An intro to C language

Slides taken (and adapted) from those by

- Stephen A. Edwards
- Janae Foss and Kenneth A. Reek
- Fred Kuhns

(with many thanks)

Overview

- Currently, the most commonly-used language for embedded systems
- “High-level assembly”
- Very portable: compilers exist for virtually every processor
- Easy-to-understand compilation
- Produces efficient code
- Fairly concise

- Developed between 1969 and 1973 along with Unix
- Due mostly to Dennis Ritchie

Overview

- Traditional C
  - *The C Programming Language, by Brian Kernighan and Dennis Ritchie, 2nd Edition, Prentice Hall*
  - Standardized in 1989 by ANSI (American National Standards Institute), known as ANSI C
  - As part of the normal evolution process the standard was updated in 1995 (C95) and 1999 (C99)
- C++
  - C++ extends C to include support for Object Oriented Programming and other features that facilitate large software development projects
  - C is not strictly a subset of C++, but it is possible to write “Clean C” that conforms to both the C++ and C standards

Hello World in C

```
#include <stdio.h>

void main()
{
  printf(“Hello, world!\n”);
}
```

The preprocessor

- Commands begin with a ‘#’. Abbreviated list:
  - #define : defines a preprocessor macro/identifier/symbol
  - #undef : removes a macro definition
  - #include : insert text from file
  - #if : conditional based on value of expression
  - #ifdef : conditional based on whether macro defined
  - #ifndef : conditional based on whether macro is not defined
  - #else : alternative
  - #elif : conditional alternative
  - defined() : preprocessor function: 1 if name defined, else 0
    #if defined(__NetBSD__)

The preprocessor

- Symbolic constants

```
#define PI 3.1415926535
```

- Macros with arguments for emulating inlining

```
#define min(x,y) ((x) < (y) ? (x) : (y))
```

- Conditional compilation

```
#ifdef __STDC__
```

- File inclusion for sharing of declarations

```
#include “myheaders.h”
```

- Be careful: possible pitfalls

Source and Header files

- Just as in C++, place related code within the same module (i.e. file)
- Header files (`*.h`) export interface definitions
  - function prototypes, data types, macros, and other common declarations
- Do not place source code (i.e. definitions) in the header file with a few exceptions
  - const definitions
  - …

C Standard Header Files you may want to use

- stdio.h – file and console (also a file) IO: perror, printf, open, close, read, write, scanf, etc.
- stdlib.h - common utility functions: malloc, calloc, strtol, atoi, etc
- string.h - string and byte manipulation: strlen, strcpy, strcat, memcpy, memset, etc.
- ctype.h – character types: isalnum, isprint, isupport, tolower, etc.
- errno.h – defines errno used for reporting system errors
- math.h – math functions: ceil, exp, floor, sqrt, etc.
- signal.h – signal handling facility: raise, signal, etc
- stdint.h – standard integer: intN_t, uintN_t, etc
- time.h – time related facility: asctime, clock, time_t, etc.

Use of comments

- Only /* … */ for comments
  - no // like Java or C++
- Can't nest comments within comments
  - /* is matched with the very next */ that comes along
- Don't use /* … */ to comment out code
  - it won't work if the commented-out code contains comments

Pieces of C

- Types and Variables
  - Definitions of data in memory
- Expressions
  - Arithmetic, logical, and assignment operators in an infix notation
- Statements
  - Sequences of conditional, iteration, and branching instructions
- Functions
  - Groups of statements and variables invoked by the main program or by another function

C Types

- Basic types: char, int, float, and double
- Meant to match the processor’s native types
- Natural translation into assembly
- Fundamentally nonportable
- Declaration syntax: string of specifiers followed by a declarator

C Type Examples

`int i;`	Integer
`int *j, k;`	j: pointer to integer, int k
`unsigned char *ch;`	ch: pointer to unsigned char
`float f[10];`	Array of 10 floats
`char nextChar(int, char*);`	2-arg function
`int a[3][5][10];`	Multidimensional array
`int *func1(float);`	function returning int *

C Structures

- A struct is an object with named fields:

```
struct {
  char *name;
  int x, y;
  int h, w;
} box;
```

- Accessed using “dot” notation:

```
box.x = 5;
box.y = 2;
```

C Storage Classes

```
#include <stdlib.h>

int global_static;
static int file_static;

void foo(int auto_param)
{
  static int func_static;
  int auto_i, auto_a[10];
  double *auto_d = malloc(sizeof(double)*5);
}
```

Linker-visible. Allocated at fixed location (global_static)

Visible within file. Allocated at fixed location. (file_static)

