

CPS104
Computer Organization and Programming
Lecture 2 : C and C++

Robert Wagner

Slides available on:
<http://www.cs.duke.edu/~raw/cps104/Lectures>

Overview of Today's Lecture:

- **C philosophy**
 - * **Debugging**
 - * **Good Programs**
- **C arrays and strings**
- **C procedures**
- **C Input Output**

C++ vs C: Philosophy

- **C++ emphasizes reuse of code, ENCAPSULATION of data types**
- **The OO philosophy:**
 - * **Define (real-world) objects, each with variables that indicate their “state”, and each changeable ONLY by their own member functions**
 - * **Insulate the programmer from machine level**
 - Impossible to apply an inappropriate operation
 - Hard to reference memory that isn't holding an object
 - * **Overloading of operators simplifies program appearance --- more easily read**

C Philosophy

- **C REVEALS machine level details**
 - * **DESIRABLE for this course!!!**
- **Excellent language for explaining how computer really works**
- **Procedure-oriented, not object oriented**
 - * **Any procedure may be applied to any item**
 - * **BE CAREFUL**
- **Has fewer concepts than C++:**
 - * **No inheritance hierarchy**
- **Uses global arrays, and explicitly allocated arrays, NOT “new”**
- **Each “executed” mention of a C name corresponds to ONE machine language instruction (unit time)**

C Philosophy (cont'd)

- **C allows programmers to write very machine-efficient code**
- **Does not require use of “access functions” to reference a global array or collection of objects**
 - * **Offers no “protection” against incorrect operations on them, either**
- **Does NO “array bounds checking”**
 - * **Programmer is responsible for using legal subscript values**

Form of a C Program

- List of GLOBAL variable and procedure DECLARATIONS
- The “main” procedure must exist, and is executed on start
- Procedures may contain LOCAL declarations of variables
 - * Declarations are written before any code
- Arrays are declared by following a variable name with [*constant expression*]:
 - * `int xyz[49];`
 - Declares xyz to be an array of 49 integers, xyz[0]...xyz[48]
 - Reserves space in memory to hold all 49 of them
 - Reference an array element by xyz[*expression*].
- GLOBAL arrays elements initially hold 0. LOCAL hold garbage.

Designing C Programs

- **First concern is the METHOD to be used for the solution**
 - * **Do I need to record input data for later processing? Or process input as it is read?**
 - * **What computation is needed?**
 - * **If a record is made, what form should it take?**
 - **Parallel arrays: `int A[100], B[100]; A[i]` and `B[i]` (same `i`) describe the `i`-th “object”**
 - **Objects can be “connected” logically: `C[i]` can hold the index of an object related in some way to `(A[i], B[i])`**
 - * **Speed is increased if direct indexing can be used, rather than searching**
- **Later, decide how to read input, and write output**

Debugging

- **C (and Assembly Language) programs are seldom correct as first written**
- **You must develop DETECTIVE skills, to find places where what you THOUGHT you told computer to do differs from what IT does**
 - * **printf statements on procedure entry and exit**
 - * **use of “gdb” (breakpoints; examine variables)**
- **Fixing the problems is usually easy**
- **Finding a problem is hard, because computer may run a long time, after the REAL error**

Testing

- **Your program must work for ANY “valid” input**
- **You CANNOT assume that the test cases given adequately test your program**
- **You must design your own:**
 - * **“Stress test” it -- by designing test cases that exercise the parts of the program that YOU find most confusing and difficult**
 - * **Add test cases that check the “special cases”**
- **Design your program to avoid duplication of code to handle special cases -- if designed well, so each part of the program performs some “general” operation, one or two tests suffice to test each such part**

Positive Attributes of Programs

- **Correct: Meets the specifications.**
- **Understandable to the reader**
 - * **Clear, precise documentation of each subroutine**
 - **Mention each argument and result**
 - **Use “mathematical English”**
 - * **Avoid error-prone C constructs**
- **Highest possible execution speed**
 - * **Each programmer name or constant costs one unit of time each time it is “executed” (approximately)**
- **Fewest possible C tokens in the source text**
 - * **Tokens are: Names, Numbers, Strings, Operators**
 - * **Use parentheses and comments freely**

C Language Material

- The following on-line source seems to be a good reference :
C Language Course
www.strath.ac.uk/CC/Courses/NewCcourse/ccourse.html
- Another source on the Web is:
www.cm.cf.ac.uk/Dave/C/CE.html
- The material on integer arithmetic, characters, and pointers will be important, along with simple I/O: `getchar()` `putchar()`, `printf()` and `scanf()`. You should learn how to use ***for*** and ***while*** loops, ***if*** statements and subroutines, as well.

