================================================================

Block 15

How do you write a function like printf() which can take an

arbitrary number of arguments, sense them, and process them?

/\*-----------------------------------------------------------

\* varargs1.c -writing funcs with variable number of arguments.

\* Written Sun May 21 20:49:52 PDT 2000

\* by TEK.

-----------------------------------------------------------\*/

#include <stdio.h>

#include <stdlib.h>

#include <stdarg.h>

int sum(int , ...) ; // Function Prototype.

int main()

{

/\*----------

\* I'm gonna call sum() with 9 ints...

\* First arg is the number of arguments following:

-----------\*/

printf("%d\n", sum(9, 1,2,3,4,5,6,7,8,9) ) ;

printf("%d\n", sum(4, 2,4,6,8) ) ;

} // main() ends

/\*-----------

\* I define the function with an elipsis (...), there

\* MUST be at least ONE defined parameter preceeding

\* the variable-length list, for the function to use

\* to determine the list's length.

-----------\*/

int sum(int x, ...)

{

va\_list ap ;

int c, sum = 0 , i = 0 ;

/\*-----------------

make ap point to the first UNNAMED argument by

providing the name of the LAST NAMED argument

-----------------\*/

va\_start( ap, x ) ;

for (i = 0 ; i < x ; i++ )

{

/\*

\* each call to va\_arg must specify WHERE to put the

\* result, and the TYPE of the thing to grab, so the

\* compiler can calculate how many bytes it has to

\* retrieve off the stack. SOME programs like this

\* won't use a for() loop, since all args won't be

\* homogenious, instead, the code would process each

\* arg explicitly. To understand this concept,

\* meditate on the int printf( const char \*, ...) ;

\* function....

\*/

c = (int) va\_arg (ap, int ) ;

printf("sum: %d c == %d\n", sum, c) ;

sum += c ; // add to sum (the actual working func code...)

} // for(i) ends

va\_end (ap) ;

return sum ;

} /\* sum() ends \*/

====================================================

Templates

SUMMARY

TEMPLATES ARE A "GENERALIZED" WAY OF PROGRAMMING, WHERE YOU IGNORE

DATA TYPES, AND WRITE CODE WHICH CAN OPERATE ON ANY KIND OF DATA.

SKIM CHAPTER AND REMEMBER THAT "TEMPLATES" EXIST, AND FACILITATE THAT

"GENERALIZED" APPROACH.

THE EASIEST WAY I'VE FOUND TO LOOK AT TEMPLATES WAS TO LEARN THE

map TEMPLATE. IT ALLOWS YOU TO DO SOME AMAZING PROCESSING WITH

SHORT PROGRAMS.

This gives us a chance to look practically at templates, but

moreover, at class objects and how they're used. Read the

map\*.cpp programs and try them out.

/\*-----------------------------------------------------------------

\* wordlengthcensus.cpp - sort all words by length into hashes

\* and count each one.

A "map" is a kind of associative array which means that the

map "values" are found via an indexing "key". They may be

of any type. A map of strings may be indexed with ints, after

the fashion of an array of strings. Here I'm using strings to

index ints.

Understand what you're reading: the map has been implimented

as a template.

-----------------------------------------------------------------\*/

#include <stdio.h>

#include <iostream>

#include <map>

using namespace std;

int main (int argc, char \*argv[], char \*\*env)

{

const int SIZE = 50 ;

std:: map<string, int> count[SIZE] ;

unsigned int len, i, j ;

string s ;

cin >> s ;

while (!cin.eof())

{

for (i = 0 ; i < s.length() ; i++) // strip junk out of string

{

s[i] = tolower(s[i]) ;

if (!isalpha(s[i]))

{

s.erase(i,1) ;

i-- ; // back up and test the next character, don't skip it!!

}

}

len = s.length() ;

if (len > 0 && len < SIZE) // store in array of maps/count how many seen

count[len][s]++ ;

cin >> s ;

} //while

std::map<string, int>::iterator iter ; // iterator steps through maps (like pointer)

for (j = 0 ; j < SIZE ; j++)

{

for ( iter = count[j].begin() ; iter != count[j].end() ; iter++)

{

cout << (\*iter).first << "\t\t" ;

cout << (\*iter).second << endl ;

} // for iter

} // for j

} // main

END SUMMARY

====================================================

Bitwise Operations, CStrings, Structs, efficient algorithms,

Boolean operators.

