CS370 Operating Systems

Colorado State University
Yashwant K Malaiya
Fall 2024 Lecture 2 Special
Intro to C Programming. V1.2

Slides based on

- Columbia, Cornell and other sources

C Overview

- C has been and still is widely used.
- Used for writing operation systems.
- First 3 Programming Assignments and many of the examples will be in C.
- Topics:
  - C vs Java
  - C compilation, preprocessor
  - Data types
  - Arrays and strings
  - Memory allocation/deallocation

C history

- C
  - Dennis Ritchie in late 1960s and early 1970s
  - Originally a *systems* programming language
    - make OS portable across hardware platforms
    - not necessarily for user applications – which could be written in Fortran or PL/I
- C++
  - Bjarne Stroustrup (Bell Labs), 1980s
  - object-oriented features
- Java
  - James Gosling in 1990s, originally for embedded systems
  - object-oriented, like C++
  - ideas and some syntax from C
- Python
  - created by Guido Van Rossum in the late 1980s
  - object-oriented language
  - dynamic binding and dynamic typing options

C for Java programmers

- Java is mid-90s high-level OO language
- C is early-70s *procedural* language
- C advantages:
  - Direct access to OS primitives (system calls)
  - Fewer library issues – just execute
- C disadvantages:
  - language is portable, APIs are not
  - memory and “handle” leaks
  - preprocessor can lead to obscure errors

C vs. Java

Java	**C**
object-oriented	function-oriented
strongly-typed	can be overridden
polymorphism (+, ==)	very limited (integer/float)
classes for name space	(mostly) single name space, file-oriented
macros are external, rarely used	macros common (preprocessor)
layered I/O model	byte-stream I/O

C vs. Java

Java	C
automatic memory management	function calls (C++ has some support)
no pointers	pointers (memory addresses) common
by-reference, by-value	by-value parameters
exceptions, exception handling	if (f() < 0) {error} OS signals
concurrency (threads)	library functions
length of array	on your own
string as type	just bytes (char []), with 0 end
dozens of common libraries	OS-defined

Simple C example

```
#include <stdio.h>

void main(void)
{
    printf("Hello World. \n \t and you ! \n ");
                /* print out a message */
    return;
}
```

```
$Hello World.
        and you !
$
```

- **`#include <stdio.h>`**
  - include header file `stdio.h`
  - # lines processed by *pre-processor*
  - No semicolon at end
  - Lower-case letters only – C is case-sensitive
- **`void main(void){ … }`** is the only code executed
- **`printf(" /* message you want printed */ ");`**
- **`\n`** = newline, **`\t`** = tab
- \ in front of other special characters within **`printf`**.
  - **`printf("Have you heard of \"The Rock\" ? \n");`**

The C compiler gcc

- C programs are normally compiled and linked:
  - gcc converts `foo.c` into `a.out`
  - `a.out` is executed by OS and hardware
- gcc invokes C compiler, translates C code into executable for some target
- default file name a.out

```
$ gcc hello.c
$ ./a.out        command to execute
Hello, World!
```

- Two-stage compilation
  - pre-process & compile: gcc –c hello.c
  - link: `gcc -o hello hello.o` names the runnable file hello
- Linking several modules:

`gcc -c a.c` → a.o

`gcc -c b.c` → b.o

`gcc -o hello a.o b.o`

- Using math library
  - `gcc -o calc calc.c -lm`

Numeric data types

type	bytes (typ.)	range
char	1	-128 ... 127
short	2	-65536…65535
int, long	4	-2,147,483,648 to 2,147,483,647
long long	8	2^{64}
float	4	3.4E+/-38 (7 digits)
double	8	1.7E+/-308 (15 digits)

- Range differs – `int` is "native" size, e.g., 64 bits on 64-bit machines, but sometimes `int` = 32 bits, `long` = 64 bits
- Also, `unsigned` versions of integer types
  - same bits, different interpretation
- `char` = 1 "character", but only true for ASCII and other Western char sets

Remarks on data types

- Range differs – `int` is "native" size, e.g., 64 bits on 64-bit machines, but sometimes `int` = 32 bits, `long` = 64 bits
- Also, `unsigned` versions of integer types
  - same bits, different interpretation
- `char` = 1 "character", but only true for ASCII and other Western char sets

