C, PART 1

David S. Wise
Chris Haynes

Bryce Himebaugh
Computer Structures Fall 2013

Unix is ...

- by far the most popular operating system for
  - servers
  - system development
  - portable devices
- the oldest and most influential operating system in wide use
- mostly implemented in C

C is ...

- the most common language used to interface to hardware
- the common language with the smallest (tiny) default footprint (memory usage)
- easily interfaced to assembly/machine code
- major high-level language
  - the "lowest-level" one, closest to assembly language
  - by many accounts the second most widely used (after Java), and by far the oldest and most influential, of the widely used languages

C also is ...

- statically typed
    - you have to keep its type checker happy
    - it finds many errors for you early
- not strongly typed ("insecurely typed")
    - programs can crash without a helpful message or do meaningless things if you make a mistake
    - does not have automatic memory management (no

Simple Program

Include library "header" module

Module variable

Application entry method

Library function

Exit status

```
/* demo.c (this is a comment naming the source file) */

#include <stdio.h>

int globalVal; /* another comment */

int main() {
     int localVal = 0;
     globalVal = 42;
     printf("local = %d global = %d\n",
               localVal, globalVal);
     return 0;
}
```

COMPILING/RUNNING

Compiling

```
$ gcc demo.c
```

Running

```
$ ./a.out
local = 0 global = 42
```

Alternatively

```
$ gcc -o demo demo.c
$ ./demo
local = 0 global = 42
```

Dialects of C

- **C99**: Modern **Standard C**
  - when calling `gcc` use `-std=c99` option
- **C89**: Original Standard C
  - a.k.a. ISO or ANSI C, as is C99
- **Traditional C**: Very old, c.a. 1978
  - a.k.a. K&R C
- **Clean C**: intersection of C99 and C++

The Main Confusion

- Which of the following may be used to define an application that takes no arguments and does nothing (using our gcc compiler)?

1. **int main(void) { return 0; }**

2. **main() {}**

3. **void main() {}**

4. **main(void) { return 0; }**

5. **void main() { return; }**

6. **int main(void) { return; }**

C is old and permissive

- **All** the versions of **main** in the last slide work !
  - They reflect compatibility with old versions of C, and C's extensive use of defaults and insecure typing.
- **int main(void) { return 0; }** is preferred.
  - no warnings or type insecurities in C99

C Program Organization

- Programs are broken into **modules**.
- Files are the unit of compilation -- a module is a file.
- There are no classes.
- There are no objects.
- Variables visibility (scope) is one of
  - **global** (declared at module level);
  - private to module;
  - private to procedure.

Global Variables

```
int value;             /* created in this module, */
                       /* available to others */

extern int value;      /* created in another module,*/
                       /* available here */

static int value;      /* created in this module, */
                       /* not available elsewhere */
```

If you do not follow this usage, sometimes things will work, and at others times (due to complicated rules and/or compiler differences) things will not work for mysterious reasons. So remember to use the qualifiers `extern` and `static` in this way!

Function Variables

```
void function1(void) {
    int count = 0;           /* local variable */
         /* created for each function call */
    count++; /* increment throw-away variable */
}
/* function parameters are also local variables */

void function2(void) {
    static int count = 0;   /* own variable */
         /* created once, private to function */
    count++; /* same variable in each call */
}
```

Static Variables

- **Static** variable storage is allocated when a program starts and exist for the life of the program.
- Static variables are initialized to zero by default.
- Global variables are always static (but only those that are module-local should have the static qualifier).
- Local variables may be declared static with the qualifier `static`.
- Static local variable initializers must be constant expressions and are only assigned once when the program starts.

```
void f(int m) { static int n = m; } // not allowed
void f(int m) { int n = m; }        // Ok
void f(int m) { static int n = 1; } // Ok
```

Good uses for local static variables, such as the following, are rare.

```
int new_id() { static int i = 0; return i++; }
```