AND OR XOR NOT

& | ^ ~

=================================================

READ LINKED-LIST ASSIGNMENTS, (asst7) THINK ABOUT HOW YOU

MIGHT ATTACK THEM.

THE POINT OF LINKED-LIST ASSTS LIKE THESE IS TO TEACH YOU

THE TECHNIQUES OF ALLOCATION AND LINKING OF STRUCTS. THE

ACTUAL "SHAPE" THE LINKED STRUCTURES TAKE ("DATABASE

TOPOLOGY") IS THE SUBJECT OF A DATA STRUCTURES CLASS.

END SUMMARY

Ponder the difference between "exercise" programs, and real "Applications

Programming". One tries to show you how to do something. The other

requires that you mobilize everything you know, and research the rest,

in order to accomplish a task which has been set you. This is what you

get paid-- or fired for.

Now consider:

struct prs

{

bool deleted ;

char name[30] ;

int num ;

struct prs \* prev ;

struct prs \* next ;

} \*P, \*phere, \*pnew ;

What's interesting here? We've seen bools, chars, and ints,

even pointers. But these pointers point to the very type

of thing that contains them.

You want a Linked List where P points to a node, which points

to another node, etc.

P = new struct prs ;

Optional Assignment 7: Create an in-memory Linked List.

ALL THAT A PROGRAM "KNOWS" IS CONTAINED IN VARIABLES.

Use struct prs \* P as the "anchor".

Use struct prs phere as the "current record".

Use struct prs pnew as the "new record".

Assignment: Create an in-memory Linked List.

Do not use list C++ list objects. This is code that can be

written in C (or assembler).

Option A: 50 points. No disk I/O.

Option B: 60 points: linked list with disk storage

Option D: 70 points access list via sortable arrays of pointers

to nodes, and provide multiple sort views into the database from

the main menu.

Option : +10 points allow edit and delete of records

=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=

Using the discussion in course materials, create an in-memory CIRCULAR

DOUBLY LINKED linked-list database program which displays a menu screen

similar to the following...

LINKED LIST PROGRAM

e) Add new agent to end of list

l) List all agents

n) Display Next record

p) Display Previous record

q) Quit program

=============================================================

Current Record:

0X567A:::0X620A:::0X640D James Bond 001

...and which processes SINGLE KEYSTROKE INPUT from the user.

On the Linux system, this is possible using the instructor's

/var2/local/include/getchne.h

code, on Microsoft, use the getch() function which should be in the

System.Console namespace now, or listed in conio.h.

If you use the conio.h library under Windows, Write the code so that

a single #define UNIX command allows the code to be compiled and run on

either system.

You can choose between STRUCT and CLASS record structures as follows:

struct prs

{

int useme ;

char name[30] ;

int mum ;

struct prs \* prev ;

struct prs \* next ;

} \*P, \*phere , \*pnew ;

OR:

class prs

{

public:

int useme ; // deleted?

char name[30] ;

int num ;

prs \* prev ; // prev node

prs \* next ; // next node

prs() ; // constructor

~prs() ; // destructor

} \*P, \*phere , \*pnew ;

(now, in a Truly Obsessively Compulsively Disordered OOP shop, you'd use:

class prs

{

private:

static int totalrecs ;

int useme ; // deleted?

char name[30] ;

int num ;

prs \* prev ; // prev node

prs \* next ; // next node

public:

prs() ; // constructor

~prs() ; // destructor

int getUseme(void) ;

void setUseme(int) ;

char \* getName(void) ;

void setName(char \*) ;

int getNum(void) ;

void setNum(int) ;

prs \* getPrev() ;

void setPrev(prs\*) ;

prs \* getNext() ;

void setNext(prs\*) ;

void edit\_rec(void) ;

} ;

...but this's for Christmas Vacation, let's try to keep focused on

linking the list.)

Access to the list will be via a prs \* "anchor" ; variable

which will serve as the access point to the list by pointing to the

first record. Thereafter, each record will be accessed via the preceeding

record. Each record should point "forward" to the next record and "back"

to the preceeding record.