C preprocessor

- The C preprocessor (cpp) is a macro-processor which
  - manages a collection of macro definitions
  - reads a C program and transforms it by text substitution
  - Example:

```
#define MAXVALUE 100
#define check(x) ((x) < MAXVALUE)
if (check(i) { …}
```

becomes

```
if ((i) < 100)  {…}
```

C preprocessor –file inclusion

```
#include "filename.h"
#include <filename.h>
```

- inserts contents of filename into file to be compiled
- "filename" relative to current directory
- <filename> relative to `/usr/include`
- `gcc -I` flag to re-define default
- import function prototypes (cf. Java import)
- Examples:

```
#include <stdio.h>
#include "mydefs.h"
#include "/home/alice/program/defs.h"
```


12

Example

```
#include <stdio.h>

void main(void)
{
  int nstudents = 0; /* Initialization, required */

  printf("How many students does CSU have ?:");
  scanf ("%d", &nstudents);  /* Read input */
  printf("CSU has %d students.\n", nstudents);

  return ;
}
```

$ How many students does CSU have ?: 33000 (enter)
CSU has 33000 students.

Comments

- *`/* any text until */`*
- `//` C++-style comments – careful!
- Convention for longer comments:

```
/*
 * AverageGrade()
 * Given an array of grades, compute the average.
 */
```

Colorado State University 14

Demo

```
Compiling and running a multi-file program.
```

Numeric data types

type	bytes (typ.)	range
char	1	-128 ... 127
short	2	-65536...65535
int, long	4	-2,147,483,648 to 2,147,483,647
long long	8	2^{64}
float	4	3.4E+/-38 (7 digits)
double	8	1.7E+/-308 (15 digits)

Supports only ASCII characters. Data type size may be machine dependent!

Type conversion

- Implicit: e.g., `s = a (int) + b (char)`
  - Promotion: `char` -> `short` -> `int` -> …
  - If one operand is `double`, the other is made `double`
  - If either is `float`, the other is made `float`, etc.
- Explicit: type casting – (*type*)

```
#include <stdio.h>
void main(void)
{
  int i,j = 12;        /* i not initialized, only j */
  float f1,f2 = 1.2;

  i = (int) f2;        /* explicit: i <- 1, 0.2 lost */
  f1 = i;              /* implicit: f1 <- 1.0 */

  f1 = f2 + (int) j; /* explicit: f1 <- 1.2 + 12.0 */
  f1 = f2 + j;         /* implicit: f1 <- 1.2 + 12.0 */
}
```

Explicit and implicit conversions

- Implicit: e.g., `s = a (int) + b (char)`
- Promotion: `char` -> `short` -> `int` -> …
- If one operand is `double`, the other is made `double`
- If either is `float`, the other is made `float`, etc.
- Explicit: type casting – (*type*)
- Almost any conversion does something – but not necessarily what you intended

Enumerated types

- Define new integer-like types as enumerated types:

```
typedef enum {
  Red, Orange, Yellow, Green, Blue, Violet
} Color;
enum weather {rain, snow=2, sun=4};
```

- look like C identifiers (names)
- are listed (enumerated) in definition
- treated like integers
  - can add, subtract – even `color + weather`

Data objects

- Every data object in C has
  - a name and data type (specified in definition)
  - an address (its relative location in memory)
  - a size (number of bytes of memory it occupies)
  - visibility (which parts of program can refer to it)
  - lifetime (period during which it exists)
- all C data objects have a fixed size over their lifetime
  - except dynamically created objects
- size of object is determined when object is created.
  - global data objects at compile time (data)
  - local data objects at run-time (stack)
  - dynamic data objects by programmer (heap)

Memory Usage

Global variables:

- Characteristic: declared outside any function.
- Space allocated statically before program execution.
- Initialization done before program execution if necessary also.
- Cannot deallocate space until program finishes.
- Name has to be unique for the whole program (C has flat name space).

Memory Usage

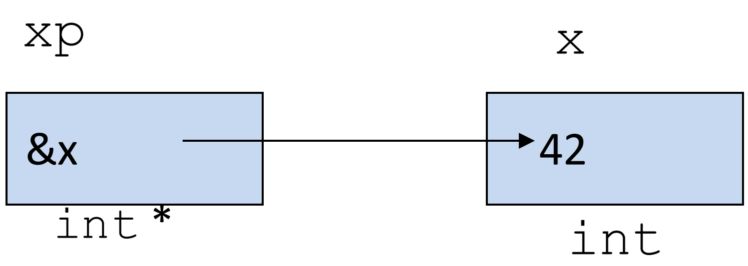
- **Local variables:**
  - Characteristic: are declared in the body of a function.
  - Space allocated when entering the function (function call).
  - Initialization before function starts executing.
  - Space automatically deallocated when function returns:
    - – **Attention:** referring to a local variable (by means of a pointer for example) after the function returned can have unexpected results.
  - Names have to be unique within the function only.

