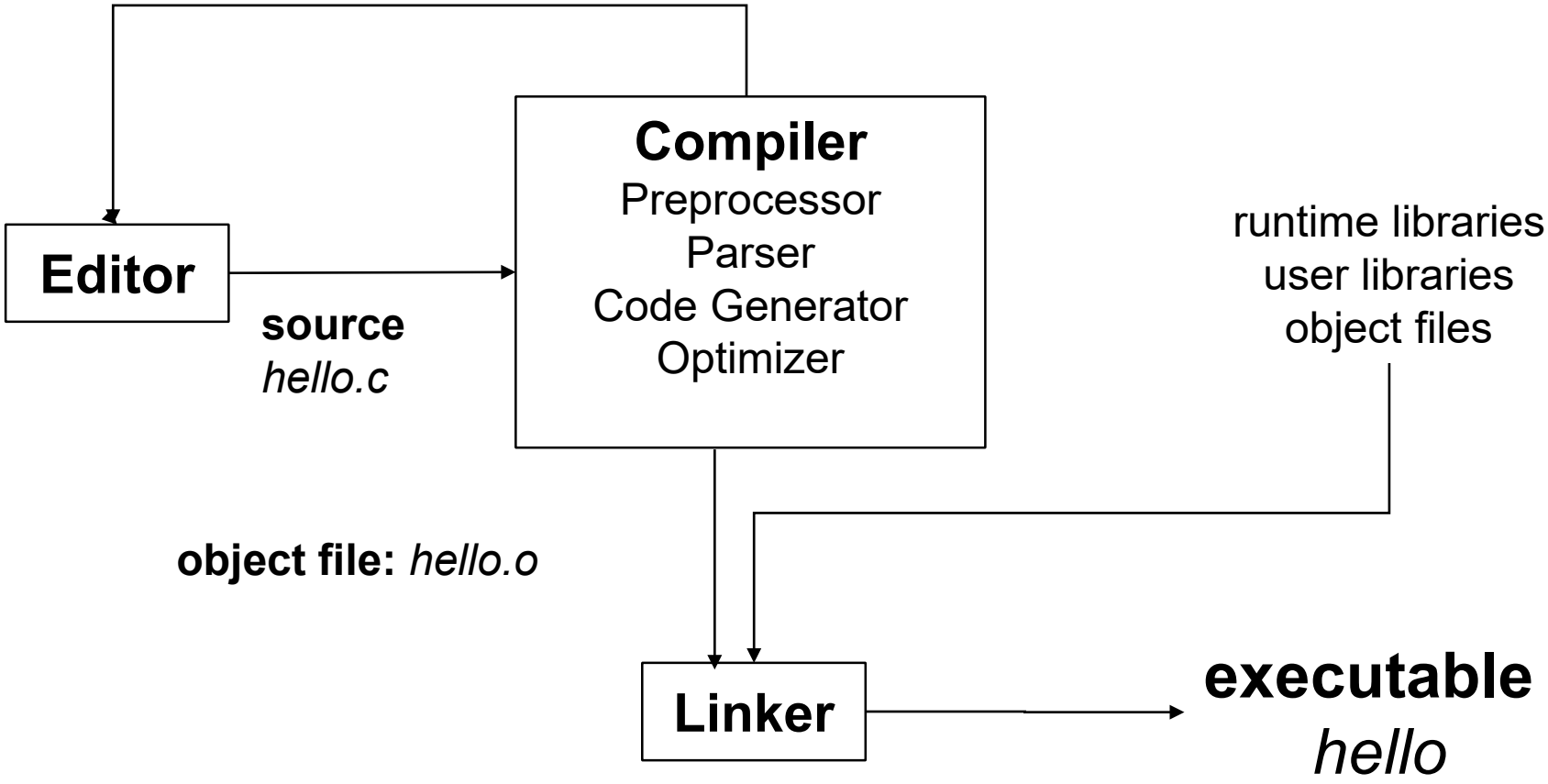
Intro to C

- Features
- How do I use C tools?
- Programming style
- C data types
- Scalar variables
- Arrays
- Control flow

```c
/*
 * 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 */
```

```
        /* 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 */

}
```

How do I use C ?


Editor
source
hello.c
Compiler
Preprocessor
Parser
Code Generator
Optimizer
runtime libraries
user libraries
object files
object file: hello.o
Linker
executable
hello

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=c11 –Wall hello.c –o hello
```

How do I use C ?

- Run the executable:

```
$ ./hello
Hello, World!
$
```

- 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

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

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 = 0xfffU;
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;
```

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
```

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 in 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.

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 */
```

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 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

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
  - C11 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 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;
}
```

Control Flow (**do while**)

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

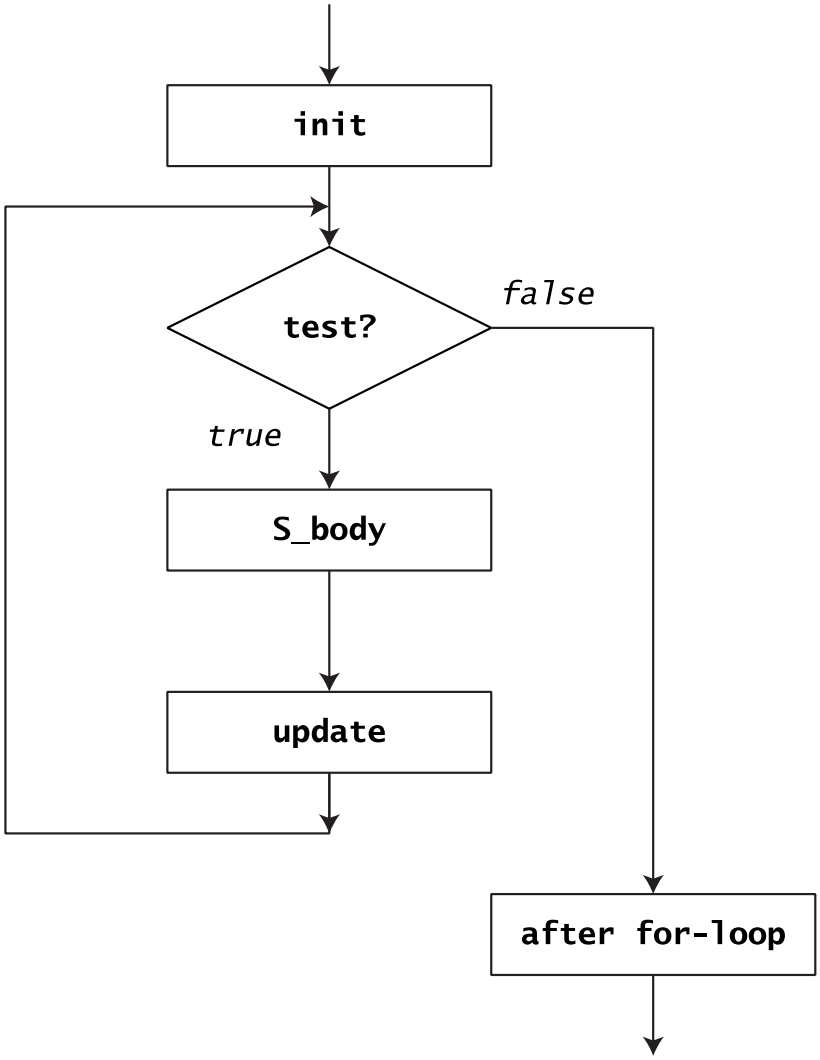
```
int ch, cnt = 0;

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 */

}
```