SUGGESTIONS FOR FUNCTIONS:

prs::print\_rec(): display a record, including pointer contents.

pop\_menu() : clear screen, display menu, get choice, and return number of

choice to calling routine. In addition to displaying the menu choices,

it should display the "current" record in the list, that is, the one

phere points to.

listall(): calls print\_rec to display all records in list.

alloc\_rec() : creates a record

edit\_rec() : prompts the user for input and fills the record fields.

Design so that later on you can include extra functions.

Work for simplicity and clarity.

MAJOR HINTS: To do "display next record" or "display

previous record" all you have to do is: "phere =

phere->next" or "phere = phere->prev" and then re-display

the program screen. If you think about it, you ought to be

able to use the same logic in the "list all agents" option.

Symbolic Situation: three records linked into list (1,

2, and 3), it is just after allocating new record (7),

but before linking it into list ("num" field omitted

for space):

p1: &1 pnew: &7

&1 &2 &3 &7

name: one name: two name: three name: dOxk2

prev: &3 prev: &1 prev: &2 prev:^&(&43 (junk)

next: &2 next: &3 next: &1 next:/9Xik5 (junk)

Situation just after linking in the new record and

getting name:

p1: &1 pnew: &7

&1 &2 &3 &7

name: one name: two name: three name: four

prev: &7 prev: &1 prev: &2 prev: &3

next: &2 next: &3 next: &7 next: &1

=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=

Create a program which does the following:

0. Manipulates Records as Class prs Objects.

(Class prs fields defined below:)

1. Loads Linked List from diskfile, if it exists. EITHER:

in one single block I/O operation, into an area of ram allocated precisely

to hold it as an array of structs. Once in RAM, step through the array

of structs, connecting the pointers in each record to the records ahead

and behind them, to produce a doubly-linked circular list (serial linkage,

both "next" and "prev" pointers correctly set)

OR: read file one record at a time, allocate each new record discretely,

and link it into the list. After file is read, close it, and display

menu for users.

If no file found, advises user and creates new database from user input.

New structs may then be added one at a time, with user input from

the keyboard. When the program ends, save all structs, one at a time,

to a binary disk file. The design of the struct is up to you, and you

may re-use as much of Assignment 6 as you like, to speed up the task.

LINKED LIST PROGRAM

e) Enter a new agent to end of list

n) Next record

p) Previous record

l) List agents in database

d) Delete agent file

q) Quit program

========================================

Current Record:

James Bond 001 Previous: 0X567A Current: 0X620A Next: 0X640D

2. Allows the user to

A.) Display the whole list to the screen,

B.) Add a record to the end of the list by keyboard input of fields contents.

C.) Display the next or previous struct in list.

D.) Quit program saving the list to disk.

3. Uses binary (unformatted fread or system-level) i/o.

PROGRAMMING SUGGESTIONS/HINTS: Modularize the program so

that reading in and reorganizing pointers in the list,

displaying a struct, writing a single struct to disk,

writing the array out, getting struct data from the

keyboard, are in separate procedures, all driven by a

main function built around a user menu. If you write this

program well, it will serve you as the basic example for

many useful programs you will want to develop for yourself

in years to come. Work for simplicity and clarity.

Extra Credit: Design program to include delete functions,

search functions, and edit functions. Extra Extra: use

dynalloced arrays of struct pointers to provide SORTED

"listall" access to list on NAME and NUMBER fields by using

a sorted array of pointers. (Download and run lldex binary

for example.)

Either alternative uses exactly the same record specs.

Sample data files are provided, which have records in

the form:

You may use eiher a:

struct prs

{

int useme ; // indicates record deleted

char name[30] ;

int num ;

prs \* pprev ; // previous rec in list

prs \* pnext ; // next rec in list

} or a:

class prs

{

int useme ; // indicates record deleted

char name[30] ;

int num ;

prs \* pprev ; // previous rec in list

prs \* pnext ; // next rec in list

prs() ; // constructor

~prs() ; // destructor

}

If you use the class, you might wish to write the constructor method

so that it links itself, so the external program need only handle the

pointers P->prev->next and P->prev; (Or not, it's up to you.)