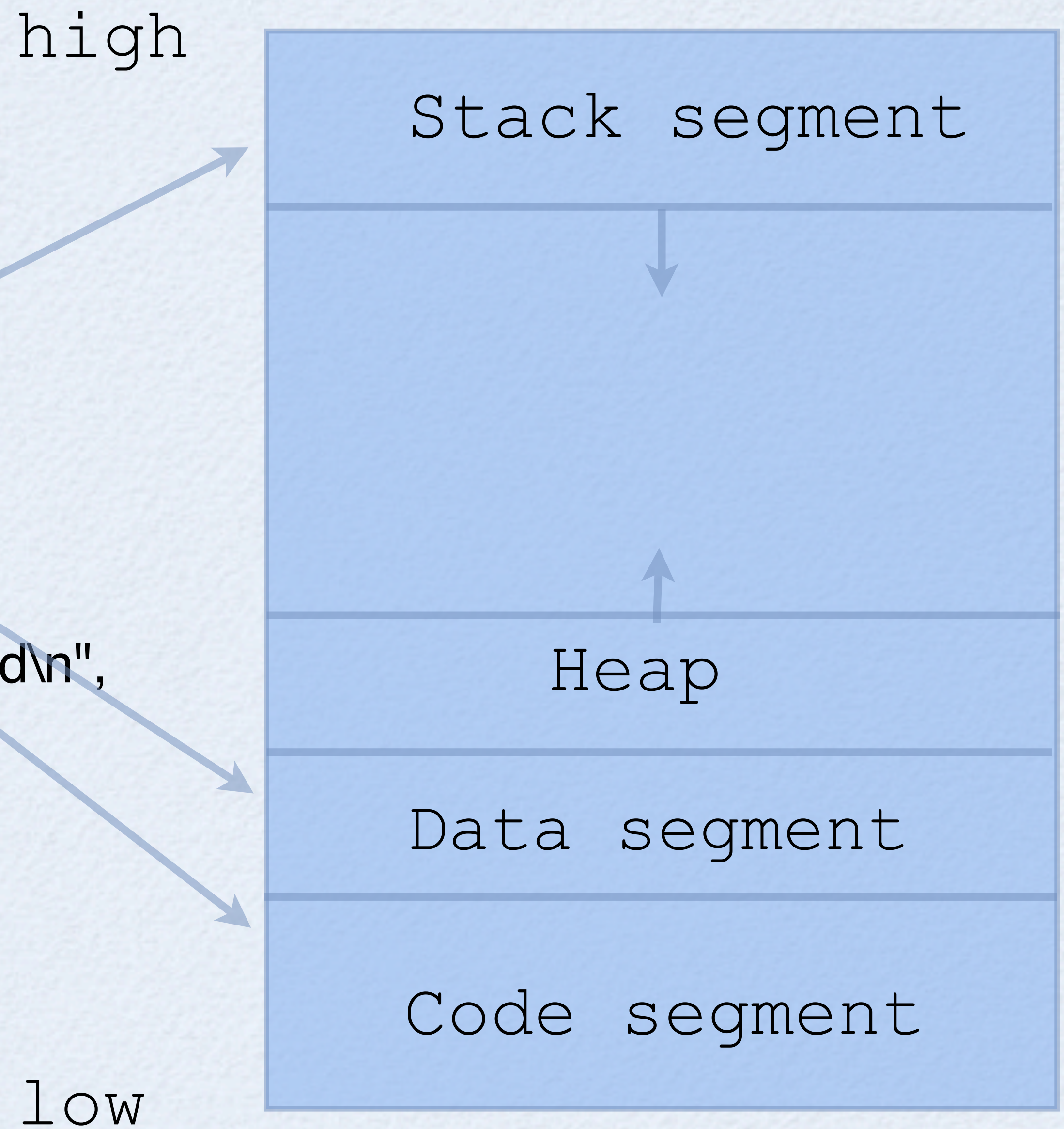
Automatic Variables

- By default local variables are **automatic**: their storage is allocated when their declaration is reached and deallocated when the function returns.
- When reallocated, they are unlikely to have the same value they had when last deallocated.
- There may be many allocations at the same time corresponding to a single automatic variable declaration (with recursion), stored at different locations on the call stack.

