# C for Java Programmers

CS 414 / CS 415

Niranjan Nagarajan
Department of Computer Science
Cornell University
`niranjan@cs.cornell.edu`
Original Slides: Alin Dobra

# Why use C instead of Java

- Intermediate-level language:

  - Low-level features like bit operations
  - High-level features like complex data-structures

- Access to all the details of the implementation

  - Explicit memory management
  - Explicit error detection

- Better performance than Java

All this make C a far better choice for system programming.

# Goals of Tutorial

- Introduce basic C concepts:

  - need to do more reading on your own

- Warn you about common mistakes:

  - more control in the language means more room for mistakes

  - C programming requires strict discipline

- Provide additional information to get you started

  - compilation and execution

  - `printf` debugging

# Hello World Example

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

void main(void){
    printf("Hello World.\n");
}
```

```
$ ./hello
Hello World.
```

# Primitive Types

- Integer types:

  - `char` : used to represent characters or one byte data (not 16 bit like in Java)

  - `int`,`short` and `long` : versions of integer (architecture dependent)

  - can be `signed` or `unsigned`

- Floating point types: `float` and `double` like in Java.

- No `boolean` type, int or char used instead.

  - $0 \Rightarrow$ `false`

  - $\neq 0 \Rightarrow$ `true`

# Primitive Types Examples

```
char c='A';
char c=100;
int i=-2343234;
unsigned int ui=100000000;

float pi=3.14;
double long_pi=0.31415e+1;
```

# Arrays and Strings

- Arrays:

```
/* declare and allocate space for array A */
int A[10];
for (int i=0; i<10; i++)
    A[i]=0;
```

- Strings: arrays of `char` terminated by `\0`

```
char[] name="CS415";
name[4]='5';
```

  - Functions to operate on strings in `string.h`.
    * `strcpy, strcmp, strcat, strstr, strchr`.

## **printf** function

- Syntax: `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`
  - `%lf` for `double`
  - `%s` for string
- Example:

```
printf("The number of students in %s is %d.\n",
       "CS415", 80);
```

## 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 months{
    JANUARY=1,
    FEBRUARY=3,
    MARCH
};
```

# Pointers

- *address of variable*: index of memory location where variable is stored (first location).

- *pointer*: variable containing address of another variable. `type*` means pointer to variable of type `type`.

- Example:

```
int i;
int* ptr_int; /* ptr_int points to some random location */

ptr_int = &i; /* ptr_int points to integer i */
(*ptr_int) = 3; /* variable pointed by ptr_int takes value 3 */
```

- `&` address operator, `*` dereference operator.

- Similar to *references* in Java.

# Pointers (cont.)

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

- **Good programming advice:**

  - if a pointer is not initialized at declaration, initialize it with NULL, the special value for uninitialized pointer
  - before dereferencing a pointer check if value is NULL

```
int* p = NULL;
.
.
.
if (p == NULL){
    printf("Cannot dereference pointer p.\n");
    exit(1);
}
```

# Structures

- The *record* type of C, like Java classes with only members:

```
struct birthday {
    char* name;
    enum months month;
    int day;
    int year;
};

struct birthday mybirthday = {"xyz",1,1,1990};
char FirsLetter = mybirthday.name[0];
mybirthday.month = FEBRUARY;
```

# Structures (cont.)

- Structures can have as elements types already defined.

- Structures can refer to pointer to themselves:

```
struct list_elem{
    int data;
    struct list_elem* next;
};
```

- -> is syntax sugaring for dereference and take element:

```
struct list_elem le={ 10, NULL };
struct list_elem*  ptr_le = &le;
printf("The data is %d\n", ptr_le->data);
```

# Data-type Synonyms

- Syntax: `typedef type alias;`
- Example:

```
typedef int Bool;
Bool bool_var;

typedef int* Intptr;
Intptr p; /* p is a pointer to int */

typedef struct list_el list_el; /* list_el is alias for struct list_el */
struct list_el {
    int data;
    list_el* next; /* this is legal */
};
```

- *Advantage*: easier to remember, cleaner code.

# **void\*** and Type Conversion

- Type conversion syntax: `(new_type)expression_old_type`

- Examples:

```
float f=1.2;
int i = (int)f; /* i assigned value 1  */
char c=i; /* implicit conversion from int to char */
float g=i; /* implicit conversion; g=1.0 */
```

- Extremely useful conversion is to and from `void*` (pointer to unspecified type):

```
#include <string.h>
char str1[100];
char str2[100];

memcpy( (void*) str2, (void*) str1, 100);
```

- Always do explicit conversions.

# Common Syntax with Java

- Operators:
    - Arithmetic:
        * `+,-,*,/,%`
        * `++,--,*=,...`
    - Relational: `<,>,<=,>=,==,!=`
    - Logical: `&&, ||, !, ? :`
    - Bit: `&,|,^,!,<<,>>`

# Common Syntax with Java (cont.)

- Language constructs:

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

- No exception handling statements.

# Memory Allocation and Deallocation

**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 Allocation and Deallocation(cont.)

**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.

# 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.

# Memory Allocation and Deallocation(ex.)

```c
#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--;
}
```

# Functions

- Provide modularization: easier to code and debug.

- Code reuse.

- Additional power to the language: recursive 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.

# Functions – Basic Example

```
#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?

```c
#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;
}

$ ./swaptest
num1=5 and num2=10                              NOTHING HAPPENED
```

# Why pass by reference?(cont.)

```c
#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
```

# Pointer to Function

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

```c
#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 */
}
```

# The Preprocessor

- ## Module support

```
/* include standard library declaration */
#include <stdio.h>
/* include custom declarations */
#include "myheader.h"
```

- ## Symbol definition (behaves like final in Java)

```
#define DEBUG 0
#define MAX_LIST_LENGTH 100

if (DEBUG)
    printf("Max length of list is %d.\n", MAX_LIST_LENGTH);
```

- ## Conditional compilation

```
#ifdef DEBUG
    printf("DEBUG: line " _LINE_ " has been reached.\n");
#endif
```

# Programs with Multiple Files

- File `mypgm.h`:

```
void myproc(void); /* function declaration */
int mydata; /* global variable */
```

- Usually no code goes into header files, only declarations.

- File `mypgm.c`:

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

void myproc(void){
    mydata=2;
    ... /* some code */
}
```

# Programs with Multiple Files (cont.)

- File `main.c`:
  ```
  #include <stdio.h>
  #include "mypgm.h"

  void main(void){
      myproc();
  }
  ```

- Have to compile files `mpgm.c` and `main.c` to produce object files `mpgm.obj` and main.obj (`mpgm.o` and main.o on UNIX).

- Have to link files `mpgm.obj`, `main.obj` and system libraries to produce executable.

- Compilation usually automated using `nmake` on Windows and `make` on UNIX.

## 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.