Visible within func. Allocated at fixed location. (func_static)

C Storage Classes

```
#include <stdlib.h>

int global_static;
static int file_static;

void foo(int auto_param)
{
  static int func_static;
  int auto_i, auto_a[10];
  double *auto_d = malloc(sizeof(double)*5);
}
```

Space allocated on stack by caller. (auto_param)

Space allocated on stack by function. (auto_i, auto_a[10])

Space allocated on heap by library routine. (malloc)

malloc() and free()

- Library routines for managing the heap

```
int *a;
k=10;
a = (int *) malloc(sizeof(int) * k);
a[5] = 3;
free(a);
```

- Allocate and free arbitrary-sized chunks of memory in any order

- More flexible than automatic variables (stacked)
- Common source of errors

Arrays

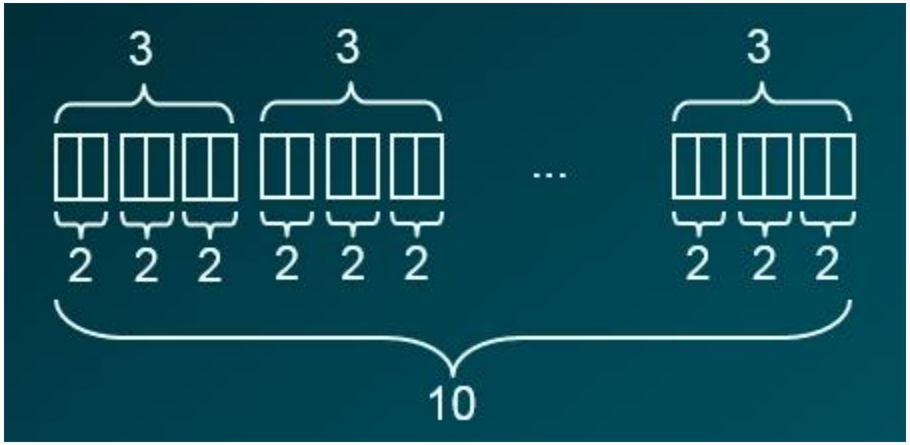
- Array: sequence of identical objects in memory
- `int a[10];` means space for ten integers
- by itself, a is the address of the first integer
- *a and a[0] mean the same thing
- The address of a is not stored in memory: the compiler inserts code to compute it when it appears

Multidimensional Arrays

- Array declarations read right-to-left:

  int a[10][3][2];

  is "an array of ten arrays of three arrays of two ints"
- in memory:



- address for an access such as `a[i][j][k]` is

`a + k + 2*(j + 3*i)`

Pointers, arrays, strings

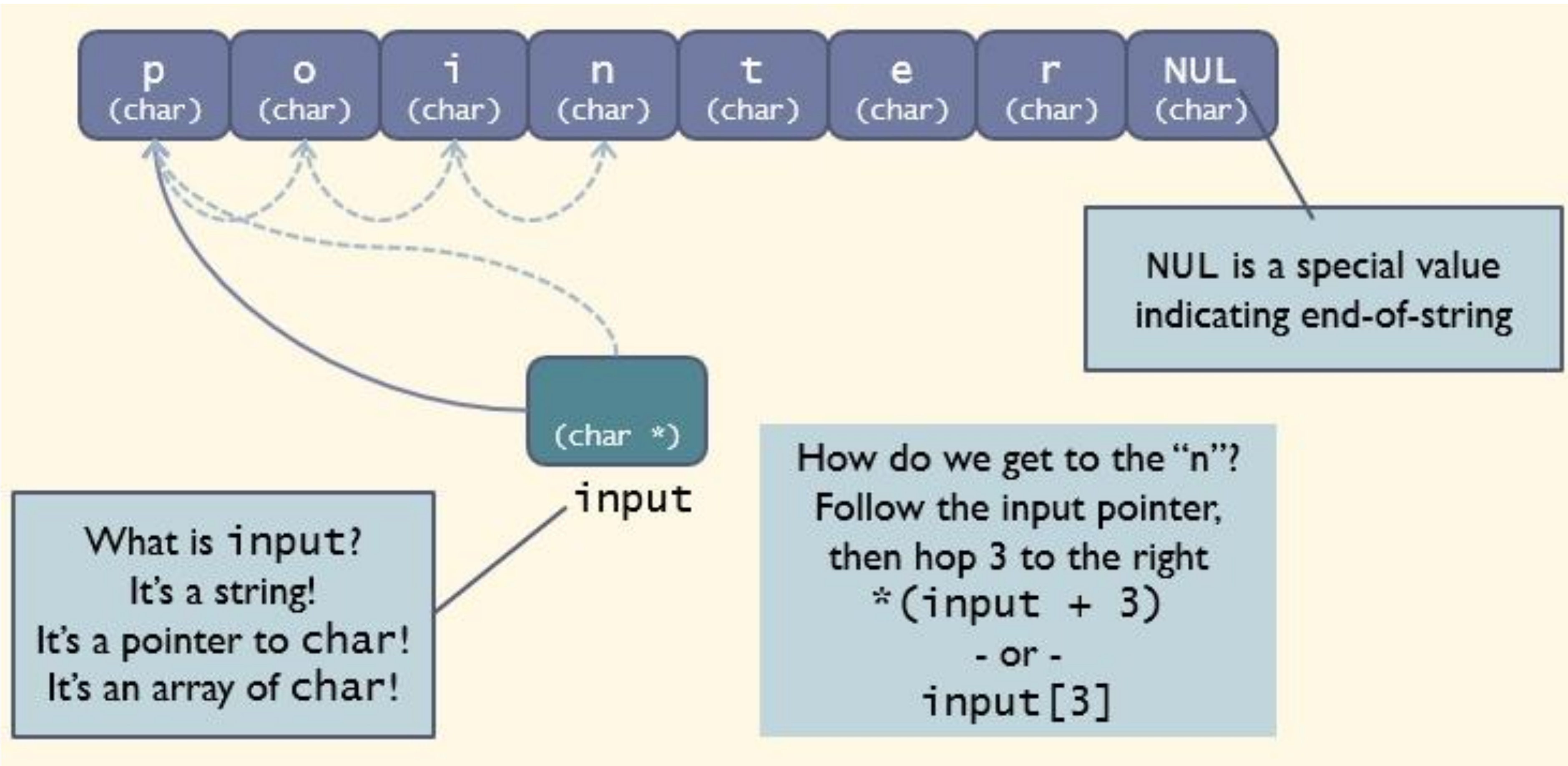
- In C, the three concepts are indeed closely related:
  - A pointer is simply a memory address. The type

```
char *
```

"pointer to character" signifies that the data at the pointer's address is to be interpreted as a character

  - An array is simply a pointer – of a special kind:
    - The array pointer is assumed to point to the first of a sequence of data items stored sequentially in memory
    - How do you get to the other array elements? By incrementing the pointer value
  - A string is simply an array of characters – unlike Java, which has a predefined String class

String layout and access



Pointer Arithmetic

- Pointer arithmetic is natural: everything is an integer

`int *p, *q;`

`*(p+5)` equivalent to `p[5]`

C Expression Classes

- arithmetic: + - * / %
- comparison: == != < <= > >=
- bitwise logical: & | ^ ~
- shifting: << >>
- lazy logical: && || !
- conditional: ? :
- assignment: = += -=
- increment/decrement: ++ --
- sequencing: ,
- pointer: * -> & []

Bitwise operators

- and: & or: | xor: ^ not: ~ left shift: << right shift: >>
- Useful for bit-field manipulations

```
#define MASK 0x040
if (a & MASK) { … }          /* Check bits */
c |= MASK;                   /* Set bits */
c &= ~MASK;                  /* Clear bits */
d = (a & MASK) >> 4;         /* Select field */
```

Lazy Logical Operators

- "Short circuit" tests save time

```
if ( a == 3 && b == 4 && c == 5 ) { … }
```

equivalent to

```
if (a == 3) { if (b ==4) { if (c == 5) { … } } }
```

- Evaluation order (left before right) provides safety

```
if ( i <= SIZEOFA && a[i] == 0 ) { … }
```

Operator Precedence

Tokens	Operator	Class	Precedence	Associates
names, *literals*	simple tokens	primary	16	n/a
a[k]	subscripting	postfix		left-to-right
f(...)	function call	postfix		left-to-right
.	direct selection	postfix		left-to-right
->	indirect selection	postfix		left to right
++ --	increment, decrement	**postfix**		left-to-right
(*type*){*init*}	compound literal	postfix		left-to-right
++ --	increment, decrement	**prefix**	15	right-to-left
sizeof	size	unary		right-to-left
~	bitwise not	unary		right-to-left
!	logical not	unary		right-to-left
- +	negation, plus	unary		right-to-left
&	address of	unary		right-to-left
*	indirection (*dereference*)	unary		right-to-left

Tokens	Operator	Class	Precedence	Associates
(*type*)	casts	unary	14	right-to-left
* / %	multiplicative	binary	13	left-to-right
+ -	additive	binary	12	left-to-right
<< >>	left, right shift	binary	11	left-to-right
< <= > >=	relational	binary	10	left-to-right
== !=	equality/ineq.	binary	9	left-to-right
&	bitwise and	binary	8	left-to-right
^	bitwise xor	binary	7	left-to-right
		bitwise or	binary	6	left-to-right
&&	logical and	binary	5	left-to-right
||	logical or	binary	4	left-to-right
?:	conditional	ternary	3	right-to-left
= += -= *= /= %= &= ^= |= <<= >>=	assignment	binary	2	right-to-left
,	sequential eval.	binary	1	left-to-right

Side-effects in expressions

- Evaluating an expression often has side-effects

a = foo()   function foo may have side-effects!!

