CSC 533: Organization of Programming Languages

Spring 2008

Imperative programming in C

- language history
- program structure
  - functions & parameters
  - input/output
  - control structures
  - arrays
  - dynamic memory
  - bitwise operators, libraries
  - file processing
- top-down design (iterative refinement)

early history

C was developed by Dennis Ritchie at Bell Labs in 1972

- designed as an in-house language for implementing UNIX
  - UNIX was originally implemented by Ken Thompson in PDP-7 assembly
  - when porting to PDP-11, Thompson decided to use a high-level language
  - considered the B language, a stripped-down version of BCPL, but it was untyped & didn't provide enough low-level machine access
  - decided to design a new language that provided high-level abstraction + low-level access
- the UNIX kernel was rewritten in C in 1973

became popular for systems-oriented applications, and also general problem solving (especially under UNIX)

- for many years, there was no official standard (other than Kernighan & Ritchie's 1978 book)
- finally standardized in 1989 (ANSI C or C89) and again in 1999 (C99)

Program structure

a C program is a collection of functions

- libraries of useful functions can be placed in files and loaded using `#include`
- here, the `stdio.h` library contains functions for standard input/output
- including `printf` for formatted output
- to be executable, a program must have a `main` method
- comments are written using `//` or `/* ... */`

```
// hello.c
////////////////////////////

#include <stdio.h>

int main() {
    printf("Hello world!\n");

    return 0;
}
```

3

Functions

one function can call another, but must call upwards

- alternatively, can place function prototypes at the top to prepare the compiler

to integrate variables in `printf` text, use format strings

- `%s` for strings, `%d` for ints, `%f` for floats,

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

void oldMacVerse(char*, char*);

int main() {
    oldMacVerse("cow", "moo");
    return 0;
}

void oldMacVerse(char* animal, char* sound) {
    printf("Old MacDonald had a farm, E-I-E-I-O.\n");
    printf("And on that farm he had a %s, E-I-E-I-O.\n", animal);
    printf("With a %s-%s here, and a %s-%s there,\n",
           sound, sound, sound, sound);
    printf("  here a %s, there a %s, everywhere a %s-%s.\n",
           sound, sound, sound, sound);
    printf("Old MacDonald had a farm, E-I-E-I-O.\n");
}
```

there is no string type in C; instead, represent as a char array (or char*)

4

User input

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

void oldMacVerse(char*, char*);

int main() {
    char animal[30];
    printf("Enter an animal name: ");
    scanf("%s", &animal);

    char sound[30];
    printf("Enter the sound it makes: ");
    scanf("%s", &sound);

    oldMacVerse(animal, sound);
    return 0;
}

void oldMacVerse(char* animal, char* sound) {
    printf("Old MacDonald had a farm, E-I-E-I-O.\n");
    printf("And on that farm he had a %s, E-I-E-I-O.\n", animal);
    printf("With a %s-%s here, and a %s-%s there,\n",
           sound, sound, sound, sound);
    printf("  here a %s, there a %s, everywhere a %s-%s.\n",
           sound, sound, sound, sound);
    printf("Old MacDonald had a farm, E-I-E-I-O.\n");
}
```

input can be read using `scanf`

- 1st parameter is a format string
- 2nd parameter is an address (here, obtained via address-of operator &)

note: must allocate the space for the text first

- here, a `char` array of size 30
- can then access as `char*`

5

`string.h` library

manipulating string values is tedious in C

- must use functions from the string.h library

```
char* str = "foobar";          // automatically allocates space & assigns

char* copy;
strcpy(copy, str);             // copies "foobar" string into copy array

int len = strlen(str);         // gets length of the string

strcat(str, "!");              // sets str to be "foobar!"

int compare = strcmp(str, "boo");
                               // evaluates to negative if <,
                               //             0 if ==,
                               //             positive if >
```

6

Control structures

- can access individual characters using [ ]
  - since a char array
- same control structures as C++/Java
  - if/else, switch
  - while, do-while, for
- there is no boolean type in C
  - if/while controlled by int values
  - 0 = false; else true