Program Memory Layout

```
int globalVal;

int main(void)
{
    int localVal = 0;
    globalVal = 42;
    printf("local = %d global = %d\n",
    localVal, globalVal);
    return 0;
}
```



Where Things Live

- Process (program) memory is divided into **segments** as follows:
  - **Code:** binary machine instructions (may be read-only)
  - **Data:** static variables
  - **Heap:** dynamically allocated storage (introduced later)
  - **Stack:** frames associated with function calls that has not returned yet
    - containing automatic variables and return addresses
    - and often frame links, debugger info, etc.

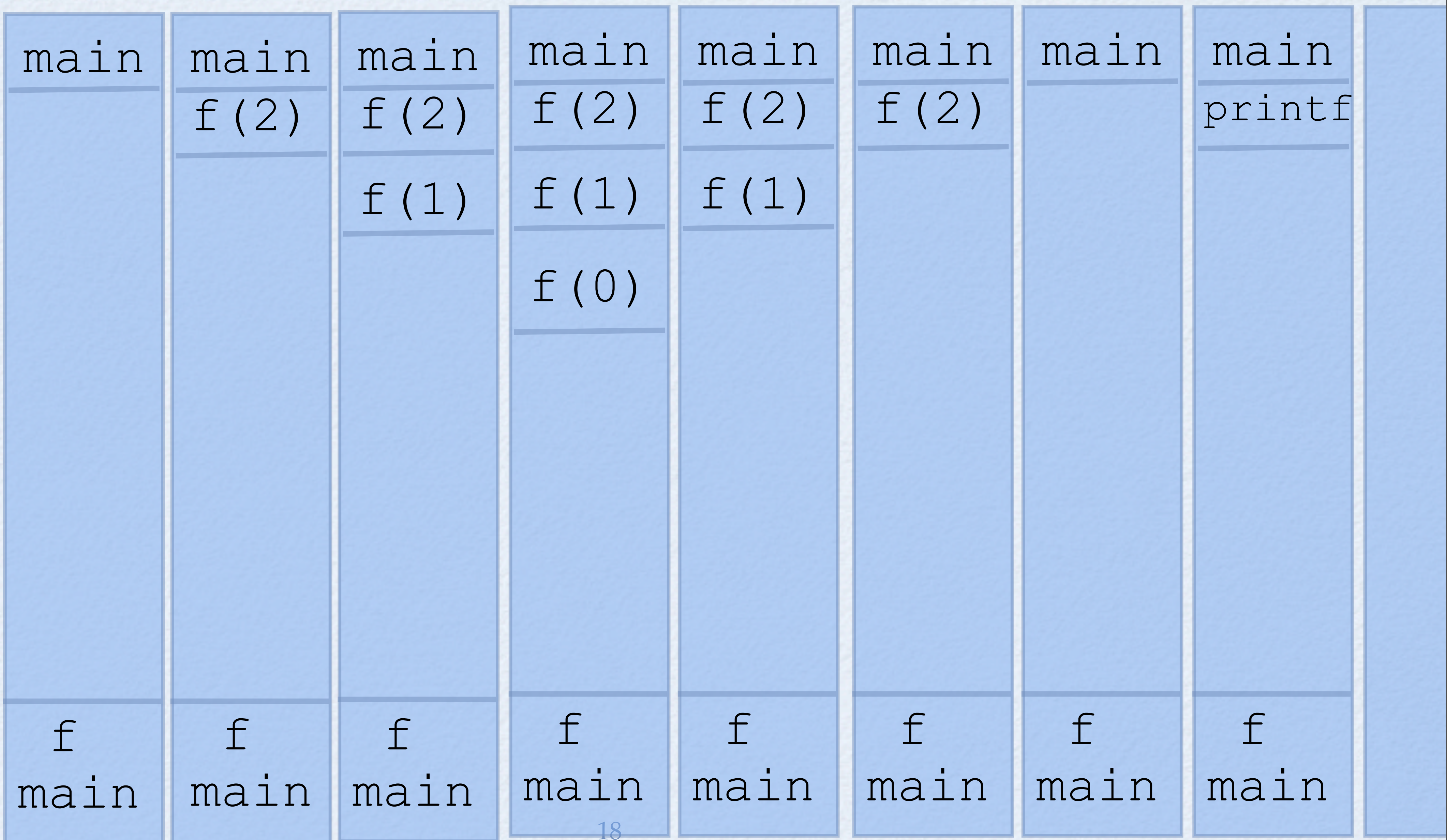
Simple Recursive Program

```
#include <stdio.h>

int f(int n)
{
  if (n == 0)
    return 1;
  else
    return n*f(n-1) ;
}

main()
{
  printf("%d", f(2));
}
```

