Intro to C

CS 3410

(Adapted from slides by Kevin Walsh
  (Adapted from cs414 slides by Niranjan Nagarajan
   (Adapted from slides by Alin Dobra
    (Adapted from slides by Indranil Gupta))))
Why use C instead of Java

- Intermediate-level language:
  - Low-level features like raw memory tweaking
  - High-level features like complex data-structures
- Access to all the details of the implementation
  - Explicit memory management
  - Explicit error detection
- More power than Java (so may be made faster)
- All this make C a far better choice for system programming.
Common Syntax with Java

- Basic types are similar (int, short, double...)
- Operators:
  - Arithmetic:
    - + - * / %
    - ++ -- *= += ...
  - Relational: <, >, <=, >=, ==, !=
  - Logical: &&, ||, !, ?: 
  - Bit: &, |, ^, !, <<, >>
Common Syntax with Java (cont.)

- Language constructs:
  
  ```java
  if( ) {...} else {...}
  while( ) {...}
  do {...} while( );
  for (i=0; i<100; i++) {...}
  switch( ) { case 0: ... break; ... }
  break, continue, return
  ```

- No exception handling statements

  → most functions return errors as special values (e.g., a negative number). Check for these!
Hello World Example

```c
/* Hello World program */
#include <stdio.h>
#include <stdlib.h>

int main(int ac, char **av) {
    printf("Hello World.");
}
```

```bash
$ ./hello
Hello World.
```
Primitive Types

- Integer types:
  - `char`: used to represent ASCII characters or one byte of data (not 16 bit like in Java)
  - `int`, `short` and `long`: versions of integer (architecture dependent, usually 4, 2, and 4 bytes)
  - `signed char/short/int/long`
  - `unsigned char/short/int/long`
    - `conversion between signed/unsigned often does unexpected things`
- Floating point types: `float` and `double` like in Java.
- No boolean type, `int` usually used instead.
  - `0 == false`
  - `everything else == true`
Primitive Types Examples

char c='A';
char c=65;
int i=-2343234;
unsigned int ui=100000000;

float pi=3.14;
double long_pi=0.31415e+1;
Arrays and Strings

• **Arrays:**
  
  ```
  int A[10]; // declare and allocate space for array
  for (int i=0; i<10; i++) // initialize the elements
      A[i]=0;
  ```

• **Strings: arrays of char terminated by ‘\0’ char**

  ```
  char name[] = "CS316"; // {'C','S','3','1','6','\0'}
  name[2] = '3';
  name[4]++;
  ```
  
  – Strings are mutable
  – Common functions strcpy, strcmp, strcat, strstr, strchr, strdup.
  – Use `#include <string.h>`
Pointers

- An 'address' is an index to a memory location (where some variable is stored).
- A 'pointer' is a variable containing an address to data of a certain type.

Declaring pointer variables:

```c
int i;
int* p; // p points to some random location – null pointer
```

Creating and using pointer values

```c
p = &i; // p points to integer i – p stores the address of i
(*p) = 3; // variable pointed by p takes value 3
```

- & is the address-of operator, * is the dereference operator.
- Similar to references in Java.
- Pointers are nearly identical to arrays in C
  - array variables can not be changed (only the contents can change)
int i = 6;
...
char name[] = "cs316";
...
char c = name[1];
...
short A[6];
for (i = 0; i < 6; i++)
    A[i] = i*i;
int *p;
p = &i;
short *ps;
ps = &A[1];
Pointers (cont.)

➡ **Attention:** dereferencing an uninitialized pointer can have arbitrary effects (bad!) (including program crash).

➡ Dereferencing a NULL pointer will crash the program (better!)

• Advice:
  – initialize with NULL, or some other value
  – if not sure of value, check it before dereferencing

```c
if (p == NULL) {
    printf("ack! where's my pointer!\n"); exit(1);
}
```
Structures

- Like Java classes, but only member variables
  - no static variables
  - no functions

```c
struct birthday {
    char* name;
    char month;
    short day;
    int year;
};
```

```c
struct birthday mybirthday = {"elliot",8,21,2002};
mybirthday.name[0] = 'E';
if (mybirthday.month == 6)
    printf("%s is a Cancer\n", mybirthday.name);
```
Structures (cont.)