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

int isPalindrome(char*);

int main() {
    char input[20];
    printf("Enter a word: ");
    scanf("%s", &input);

    if (IsPalindrome(input)) {
        printf("%s is a palindrome\n", input);
    }
    else {
        printf("%s is NOT a palindrome\n", input);
    }

    return 0;
}

int isPalindrome(char* word) {
    int len = strlen(word);

    int i;
    for (i = 0; i < len/2; i++) {
        if (word[i] != word[len-i-1]) {
            return 0;
        }
    }
    return 1;
}
```

7

#define

- you can define constants using #define
  - this is a preprocessor directive
  - the first step in compilation is globally replacing each constant with its value
- #define is for user convenience only
  - makes no difference in the compiled code

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

#define MAX_LENGTH 20
#define BOOLEAN int
#define TRUE 1
#define FALSE 0

BOOLEAN isPalindrome(char*);

int main() {
    char input[MAX_LENGTH];
    printf("Enter a word: ");
    scanf("%s", &input);

    if (isPalindrome(input)) {
        printf("%s is a palindrome\n", input);
    }
    else {
        printf("%s is NOT a palindrome\n", input);
    }

    return 0;
}

BOOLEAN isPalindrome(char* word) {
    int len = strlen(word);

    int i;
    for (i = 0; i < len/2; i++) {
        if (word[i] != word[len-i-1]) {
            return FALSE;
        }
    }
    return TRUE;
}
```

8

Function parameters

all parameter passing is by-value

- but the & operator means the user can achieve by-reference passing

- get the address of the variable using &
- pass the address to the function as parameter
- then dereference the address if needed

```
#include <stdio.h>

void getValues(int*, int*);
void process(int*, int*);
void display(int, int);

int main() {
    int x, y;
    GetValues(&x, &y);
    Process(&x, &y);
    Display(x, y);

    return 0;
}

void getValues(int* a, int* b) {
    printf("Enter two numbers: ");
    scanf("%d%d", a, b);
}

void process(int* a, int* b) {
    if (*a > *b) {
        int temp = *a;
         *a = *b;
         *b = temp;
    }
}

void display(int a, int b) {
    printf("%d + %d = %d\n", a, b, (a+b));
}
```

9

C arrays

by default, C arrays allocation is:

- static (allocated on the stack at compile time), if a global variable
- fixed stack-dynamic (size is fixed at compile time, memory is allocated on the stack during run time), if a local variable

```
char letters[10];

int counts[NUM_LETTERS];
```

- access elements using indexing (via [ ])

```
letters[0] = '*';

int i;
for (i = 0; i < NUM_LETTERS; i++) {
    counts[i] = 0;
}
```

- unlike Java, there is no length field associate with an array – you must keep track!

10

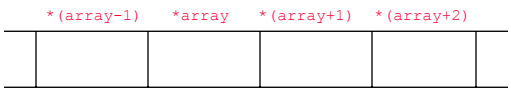
C arrays

arrays and pointers are linked in C

- can think of an array as a pointer to the first element
- when referred to, the array name is converted to its starting address

```
int counts[NUM_LETTERS];              // counts ≡ &counts[0]
```

- array indexing is implemented via pointer arithmetic: `array[k] ≡ *(array+k)`



the pointer type determines the distance added to the pointer

- because of the nature of arrays, no bounds checking is done in C
  *** EXTREMELY DANGEROUS!!!! ***

11

Array example

because of the size limit, storing an arbitrary number of items is ugly

- must set a max size
- as you read in items, must keep count and make sure don't exceed the limit
- must then pass the size around with the array in order to process

- note: could have written
  `int* nums`
  instead of
  `int nums[]`

```
#include <stdio.h>
#define MAX_SIZE 20

void getNums(int[], int*);
int smallest(int[], int);

int main() {
    int numbers[MAX_SIZE];
    int count = 0;

    getNums(numbers, &count);
    printf("The smallest number is %d\n",
           smallest(numbers, count));

    return 0;
}

void getNums(int nums[], int* cnt) {
    int nextNum;

    printf("Enter numbers (end with -1): ");
    scanf("%d", &nextNum);
    while (nextNum != -1 && *cnt < MAX_SIZE) {
        nums[*cnt] = nextNum;
        (*cnt)++;
        scanf("%d", &nextNum);
    }
}

int smallest(int nums[], int cnt) {
    int small = nums[0];
    int i;
    for (i = 1; i < cnt; i++) {
        if (nums[i] < small) {
            small = nums[i];
        }
    }
    return small;
}
```