C Statements

- Expression
- Conditional
  - if (expr) { … } else {…}
  - switch (expr) { case c1: case c2: … }
- Iteration
  - while (expr) { … } — zero or more iterations
  - do … while (expr) — at least one iteration
  - for (init ; valid ; next) { … }
- Jump
  - goto label
  - continue; — go to start of loop
  - break; — exit loop or switch
  - return expr; — return from function

Conditional Statements (if/else)

- If statement

```
if (a < 10)
  printf("a is less than 10\n");
else if (a == 10)
  printf("a is 10\n");
else
  printf("a is greater than 10\n");
```

- If you have compound statements then use brackets (blocks)

```
if (a < 4 && b > 10) {
  c = a * b; b = 0;
  printf("a = %d, a\'s address = 0x%08x\n", a, (uint32_t)&a);
  } else {
  c = a + b; b = a;
  }
```

Loops

```
for
   (i = 0; i < MAXVALUE; i++) {
      dowork();
   }
while(c != 12) {
      dowork();
   }
do {
      dowork();
   } while (c < 12);
```

- flow control
  - **break** – exit innermost loop

The Switch Statement

- Performs multi-way branches

```
switch (expr) {
case 1: …
  break;
case 6: …
  break;
default: …
  break;
}
```

Function prototypes

- Always use function prototypes

```
int myfunc (char *, int, struct MyStruct *);
int myfunc_noargs (void);
void myfunc_noreturn (int i);
```

- These look like function definitions – they have the name and all the type information – but each ends abruptly with a semicolon
  - C programs are typically arranged in “top-down” order, so functions are used (called) before they’re defined
  - When the compiler sees a call to a function, it must check whether the call is valid (the right number and types of parameters, and the right return type)
  - The prototype gives the compiler advance information about the function that is being called

Functions

- In function calls, parameters are normally passed *by value: copy of parameter passed to function*
- Arrays are *always* passed by reference in C
  - Any change made to the parameter containing the array will change the value of the original array
- To pass a “normal” parameter by reference, use the ampersand symbol
  - any changes made to the parameter also modify the original variable containing the data

```
void f(int *j) {
  (*j)++;
}

int main() {
  int i = 20;
  f(&i);
  printf("i = %d\n", i)
…
```

The printf() function

- printf("Original input : %s\n", input);
- printf() is a library function declared in <stdio.h>
- Syntax: printf(FormatString, Expr, Expr...)
- FormatString: string of text to print
- Exprs: Values to print
- FormatString has placeholders to show how to print the values (note: #placeholders should match #Exprs)
  - %s (print as string),
  - %c (print as char),
  - %d (print as integer),
  - %f (print as floating-point)
- \n indicates a newline character (don’t forget it!)

Nondeterminism in C

- Argument evaluation order: in

  myfunc(func1(), func2(), func3())

  func1, func2, and func3 may be called in any order
- Word sizes

  ```
  int a;
  a = 1 << 16;    /* might be zero */
  a = 1 << 32;    /* might be zero */
  ```
- Pointer dereference
  - *a undefined unless it points within an allocated array and has been initialized
  - very easy to violate these rules
  - legal: int a[10]; a[-1] = 3; a[10] = 2; a[11] = 5;

Nondeterminism in C

- “C treats you like a consenting adult”
  - Created by a systems programmer (Ritchie)
- “Pascal treats you like a misbehaving child”
  - Created by an educator (Wirth)
- “Ada treats you like a criminal”
  - Created by the Department of Defense

C: Dangers

- C is not object oriented!
  - can’t “hide” data as “private” or “protected” fields
  - you can follow standards to write C code that looks object-oriented, but you have to be disciplined – will the other people working on your code also be disciplined?
- C has portability issues
  - low-level “tricks” may make your C code run well on one platform – but the tricks might not work elsewhere
- The compiler and runtime system will rarely stop your C program from doing stupid/bad things
  - compile-time type checking is weak
  - no run-time checks for array bounds errors, etc. like in Java