Data objects and pointers

- The memory **address** of a data object, e.g., `int x`
  - can be obtained via `&x`
  - has a data type `int *` (in general, *`type *`*)
  - has a value which is a large (4/8 byte) unsigned integer
  - can have pointers to pointers: `int **`
- The **size** of a data object, e.g., `int x`
  - can be obtained via sizeof x or sizeof(x)
  - has data type `size_t`, but is often assigned to `int` (bad!)
  - has a value which is a small(ish) integer
  - is measured in bytes

Data objects and pointers

- Every data type T in C/C++ has an associated pointer type T *
- A value of type * is the address of an object of type T
- If an object `int *xp` has value `&x`, the expression `*xp` dereferences the pointer and refers to `x`, thus has type `int`



Data objects and pointers

- If p contains the address of a data object, then *p allows you to use that object
- *p is treated just like normal data object

```
int a, b, *c, *d;
*d = 17; /* BAD idea */
a = 2; b = 3; c = &a; d = &b;
if (*c == *d) puts(“Same value”);
*c = 3;
if (*c == *d) puts(“Now same value”);
c = d;
if (c == d) puts (“Now same address”);
```

void pointers

- Generic pointer
- Unlike other pointers, can be assigned to any other pointer type:

```
void *v;
char *s = v;
```

- Acts like char * otherwise:

```
v++, sizeof(*v) = 1;
```

Arrays

- Arrays are defined by specifying an element type and number of elements
  - `int vec[100];`
  - `char str[30];`
  - `float m[10][10];`
- For array containing *N* elements, indexes are 0..*N*-1
- Stored as linear arrangement of elements
- Often similar to pointers

Arrays

- C does not remember how large arrays are (i.e., no length attribute)
- `int x[10]; x[10] = 5;` may work (for a while)
- In the block where array A is defined:
  - `sizeof A` gives the number of bytes in array
  - can compute length via `sizeof A /sizeof A[0]`
- When an array is passed as a parameter to a function
  - the size information is not available inside the function
  - array size is typically passed as an additional parameter
    - `PrintArray(A, VECSIZE);`
  - or as part of a `struct` (best, object-like)
  - or globally
    - `#define VECSIZE 10`

Arrays

- Array elements are accessed using the same syntax as in Java: array[index]
- Example (iteration over array):

```
int i, sum = 0;
...
for (i = 0; i < VECSIZE; i++)
  sum += vec[i];
```

- C does not check whether array index values are sensible (i.e., no bounds checking)
  - `vec[-1]` or `vec[10000]` will not generate a compiler warning!
  - if you're lucky, the program crashes with

```
Segmentation fault
```

Arrays

- C references arrays by the address of their first element
- `array` is equivalent to &array[0]
- can iterate through arrays using pointers as well as indexes:

```
int *v, *last;
int sum = 0;
last = &vec[VECSIZE-1];
for (v = vec; v <= last; v++)
  sum += *v;
```

Colorado State University 30

Arrays - example

```
#include <stdio.h>
void main(void) {
    int number[12]; /* 12 cells, one cell per student */
    int index, sum = 0;
             /* Always initialize array before use */
    for (index = 0; index < 12; index++) {
       number[index] = index;
    }
    /* now, number[index]=index; will cause error:why ?*/

    for (index = 0; index < 12; index = index + 1) {
       sum += number[index];  /* sum array elements */
    }
    return;
}
```

Strings

- In Java, strings are regular objects
- In C, strings are just `char` arrays with a `NUL` `('\0')` terminator
- “a cat” =

a		c	a	t	\0

- A literal string (“a cat”)
  - is automatically allocated memory space to contain it and the terminating \0
  - has a value which is the address of the first character
  - can’t be changed by the program (common bug!)
- All other strings must have space allocated to them by the program

Strings

- We normally refer to a string via a pointer to its first character:

```
char *str = “my string”;
char *s;
s = &str[0]; s = str;
```