(This assignment points to the subject material you will encounter in an

upper division class in Data Structures. Students who want to struggle

with this may email the instructor (tharrisb@sdccd.edu) for solutions to

these two programs after the semester's end.) If you don't see the

point, you will some day, when you're doing real work. Without this

kind of linked data structure, programs would have to use arrays,

and have no clean way of handling arbitrary numbers of duplicate input

lines, resulting in the loss of duplicate input data, or slow runtime

resolving collisions in an array. But with a linked data structure,

such complications evaporate.

Also, being able to arrange data in an organized way makes searching

databases very fast. The "B-Tree" is the foundation of modern high-speed

searching, and this assignment teaches all the tools needed to use one.

All contents copyright (C) T.E.H., all rights reserved. None of this

text may be displayed or stored on any public access or private access

system without the written consent of the author.

Full-on extra point mode accessing nodes via sorted pointer arrays:

(Arrays point to records, and each is sorted on a different field.

The nodes don't change positions in the list, that would be slow.

This is what is meant by "indexing" databases. In fact, this is the

same kind of code Ashton Tate used on the first dBase product, and

made millions.)

LINKED LIST PROGRAM

d (elete current record

e (nter new agent

l (ist all agents

m (odify current record

n (ext agent

p (revious agent

q (uit program

s (ort by name

# (ort by number

t (hrow away file

DESIGN CONSIDERATION:

There are TWO basic states of nature:

1) P has no address in it, which indicates that the user

is creating the FIRST node, and

2) P has an address in it, indicating that a list already

exists.

Obviously, it's crucial to initialize all your pointers

when the program starts.

This kind of assignment really tells how much you know.

If I had three struct prs objects, one, two, and three,

and I wanted to link one to P, two to one, and three to

two, and have them make up a doubly-linked circular list.

struct prs \* P , \* phere , \*pnew ;

P = 0 ;

phere = P ;

pnew = one ;

phere->next = pnew ;

pnew -> prev = phere ;

phere->prev = pnew ;

pnew->next = phere ;

At this point, we have a circular linked list of two nodes.

You can go forward or backward without limit, circling.

Now, to introduce a new node, we first go to the last node

in the list. This is the node P-> prev points to. We tuck

the new node in at the end, point it forward to P and backward

to the former P->prev, and join up the other two pointers:

phere = P->prev ;

pnew = &three ;

pnew -> next = P ;

pnew -> prev = P->prev ;

P -> prev = pnew ;

phere -> next = pnew ;

Work this out with a drawing until you understand and

believe it.

.............................

Every time the user creates a new node, first you allocated a struct

prs, then you link it in to the list, then you pass its address

to edit\_rec(struct prs \*) and prompt the user to fill in the data

fields.

I advise printing the node records on a single line of

output, so you can verify linkage by comparing the hex

addresses of prev , here , and next.

Enter name: three

Enter number: 3

0003 total records

e) Enter a new agent

l) List agents in file

n) Next Struct in List

p) Prev Struct in List

q) Quit program

0x804c040 0x804c078 0x804c008 three 3

0x804c078 0x804c008 0x804c040 one 1

0x804c008 0x804c040 0x804c078 two 2

0x804c040 0x804c078 0x804c008 three 3

(prev) (here) (next)

If you observe closely, you'll see a slanted relationship,

where each prev points right and up to the here on the

previous line, and each next points down and left to the

here on the next line. That is to say, each here points up

and right to the next on the previous line, and each here

points down and left to the prev on the following line.

Think about this until you understand it.

If you cannot reliably and perfectly link nodes in a simple

sequentially linked database, you will be ineligible for

your upper division data structures class until you can.

What we're doing, a double-linked circle, is just an easy

exercise. In the real world, these same techniques are

used to make B-Trees, Red/Black Trees, null-terminated

lists, n-dimensional data structures, etc. This set of

tools opens the door to the most powerful problem-solving

tools in the world of simple programming. I used them

when I wrote unsort.cpp, since no array structure was

suitable for unsorting data which might have duplicate

keys, I chose a B-Tree instead.

It worked the first time, such was the power of a

good algorithm.