12

Data structures

can define new, composite data types using `struct`

- `struct { … }` defines a new structure
- `typedef … NAME;` attaches a type name to the struct

note: a struct is NOT a class

- there is no information hiding (i.e., no `private`)
- there are no methods

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

typedef struct {
    int x;
    int y;
} Point;

double distance(Point, Point);

int main() {
    Point pt1;
    Point pt2;

    pt1.x = 0;
    pt1.y = 0;

    pt2.x = 3;
    pt2.y = 4;

    printf("Distance = %f\n",
            distance(pt1, pt2));

    return 0;
}

double distance(Point p1, Point p2) {
    return sqrt(pow(p1.x - p2.x, 2.0) +
                pow(p1.y - p2.y, 2.0));
}
```

13

Dynamic memory

by default, data is allocated on the stack

- to allocate dynamic memory (from the heap), must explicitly call `malloc`
- `malloc` returns a `(void*)`
- must cast to the appropriate pointer type
- when done, must explicitly deallocate memory using `free`

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

void getNums(int[], int);
int smallest(int[], int);

int main() {
    int* numbers;
    int count = 0;

    printf("How many numbers? ");
    scanf("%d", &count);
    numbers = (int *)malloc(count * sizeof(int));

    getNums(numbers, count);
    printf("The smallest number is %d\n",
           smallest(numbers, count));

    free(numbers);

    return 0;
}

void getNums(int nums[], int cnt) {
    int i;
    printf("Enter the numbers: ");
    for (i = 0; i < cnt; i++) {
        scanf("%d", &nums[i]);
    }
}

int smallest(int nums[], int cnt) {
    int i, small = nums[0];
    for (i = 1; i < cnt; i++) {
        if (nums[i] < small) {
            small = nums[i];
        }
    }
    return small;
}
```

14

Bitwise operators & libraries

in addition to standard math operators, C has low-level bitwise operators

- `&` bitwise AND
- `|` bitwise OR
- `^` bitwise XOR
- `~` bitwise NOT (one's complement)
- `<<` left shift
- `>>` right shift

also has a variety of standard libraries

- `math.h` mathematical functions
- `stdio.h` input/output functions, including file processing
- `stdlib.h` various useful functions, including memory allocation, system calls, random, searching & sorting, ...
- `time.h` functions dealing with time
- `sys/socket.h` functions for interprocess communication, networking

15

File processing

functions for reading and writing files are also defined in `stdio.h`

- `fopen` opens a file and returns a file pointer (`FILE*`)
- the 2nd parameter specifies the type of access ("r" = read, "w" = write)
- `fgetc` reads an int (i.e., a character)
- `fputc` writes an int (i.e., a character)
- `fclose` closes the file

```
#include <stdio.h>

void copyFile(char*, char*);

int main() {
    char* inName = "input.txt";
    char* outName = "copy.txt";

    copyFile(inName, outName);

    return 0;
}

void copyFile(char* inName, char* outName) {
    FILE* inFile = fopen(inName, "r");
    FILE* outFile = fopen(outName, "w");

    int ch;
    while ((ch = fgetc(inFile)) != EOF) {
        fputc(ch, outFile);
    }

    fclose(inFile);
    fclose(outFile);
}
```

other functions for reading other data types also exist

16

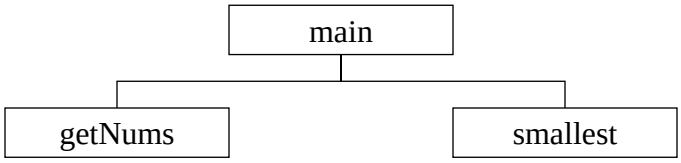
Top-down design

the dominant approach to program design in the 70's was top-down design

- also known as *iterative refinement*

general idea:

- focus on the sequence of tasks that must be performed to solve the problem
- design a function for each task
- if the task is too complex to be implemented as a single function, break it into subtasks and repeat as necessary

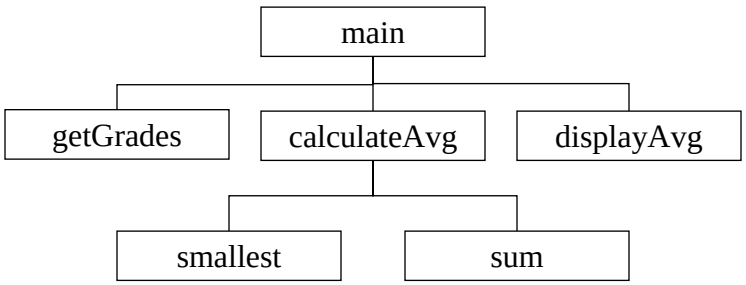


17

Design example

suppose we wanted to calculate a homework average, where the lowest grade is dropped

- top down design?



18