Pointers

- A “pointer” is an integer intended to be used as an address for some object in memory
- In C, the type of the object pointed to must be declared, along with the pointer:
 - * int *P; Declares P to hold a pointer to an integer
 - * char *C; Declares C to point to a character
- A pointer to any variable or array element can be built by the “address of” operator “&”: &X[i] is an integer, giving the address of X[i].
- An array reference A[I] is equivalent to *(&A+I), where “*” “de-references” its argument -- obtains the contents of the memory location the argument points to
- Addition of pointers and integers “scales” the integer to the size (in bytes) of the pointers reference type, so P+1 computes an address 4 bytes more than that held in P because each int occupies 4 bytes of memory.

C Strings

- **C strings are represented as arrays of characters:**
 - * **char c[10];** Declares space for a 9-character string
 - * **Why 9?** Because a C string is terminated with a special “sentinel” character `\0`. The C string constant “A STRING” is a pointer to the first of a sequence of bytes, whose contents are ‘A’, ‘ ‘, ‘S’, ‘T’, ‘R’, ‘I’, ‘N’, ‘G’, and ‘\0’.
- **There are NO C “operators” on strings. All string operations are handled by programs**

C Memory Allocation

- **C has no “new” operator**
- **When a new object is needed, space for its state variables must be reserved, or “allocated”**
- **This could be done from a programmer-declared array, OR by use of the library function “malloc”**
- **I recommend declaring “large enough” arrays (whose size must be fixed before compilation).**
 - * **Choose size based on statistics about input data**
 - **Ask problem poser**
- **The library function “(void *) malloc(int n)” can be used to reserve n bytes of memory, and return a pointer to the first byte of that area of memory.**

C Procedures and Naming

- All C procedures are accessible (callable) anywhere in the file after their declaration (header or signature) has appeared
- Declarations appearing outside procedures are ALSO accessible anywhere later in the file -- including global variables
- The procedure “main” must be defined
`int main(int argc, char ** argv);`
- Names declared inside procedures (including parameters) are local to those procedures
- Procedures may not be declared inside procedures

C Input / Output (a small subset)

- **#include <stdio.h> // Include signatures and constant defs**
- **char getchar(void); // Return next character from stdin**
 - * **stdin is UNIX default input stream; opened automatically when program is started**
- **int printf(char *, ...);**
 - * **formatted output, to stdout**
 - **default output stream**
- **int scanf(char *, ...);**
 - * **formatted input, from stdin**
- **Definitions of what these and other functions do can be found by running “man -s<section> <name of subroutine>“ where <section> is 3s or 3c.**

Connection to UNIX

- UNIX command line:
 - * ***prog arglist [<in] [>out]***
 - Runs *prog* connecting its “stdin” to *in*, “stdout” to *out*
 - *in* and *out* are files on disk (so is *prog*)
 - If either *<in* or *>out* is absent, that file is the console
 - * **UNIX loads *prog* into memory, calls its `main(argc, argv)`**
 - `argc` is set to the number of arguments in `arglist`
 - `argv[i]` is set to the *i*-th argument (a string)
 - `argv[0]` is set to the name of the program
 - * **`xyz 35 abc <in.1`**
 - calls `xyz.main(2, {"xyz", "35", "abc"})`
 - `stdin` is file “*in.1*”, `stdout` is the screen

C Input / Output: printf

- **Returns EOF if error occurs, otherwise != EOF**
- **Variable number of arguments, no types specified**
- **First argument is a format string, telling how each subsequent argument is to be converted (this string specifies the type of each argument, and how many columns it occupies on the output page)**
- **printf sends a stream of characters to stdout**
- **The formatting string includes:**
 - * **text to be output as is**
 - * **Characters to control the printer, like \n (new line) and \t (tab)**
 - * **Conversion specifiers**

C Input / Output: printf Conversion Specifiers

- **Each specifier begins with % and ends with a field type character. Between may be some characters which give minimum field width, and precision to use**
- **Examples:**
 - * **%s** :- **Print a string. Field width is string's length**
 - * **%7s** :- **Print 7 chars of a string, pad with space**
 - * **%d** :- **Print an integer**
 - * **%12d** :- **Print an integer in a field of at least 12 chars, pad with space**
 - * **%012d** :- **Print an integer in a field of at least 12 chars, pad with 0**
 - * **%10.2f** :- **Print a float, in a field of 10 chars, show 2 digits right of the decimal point**

Example 1

```
#include <stdio.h>
main() {
    int a = 23;
    char * s = "Have a happy one!";
    float f = 12.678;

    printf("print: Ival=%d, Fval=%f, String=%s\n",
          a, f, s);
}
```

Result:

print: Ival=23, Fval=12.678, String=Have a happy one!