/\*-----------------------------------------------------------------

\* map1.cpp - demo the map construct

\* Mon Jan 10 14:00:55 2005 TEK

A "map" is a kind of associative array which means that the

map "values" are found via an indexing "key". They may be

of any type. A map of strings may be indexed with ints, after

the fashion of an array of strings. Here I'm using strings to

index other strings.

-----------------------------------------------------------------\*/

#include <stdio.h>

#include <iostream>

#include <map>

using namespace std;

int main (int argc, char \*argv[], char \*\*env)

{

std::ios::sync\_with\_stdio() ;

std::map<string, string> nodemap ;

/\*

\* this is the older method

\* of loading a map. Note that elements are

\* not "overwritten": \*/

nodemap.insert(map<string, string>::value\_type("hello", "HELLO")) ;

nodemap.insert(map<string, string>::value\_type("goodbye", "PROSCHAI")) ;

nodemap.insert(map<string, string>::value\_type("goodbye", "GOODBYE")) ; // no effect

nodemap.insert(map<string, string>::value\_type("San Diego", "CALIFORNIA")) ;

nodemap.insert(map<string, string>::value\_type("Las Vegas", "NEVADA")) ;

nodemap.insert(map<string, string>::value\_type("Seattle", "WASHINGTON")) ;

nodemap.insert(map<string, string>::value\_type("San Francisco", "CALIFORNIA")) ;

nodemap.insert(map<string, string>::value\_type(" ", "BLANK")) ;

/\*

\* an "iterator" is a specially designed index used to

\* grovel through the map, like an int is used to

\* index an array.

\* First, define an interator to the PRECISE KIND OF

\* OBJECT it iterates over, then use it:

\*/

std::map<string, string>::iterator iter ;

for ( iter = nodemap.begin() ; iter != nodemap.end() ; iter++)

{

std::cout << (\*iter).first << " --> " ;

std::cout << (\*iter).second << std::endl ;

} // For() ends.

} // Main() ends.

/\*-----------------------------------------------------------------

\* map2.cpp - demo the map template

\* Mon Jan 10 14:00:55 2005 TEK

A "map" is a kind of associative array which means that the

map "values" are found via an indexing "key". They may be

of any type. A map of strings may be indexed with ints, after

the fashion of an array of strings. Here I'm using strings to

index other strings.

-----------------------------------------------------------------\*/

#include <stdio.h>

#include <iostream>

#include <map>

using namespace std;

int main (int argc, char \*argv[], char \*\*env)

{

std::ios::sync\_with\_stdio() ;

std::map<string, string> nodemap ;

/\*

\* this is the simpler way of loading a map:

\*/

nodemap["hello" ] = "Hello" ;

nodemap["goodbye" ] = "PROSCHAI" ;

nodemap["goodbye" ] = "GOODBYE" ;

nodemap["San Diego" ] = "CALIFORNIA" ;

nodemap["Las Vegas" ] = "NEVADA" ;

nodemap["San Francisco"] = "CALIFORNIA" ;

nodemap["Seattle" ] = "WASHINGTON" ;

nodemap[" " ] = "BLANK" ;

nodemap["Together" ] = "DROOG DROOGA" ;

nodemap["Sail" ] = "PAROOS" ;

nodemap["Seagull" ] = "CHAIKA" ;

/\*

\* this is the much more involved way:

\*/

//nodemap.insert(map<string, string>::value\_type("hello", "HELLO")) ;

std::map<string, string>::iterator iter ;

for ( iter = nodemap.begin() ; iter != nodemap.end() ; iter++)

{

std::cout << (\*iter).first << " --> " ;

std::cout << (\*iter).second << std::endl ;

}

}

/\*-----------------------------------------------------------------

\* map3.cpp - demo the map template

\* Mon Dec 4 15:18:23 PST 2006 TEK

A "map" is a kind of associative array which means that the

map "values" are found via an indexing "key". They may be

of any type. A map of strings may be indexed with ints, after

the fashion of an array of strings. Here I'm using strings to

index other strings.