Memory over time



Use of printf()

printf has a parameter list consisting of a control string followed by the control string arguments.

Example: the following all print ABC.

```
printf("ABC");
printf("%s", "ABC");
printf("%c%c%c", 'A', 'B', 'C');
```

It lives in the stdio.h library, which must be imported with

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

WARNING: printf is an expensive (slow and large) library function – we will write a simpler alternative.

FLOW OF CONTROL

```
if ( x == 0 ) { }

if ( y == 3) { x = 4; } else { x = 5; }

while ( i > 0) { }

for (i = 0; i < 100; i++) { };

for (int i = 0; i < 100; i++) { }; /* only in C99 */
```

Some Tricky Cases

```
int x = 100;          /* anything but 0 is true */
int y = 0;            /* false */
if (x) {...}          /* taken */
if (y) {...}          /* not taken */

if (x) {
   if (y) { g(); }
}                     /* use braces for failsafe style */
else {
   f();
}

if (y = 1) {...}      /* taken (value assigned is non-
zero) */

while (--x) {...}     /* taken 99 times */
while (x--) {...}     /* taken 100 times */
while (--y) {...}     /* ?? */
```

Arrays

```
int a1[5];                    /* 5 element array */
int a2[] = {1, 2, 3, 4};  /* 4 element array via initialization */

/* the above arrays are stored directly in variables a1 and a2 */

void foo(int a[]) { /* arrays are call-by-reference parameters */
    a[3] = 0;  /* a[] contains a pointer to the array argument */
}

int main() {
    a1[0] = 1;                /* first element of array */
    a1[5] = 3;                /* oops, index too large */
    a1[3] = a1[2];
    foo(a1);  /* passing array reference (pointer) as argument */
}   /* above the argument is the memory address of variable a1 */
```

Seeing is Understanding

- Draw a picture of program memory immediately after control enters **foo()** in the last slide's program.
  - recall there are code, data, heap (not used) and stack segments

Arrays: Similarities with Java

- Indexing syntax: same as Java
- Array initializer syntax: more flexible than Java
- Array type syntax: more limited than Java
  - types with no indication of size, such as `int a[]`, are allowed only in parameter types
  - you must use `int a[...]`, not `int[...] a`

Arrays: Not like Java

- An array's dimension is **not** stored in the array.
  - So bounds checking is not possible!
- Arrays as objects: **NOT**
  - **Arrays are (usually) not heap allocated.**
  - This has **many consequences**, some of which are explored in the following slides.

First-class values

- **First-class values** can be assigned, passed to functions, and returned from functions.
    - for example, *all values* in Java, Python, and Scheme are first-class (though some things, like Java methods, are not values)
- C arrays are **not** first-class.

Arrays are not first-class

- In C, arrays are passed to functions and assigned **by reference**.
  - a reference ("pointer") to the array (the memory address of the array) is passed or assigned
  - the array is not copied
  - as in Java, Python, etc. So far so good, but...
- An array cannot be (directly) returned from a function.
```
??? f(int a[]) { ... return a; } // not allowed
```
- An array cannot be (directly) assigned.
```
int a[10]; int b[10]; a = b;      // not allowed
```

Global Arrays

- A static (includes global) array variable declaration must indicate the array size with a constant expression, unless it has an initializer.

```
int a[10];
int b[10][5+4];
int c[] = {1, 2, 3};
```

- Their values are initialized to zero unless an initializer gives another value.