Example 2

```
#include <stdio.h>
main() {
    int a = 23;
    char * s = "Have a happy one!";
    float f = 12.678;

    printf("print: Ival=%5d, Fval=%7.2f, String=%10s\n",
          a, f, s);
}
```

Result:

```
print: Ival= 23, Fval= 12.68, String=Have a hap
```

Programming with Ingenuity

Problem: Report the number of words which begin with each possible letter.

Word == Seq of one or more letters, surrounded by non-letters

Method: At start of word, increment the count for its first letter

Linear search for the letter?

Hash table search for the first letter?

Better method?

How detect that it IS a letter? How detect start of word?

Ingenuity (cont'd)

- **Linear search? 52 letters, so ~ 26 loop iterations @ 10 ops per iteration = 260 ops per letter read**
- **Hash table lookup? About 2 “probes” per letter @ 10 ops per probe = 20 ops per letter**
- **Better method: Use ASCII code as index into table!!**
 - * **Ct[c]++; // 2 ops per letter**
 - * **if (let[c]) ... // Pre-initialize let[i]==0 if i does NOT // code a letter, 1 if i DOES code one**
- **Design “heart” of program -- part that will be executed most -- FIRST, fit rest of program around it, so heart does not have to change, to get the details correct**

Ingenuity (3)

// Assume file holds N characters, Q of them non-letters

#include <stdio.h>

Int c, PrevlLet=0, let[256], Ct[256];

Hrt1() {

while ((c=getchar())!= EOF) {	// 3 ops, 13 toks
if (! PrevlLet)	// 1 op, 5 tokens
 Ct[c]++;	// 2 ops, 6 toks
 PrevlLet = let[c];	// 3 ops, 7 toks
 }	// 0 ops, 1 tok
 }	// loop above: 7N+2Q ops, 33 tokens

Note: Ct[c]++ increments non-letter elements, which output routine ignores; Output routine combines UC and LC counts.

Ingenuity (4)

Is faster possible? Principle: “skip over” letters inside words.

```
Hrt2() {  
    while ((c=getchar())!=EOF) { // 3 ops, 13 toks  
        if ( let[c] ) { // 2 ops, 8 toks  
            Ct[c]++; // 2 ops, 6 toks  
            while (((c=getchar())!=EOF) && let[c] ) ;  
                // 5 ops, 19 toks  
        }  
    }  
} // 5N+2W ops, where W words in file; 46 toks
```

Ingenuity (5): Comparison

Guess that text file contains 5 letters per word, and each word is followed by an average of 1.5 non-letters. $W=N/5$, $Q=1.5*N/5$

Guess the rest of program contains 100 tokens.

	Toks	Speed	Eleg	Speed
Hrt1	133	$7N + 2*0.3*N = 7.6N$	10	7.1
Hrt2	146	$5N + 2*N/5 = 5.4N$	$10*133/146 = 9.1$	10

Conclude: Hrt2 should earn more total points than Hrt1.

(assumes Hrt1 is most elegant, Hrt2 is fastest program)

Programming Example

```
#include <stdio.h>
int White[256];    // Sets White[0..255]=0; 0 acts as "false" in tests.
main(){          // Parameters not needed
    char c, s[256], *p;    // s will hold the string
    White[' '=White['\t']=White['\n']=1; // non-zero acts as "true" in tests
    p = s;            // Set pointer p to the address of s[0]
    while ( (c=getchar() ) != EOF ) { // skip white space
        if ( ! White[c] ) break;
    }
    *p++ = c;        // Increment p, after storing c "through" p's old value
    while ( (c=getchar() ) != EOF ) { /* Copy non-white space into s */
        if (White[c] ) break;
        *p++ = c;
    }
    *p = '\0';      // Null terminate the string
    printf("%s\n",s); // Print first word of input.
}
```

Programming Example Using scanf

```
#include <stdio.h>
main(){          // Parameters not needed
    char s[256]; // s will hold the string
    int J, K;
    scanf("%d %d %s", &J, &K, s);
        /* scanf's arguments are POINTERS. Even "s" is
        changed by the compiler into a pointer, because this
        array name has no subscript here. */
    // This reads two white-space separated integers, and one
    // string, consisting of non-white-space characters.
}
```

Summary

- **C programs are procedure oriented, not OO**
- **They reflect actual features of machine hardware**
 - * **`A>>2, A&1` // Each performs one machine instruction**
- **Allow programmer control over sequence of instructions generated, primarily for efficiency at run-time**
 - * **`while (t[k] = s[k]) k++;` // TRICKY. Copies until `S[k]==0`**
- **Naming conventions are primitive**
- **No run-time array bounds or pointer checking**
- **Very useful for describing what machine hardware is supposed to do**