-----------------------------------------------------------------\*/

#include <stdio.h>

#include <string.h>

#include <stdlib.h>

#include <unistd.h>

#include <iostream>

#include <map>

using namespace std;

int main (int argc, char \*argv[], char \*\*env)

{

FILE \* ip = fopen("russ.eng.dict", "r") ;

char one[1024] , two[1024] ; // char arrays named for map fields just for clarity

if (!ip) { perror ("can't open russ.dict\n") ; exit(-1) ; }

std::map<string, string> nodemap ;

/\*

\* this is the simpler way of loading a map:

\*/

while (!feof(ip))

{

fscanf(ip,"%s", one) ;

fgets(two, sizeof(two), ip) ;

two[strlen(two)-1] = 0 ;

nodemap[one] = two ;

}

fclose(ip) ;

std::map<string, string>::iterator iter ;

// for ( iter = nodemap.begin() ; iter != nodemap.end() ; iter++)

// {

// std::cout << (\*iter).first << " --> " ;

// std::cout << (\*iter).second << std::endl ;

// }

ip = fopen(argv[1], "r") ;

if (!ip) ip = stdin ;

while (!feof(ip))

{

fscanf(ip, "%s", one) ;

for ( iter = nodemap.begin() ; iter != nodemap.end() ; iter++)

{

if (0 == strcmp(one , (\*iter).first.c\_str() ))

printf("%s ", (\*iter).second.c\_str() ) ;

}

} // while input text remains

}// main

/\*-----------------------------------------------------------------

\* wordcount.cpp - sort all words by length into hashes

\* and count each one.

A "map" is a kind of associative array which means that the

map "values" are found via an indexing "key". They may be

of any type. A map of strings may be indexed with ints, after

the fashion of an array of strings. Here I'm using strings to

index ints.

-----------------------------------------------------------------\*/

#include <stdio.h>

#include <iostream>

#include <iomanip>

#include <string.h>

#include <map>

using namespace std;

int main (int argc, char \*argv[], char \*\*env)

{

std:: map<string, int> count ;

unsigned int i, j ;

string s ;

cin >> s ; // read first word

while (!cin.eof())

{

for (i = 0 ; i < s.length() ; i++) // strip junk out of string

{

s[i] = tolower(s[i]) ;

if (!isalpha(s[i]))

{

s.erase(i,1) ;

i-- ; // back up and test the next character, don't skip it!!

}

}

count[s]++ ; //use word as index, count it with int: <string,int>

cin >> s ;

} //while

// input done, begin output

std::map<string, int>::iterator iter ; // iterator steps through maps (like pointer)

for ( iter = count.begin() ; ++iter != count.end() ; )

{

//++iter ; // skip count of null chars in map

cout << setw(18) << std::left << (\*iter).first << "\t" ;

cout << setw(5) << std::right <<(\*iter).second << endl ;

} // for iter

} // main

/\*-----------------------------------------------------------------

\* wordlengthcensus.cpp - sort all words by length into hashes

\* and count each one.

A "map" is a kind of associative array which means that the

map "values" are found via an indexing "key". They may be

of any type. A map of strings may be indexed with ints, after

the fashion of an array of strings. Here I'm using strings to

index ints.

-----------------------------------------------------------------\*/

#include <stdio.h>

#include <iostream>

#include <map>

using namespace std;

int main (int argc, char \*argv[], char \*\*env)

{

const int SIZE = 50 ;

std:: map<string, int> count[SIZE] ;

unsigned int len, i, j ;

string s ;

cin >> s ;

while (!cin.eof())

{

for (i = 0 ; i < s.length() ; i++) // strip junk out of string

{

s[i] = tolower(s[i]) ;

if (!isalpha(s[i]))

{

s.erase(i,1) ;

i-- ; // back up and test the next character, don't skip it!!

}

}

len = s.length() ;

if (len > 0 && len < SIZE) // store in array of maps/count how many seen

count[len][s]++ ;

cin >> s ;

} //while

std::map<string, int>::iterator iter ; // iterator steps through maps (like pointer)

for (j = 0 ; j < SIZE ; j++)

{

for ( iter = count[j].begin() ; iter != count[j].end() ; iter++)

{

cout << (\*iter).first << "\t\t" ;

cout << (\*iter).second << endl ;

} // for iter

} // for j

} // main