- C functions only know string ending by \0:

```
char *str = “my string”;
...
int i;
for (i = 0; str[i] != ‘\0’; i++) putchar(str[i]);
char *s;
for (s = str; *s; s++) putchar(*s);
```

- Can treat like arrays:

```
char c;
char line[100];
for (i = 0; i < 100 && line[c]; i++) {
  if (isalpha(line[c]) ...
}
```


Colorado State University

Copying strings

- Copying content vs. copying pointer to content
- `s = t` copies pointer – `s` and `t` now refer to the same memory location
- strcpy(s, t); copies content of t to s

```
char mybuffer[100];
...
mybuffer = "a cat";
```

- is incorrect (but appears to work!)
- Use `strcpy(mybuffer, "a cat")` instead

Example string manipulation

```
#include <stdio.h>
#include <string.h>
int main(void) {
  char line[100];
  char *family, *given, *gap;
  printf("Enter your name:"); fgets(line,100,stdin);
  given = line;
  for (gap = line; *gap; gap++)
    if (isspace(*gap)) break;
  *gap = '\0';
  family = gap+1;
  printf("Your name: %s, %s\n", family, given);
  return 0;
}
```


Colorado State University

Memory Allocation and Deallocation(cont.)

Heap variables:

- Characteristic: memory has to be explicitly:
  - allocated: `void* malloc(int)` (similar to `new` in Java)
  - deallocated: `void free(void*)`
- Memory has to be explicitly deallocated otherwise all the memory in the system can be consumed (no garbage collector).
- Memory has to be deallocated exactly once, strange behavior can result otherwise.

Colorado State University

Memory Allocation and Deallocation(ex.)

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

int no_alloc_var; /* global variable counting number of

allocations */ void main(void){
    int* ptr; /* local variable of type int* */

    /* allocate space to hold an int
    */ ptr = (int*)
    malloc(sizeof(int));
    no_alloc_var++;

    /* check if successfull
    */ if (ptr == NULL)
        exit(1); /* not enough memory in the system, exiting */

    *ptr = 4; /* use the memory allocated to store value 4

    */ free(ptr); /* dealocate memory */
    no_alloc_var--;
}
```

Colorado State University

Functions

- Arguments can be passed:
  - by value: a copy of the value of the parameter handed to the function
  - by reference: a pointer to the parameter variable is handed to the function
- Returned values from functions: by value or by reference.

```
#include <stdio.h>

int sum(int a, int b); /* function declaration or prototype */
int psum(int* pa, int* pb);

void main(void){
    int total=sum(2+2,5); /* call function sum with parameters 4 and 5 */

    printf("The total is %d.\",total);
}

/* definition of function sum; has to match declaration signature */
int sum(int a, int b){ /* arguments passed by value */
    return (a+b); /* return by value */
}

int psum(int* pa, int* pb){ /* arguments passed by reference */
    return ((*a)+(*b));
}
```

Why pass by reference?

```
#include <stdio.h>

void swap(int,

int); void

main(void){
    int num1=5,
    num2=10;
    swap(num1, num2);
    printf("num1=%d and num2=%d\n", num1,
    num2);
}

void swap(int n1, int n2){ /* pass by
    value */
      int temp;
      temp = n1;
      n1 = n2;
      n2 = temp;
}
```


$ ./swaptest2
num1=10 and num2=5


```
$ ./swaptest
num1=5 and num2=10
Nothing happened!
```

```
#include <stdio.h>

void swap(int*,

int*); void

main(void){
    int num1=5,
    num2=10; int*
    ptr = &num1;
    swap(ptr,
    &num2);
    printf("num1=%d and num2=%d\n", num1,
    num2);
}

void swap(int* p1, int* p2){ /* pass by
    reference */
      int temp;temp = *p1;
      (*p1) =   *p2;
      (*p2) = temp;
}
```

```
$ ./swaptest2
num1=10 and num2=5
CORRECT NOW
```

Colorado State University

Pointer to Function

- **Goal:** have variables of type function.
- Example:

```
#include <stdio.h> void

myproc(int d){
    ...        /* do something */
}

void mycaller(void (*f)(int), int param){ f(param);
    /* call function f with param */
}

void main(void){
    myproc(10); /* call myproc */
    mycaller(myproc, 10); /* call myproc using mycaller */
}
```

Colorado State University

Demo

- Running a program with dynamic memory allocation

Things to remember

- Initialize variables before using, especially pointers.
- Make sure the life of the pointer is smaller or equal to the life of the object it points to.
  - **do not** return local variables of functions by reference
  - **do not** dereference pointers before initialization or after deallocation
- C has no exceptions so have to do explicit error handling.
- Need to do more reading on your own and try some small programs.

Appendix

- For reference purposes

Colorado State University

The stdio library

- Access stdio functions by
  - using `#include <stdio.h>` for prototypes
  - compiler links it automatically
- defines `FILE *` type and functions of that type
- data objects of type `FILE *`
  - can be connected to file system files for reading and writing
  - represent a buffered stream of chars (bytes) to be written or read
- always defines `stdin`, `stdout`, `stderr`

The stdio library: fopen(), fclose()

- Opening and closing FILE * streams:

```
FILE *fopen(const char *path, const char *mode)
```

  - open the file called path in the appropriate mode
  - modes: “r” (read), “w” (write), “a” (append), “r+” (read & write)
  - returns a new FILE * if successful, NULL otherwise

```
int fclose(FILE *stream)
```

  - close the stream FILE *
  - return 0 if successful, EOF if not

stdio – character I/O

`int getchar()`

– read the next character from `stdin`; returns EOF if none

`int fgetc(FILE *in)`

– read the next character from FILE *in*; returns EOF if none

`int putchar(int c)`

– write the character *c* onto stdout; returns *c* or EOF

`int fputc(int c, FILE *out)`

– write the character `c` onto out; returns `c` or EOF

stdio – line I/O

`char *fgets(char *buf, int size, FILE *in)`

- read the next line from `in` into buffer `buf`
- halts at `'\n'` or after size-1 characters have been read
- the '\n' is read, but not included in buf
- returns pointer to strbuf if ok, NULL otherwise
- do **not** use `gets(char *)` – buffer overflow

`int fputs(const char *str, FILE *out)`

- writes the string `str` to `out`, stopping at `'\0'`
- returns number of characters written or EOF

stdio – formatted I/O

`int fscanf(FILE *in, const char *format, ...)`

- read text from stream according to format

`int fprintf(FILE *out, const char *format, ...)`

- write the string to output file, according to format

`int printf(const char *format, ...)`

- equivalent to fprintf(stdout, format, ...)

- Warning: do not use `fscanf(...);` use `fgets(str, ...); sscanf(str, ...);`

Libraries

- C provides a set of standard libraries for

numerical math functions	<math.h>	-lm
character strings	<string.h>	
character types	<ctype.h>	
I/O	<stdio.h>	

The math library

- `#include <math.h>`
  - careful: `sqrt(5)` without header file may give wrong result!
- `gcc -o` compute `main.o f.o -lm`
- Uses normal mathematical notation:

`Math.sqrt(2)`	`sqrt(2)`
`Math.pow(x,5)`	`pow(x,5)`
`4*math.pow(x,3)`	`4*pow(x,3)`

Characters

- The char type is an 8-bit byte containing ASCII code values (e.g., ‘A’ = 65, ‘B’ = 66, ...)
- Often, char is treated like (and converted to) int
- `<ctype.h>` contains character classification functions:

`isalnum(ch)`	alphanumeric	[a-zA-Z0-9]
`isalpha (ch)`	alphabetic	[a-zA-Z]
`isdigit(ch)`	digit	[0-9]
`ispunct(ch)`	punctuation	[~!@#%^&...]
`isspace(ch)`	white space	[\t\n]
`isupper(ch)`	upper-case	[A-Z]
`islower(ch)`	lower-case	[a-z]

Pointer to function

```
int func(); /*function returning integer*/
int *func(); /*function returning pointer to integer*/
int (*func)(); /*pointer to function returning integer*/
int *(*func)(); /*pointer to func returning ptr to int*/
```

Function pointers

```
int (*fp)(void);
double* (*gp)(int);
int f(void)
double *g(int);

fp=f;
gp=g;

int i = fp();
double *g = (*gp)(17);  /* alternative */
```

Pointer to function - example

```
#include <stdio.h>

void myproc (int d);
void mycaller(void (* f)(int), int param);

void main(void) {
      myproc(10);         /* call myproc with parameter 10*/
      mycaller(myproc, 10); /* and do the same again ! */
}

void mycaller(void (* f)(int), int param){
      (*f)(param);        /* call function *f with param */
}

void myproc (int d){
      . . .               /* do something with d */
}
```