Variable Length Arrays

- An automatic variable may contain a **variable length array** whose length is computed when the variable declaration is executed runtime.

```
int a[height][width];
```

  - For some reason, if an automatic array declaration has an initializer, its dimensions must be indicated with a constant expressions.

- Automatic arrays declared without initializers, like all automatic variables, are *not* initialized: they contain garbage until assigned!

Array Insecurity

- C does **no array bounds checking:** if an array index is out of bounds (too big or less than zero) it will return garbage or, in an assignment, clobber some other part of memory!
- The result may be a segmentation fault or other kind of program abort, without any indication of what caused the problem. Even worse, your program might silently give bogus results.
- If there is any possibility that an index computation may be out of bounds, you *must* program a runtime check.
- Even if you think there is no such possibility, it is often a good idea to program bounds checks anyway. That is good **defensive programming.**

Characters and Strings

- The **`char`** (character) type is a one-byte long integer type.
- `char` literals are as in Java, e.g. `'A' and` `'\0'` (the **null** character with value 0).
- A C string is just an array of chars with the first **null** (zero: `'\0'`) character indicating the end of the string. There is no explicit string type.
- String literals are like those in Java.

Strings Example

```
#include <stdio.h>

char s1[] = {'a', 'b', 'c', '\0'};
char s2[] = "abc"; // does the same as above

void foo(char s3[]) {
    s3[0] = 'd';
}

int main(void) {
    foo(s2); // s2 is modified
    printf("%s %s\n", s1, s2); // abc dbc
}
```

Strings are arrays

```
/* copy source string to dest */
char *my_strcpy(char dest[], char source[] {
  int i = 0;
  while (source[i] != '\0') {
    dest[i] = source[i];
    i++;
  }
  dest[i] = '\0';
  return dest;
}
```

Functions Example

```
/* syntax:
     type identifier(parameter-list) { function-body }
*/
int max(int, int);   /* prototype, assigned later */

int result;
int one = 1;
int two = 2;

void example(void) { /* YUCK:  result returned via */
   result = max(5,7); /* side-effect of non-local variable */
}

int max(int one, int two) { /* GOOD: result via return value */
   return (one < two) ? two : one;
}
```

Function Prototypes

- A **function prototype** includes its name, parameter types and return type.
- Prototypes may appear at the start of a function definition, with a code body, or "stand alone", in which case the body is replaced by a semi-colon and parameter names are optional.
- An empty parameter list indicates the arguments are *unspecified!*
- Use `void` in place of the parameters to indicate no arguments.
- An ellipsis "..." in a parameter list indicates a variable number of arguments. (We won't go into this, but you should know it's possible.)

Stand-Alone Prototypes

- A call to a function that has not been defined yet is called a **forward reference** to the function.
  - Forward references can often be avoided with the right ordering of functions, but with mutually recursive functions they are unavoidable, and sometimes program organization is better if they are used.
- **Every forward reference should be preceded by a prototype for the function.**
  - If not, the compiler will often complain, but may not (there is a strange default prototype!).

RETURNING

- As with Java
  - use a return statement without an expression when the return type is **void**, or just "fall off the end" of the function
  - the type of a return expression must match the return type
- Unlike Java
  - if you "fall off the end" of a function with **int** return type, 0 is returned
  - the return type defaults to **int** (with a warning in C99)

Exercise

- Define a function **threeonly** that takes a string (assumed to contain at least three characters) and modifies the string so it contains only its first three characters.

  - If time permits, have it do nothing if the string has fewer than three characters.

  - If you have more time, rename it **truncate** and add a second argument: take a positive integer argument after the string that specifies the number of characters in the resulting string (instead of fixing that at 3).

```
char s[] = "abcde";
threeonly(s);
printf("%s\n", s); // abc
```