- Members of a struct can be of any type that is already defined.
- Trick: 'struct X' can contain a pointer to 'struct X'
  ```
  struct intlist {
      int data;
      struct intlist* next;
  };
  ```
- `->` is syntax sugaring for dereference and take element:
  ```
  struct intlist one = {10, NULL};
  struct intlist two = {20, NULL};
  struct intlist *head = &one;
  one->next = &two;
  (*one).next = &two; // Does same thing as previous line
  ```
printf function

- `printf(formating_string, param1, ...)`
- Formating string: text to be displayed containing special markers where values of parameters will be filled:
  - `%d` for `int`
  - `%c` for `char`
  - `%f` for `float`
  - `%g` for `double`
  - `%s` for null-terminated strings
- Example:
  ```c
  int numstudents = 39;
  printf("The number of students in %s is %d.\n", name, numstudents);
  -> printf will not complain about wrong types, number of params, etc.
  ```
enum: enumerated data-types

enum months {
    JANUARY,
    FEBRUARY,
    MARCH,
    ...
};

• Each element of enum gets an integer value and can be used as an integer.

enum signs {
    CANCER = 6,
    ARIES = 1,
    ...
};
Memory Allocation and Deallocation

- **Global variables**: declared outside any function.
- Space allocated statically before program execution.
- Initialization statements (if any) done before main() starts.
- Space is deallocated when program finishes.
- Name has to be unique for the whole program.
Memory Allocation and Deallocation

- **Local variables:** declared in the body of a function or inside a '{ }' block.
- Space allocated when entering the function/block.
- Initialization (if any) before function/block starts.
- Space automatically deallocated when function returns or when block finishes
  - Attention: referring to a local variable (by means of a pointer for example) after the function returned will cause unexpected behavior.
- Names are visible only within the function/block
Memory Allocation and Deallocation

• **Heap variables**: memory has to be explicitly
  - allocated: `void* malloc(int)` (similar to new in Java)
    ```
    char *message = (char *)malloc(100);
    intlist *mylist = (intlist *)malloc(sizeof(intlist));
    mylist->data = 1;
    mylist->next = (intlist *)malloc(sizeof(intlist));
    mylist->next->data = 2;
    mylist->next->next = NULL;
    ```
  - deallocated: `void free(void*)`
    ```
    free(msg); msg = NULL;
    free(mylist->next);
    free(mylist);
    mylist = NULL;
    ```
Malloc/Free and pointers

→ You must malloc()
    reading/writing from random addresses is bad.

→ You must malloc() the right amount:
    reading/writing over the end of the space is bad
    `sizeof(struct birthday) 
    strlen(name)+1; /* +1 is for the '\0'

→ You must free()
    No garbage collector; if you don't have a free() for every malloc(), you will eventually run out of memory.

→ … but not too much
    Freeing same memory twice is bad ("double free").

→ … and don't use the memory after it is freed
    set pointers to NULL after free.
Memory Allocation and Deallocation

```c
struct birthday *clone_student(struct birthday *b) {
    struct birthday *b2 = (struct birthday *)malloc(sizeof(struct birthday));
    b2->name = (char *)malloc(strlen(b->name)+1); // or use strdup()
    memcpy(b2->name, b->name, strlen(b->name)+1);
    b2->day = b->day;
    b2->year = b->year;
    b2->month = b->month;
    return b2;
}

void rename(struct birthday *b, char *new_name) {
    free(b->name); // danger: b->name must be a heap variable
    b->name = strdup(new_name); // same as malloc(...) then memcpy(...)
}
```
Functions

• Can declare using a prototype, then define the body of the function later
  – lets function be used before it is defined.

• Arguments passed by value
  – Use pointers to pass by reference

• Return value passed by value
  – Use malloc()'ed pointer to return by reference
Functions - Basic Example

```c
#include <stdio.h>

int sum(int a, int b); // function declaration or prototype

int main(int ac, char **av) {
    int total = sum(2+2,5); // call function sum with parameters 4 and 5
    printf("The total is %d\n", total);
}

/* definition of sum; has to match prototype */
int sum(int a, int b) { // arguments passed by value
    return (a+b); // return by value
}
```
Why pass via pointers?

void swap(int, int);
int main(int ac, char **av) {
    int five = 5, ten = 10;
    swap(five, ten);
    printf("five = %d and ten = %d", five, ten);
}

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

$ ./swaptest
five = 5 and ten = 10
void swap(int *, int *);

int main(int ac, char **av) {
    int five = 5, ten = 10;
    swap(&five, &ten);
    printf("five = %d and ten = %d", five, ten);
}

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

$ ./swaptest
five = 10 and ten = 5