

**CIS \* 2650**  
**Winter 2004**

W. Elazmeh      X. Li

**Lecture Notes**

Compiled by:  
**Dave Heppenstall**  
**dheppens@uoguelph.ca**

## Lecture 01 – Introduction

05 Jan 2004

### Course Web Page:

[www.cis.uoguelph.ca/~welazmeh/teaching/cis265/winter04/](http://www.cis.uoguelph.ca/~welazmeh/teaching/cis265/winter04/)

### Course Methodology:

- I am interested in the reasoning and not the answer.
- Just because it works does not mean it is correct.

### Recommended Book:

The Joy of C (3rd edition)

L.H. Miller and A.E. Quilici, Wiley, 1997.

---

## Lecture 02 – Introduction to C

07 Jan 2004

- **Algorithm:** = a finite set of precise instructions for performing a computation for solving a problem.
- An algorithm is translated into a programming language to make up a computer program.
- Programming model solutions using different paradigms
- A programming paradigm explains the model by which algorithms and their entities are implemented.
  
- Keep an algorithm general, do not make it dependant on any particular language.
- You can develop an algorithm and a solution independent of a computer altogether.
  
- Extract four items from any problem:
  - + What is the data? What do you know? (Input)
  - + What is missing? What is the result? What do you need to find out? (Output)
  - + What are the constraints? Things that you cannot accept or parameters that you must abide by.
  - + Assumptions. Things which are safe to assume for the particular problem.  
Example, an average of positive numbers will itself be positive.
  
- **Analysis:** Read the problem statement
- **Solution:** Create a solution in pseudocode
- **Program:** Program the algorithm into code.

## **Programming Paradigms**

- Programming languages implement at least one programming paradigm
- A pure programming language implements exactly one programming paradigm (C, Java).
- Programming Paradigms include:
  1. Procedural or imperative (Pascal, C, C++)
  2. Object oriented (C++, Java)
  3. Functional (Lisp, ML)
  4. Logic programming or declarative (Prolog).

### Procedural Paradigm

- The algorithm is abstracted into tasks
- Tasks are decomposed into sub-tasks
- Each sub-task is implemented as a procedure which may call another procedure.
- When a procedure calls another, information is exchanged by passing parameters.
- A good decomposition identifies general tasks which can be used in many contexts.
- Examples: C functions, Pascal procedures or functions.
- static means hardcoded. They are very slow and inefficient. Avoid static methods. However, the entire program itself IS static.
- global variables are NOT a good idea. Avoid these at any and all costs.

### Functional Paradigm

- Tasks are decomposed into sub-tasks
- Sub-tasks are implemented as function which may accept parameters but must always return a value.
- Information is not saved in variables but propagated through a series of function calls.
- A function is a first class object and can be manipulated as if it were data.
- Most popular in Artificial Intelligence.

### Logic Programming

- Programs are defined by rules and facts
- The rules are formulated as HORN CLAUSES and define inductive (recursive) steps of the algorithm.
- The facts stop the recursion.
- Require complicated execution (matching and backtracking)
- Applications are specified rather than implemented (declarative approach)
- Also very popular in Artificial Intelligence.

**Program correctness: Are you getting the right answer because it is correct? Or you somehow got the correct answer by some other means.**

### Object Oriented Paradigm

- An object is the basic unit of decomposition
- An object contains both data and functionality.
- A class is a part of hierarchy

- Every object is an instance of a class
- Instance-class relationship is analogous to variable-type relationship.
- A class defines common attributes, and associations but does not instantiate them.

**Isomorphism:** Objects appear to all have the same action, but are carried out differently.

(Inheritance down, isomorphism up.)

### Translation into machine Code

- To execute programs, we must translate them into machine code.
- There are two ways:
  1. **Compilation:** translate all program statements together and generate one machine code executable file.
  2. **Interpretation:** translate each program statement into machine code, run it then do the next statement and so on.
- C/C++ are compiled, but Java is compiled into Byte Code which is interpreted by the Java Virtual Machine.

### The C Language

- Flexible, supports:
  1. Low level programming (Unix OS, device drivers)
  2. High level programming, (user defined types, functions, parameter passing).
- Concise but small. Originally K&R, ANSI later.
- Procedural where decomposition is implemented as functions which accept parameters and return values

## Lecture 03 – Introduction to C (Part II)

09 Jan 2004

### The Architecture of C Programs

- Supports multi-component development by one or more programmers (separate compilation)
- Files: header, source, object, and libraries. File extensions are important (case sensitive), the extensions are .h, .c, .o, .a (or .lib) respectively

### C Header Files (.h)

1. C Preprocessor directives (macros, definitions)
  2. Declarations of global variables
  3. References to external variables
  4. Function prototypes
- Processes as text by the C preprocessor
  - Should NOT contain implementation code

## C Source Files (.c)

1. Global variables (within this file) declaration
2. Implementations of functions
  - Compiled into object code file (.o)
  - There should be a separate file for the function main(), and another for the function used by main()

## C Object Files (.o)

- Implement reusable generic software components
- Matching .h and .c are compiled into .o
- Can be compiled with any main program
- Contain the machine code for functions implemented in the .c only
- .o files are linked together (one must contain the main function) to generate the executable

## Compiling C Programs

- Preprocessing: text editing
- Compiling: generating .o files (separately)
- Linking: linking several software modules (.o files) to generate an executable
- Archiving: adding reusable generic software components (.o files) into the compiler's executable libraries.
- Make Utility: **makefiles** allow the automation of separate compilation process

## The C Preprocessor

- Process the code by editing the text which can:
  1. Expand definitions
  2. Expand macros
  3. Include or exclude header files (by text copy)
  4. Include or exclude source code (using definitions)
- Driven by the preprocessor directives (statements starting with #)
- Can be found in /lib/cpp and is called automatically by the compiler

## The C Compiler

- /usr/bin/gcc is the GNU C compiler on UNIX (which we will be using) and when called, it calls the preprocessor automatically
- “gcc -c” compiles .c files into .o object files
- Example: assume myFile.h and myFile.c “Gcc -c myFile.c” generates myFile.o

## The C Linker

- Combines libraries and .o into executables
- /usr/bin/ld is the linker on UNIX
- Example: ld stack.o main.o -o -lmath
  - The above generates by linking the machine code of functions implemented in stack.o and called in main.o along with math functions available in the math library
  - “-o” flag tells the linker to use the next argument (main) as the name of the executable file

- “-l” flag tells the linker to use the standard library (libmath.a)

### Creating C Libraries

- Example: create math library which includes the object files calc.o and matrix.o
  - demo.c contains main program code and includes stack.h
  - stack.h contains prototypes for functions which implement Stacks
  - stack.c contains code which implements functions declared in stack.h
- Compile the above program into one executable

### The Make Utility

- Determines which component of the software needs to be compiled
- To use it, a programmer must write a makefile to describe the relationships between the software components and to provide commands to perform the compilation
- Once a make file is written, use the command “make” to update the executable file
- The make used the makefile and last modification time to determine which source files need to be recompiled

### Makefile for the Stack Example...

CC = gcc

Specify compiler options:

CFLAGS = -Wall -g -ansi

all: stack

stack.o: stack.c

<tab> \$(CC) \$(CFLAGS) stack.c -c

demo.o: demo.c

<tab> \$(CC) \$(CFLAGS) demo.c -c

stack: demo.o stack.o

<tab> \$(CC) \$(CFLAGS) stack.o demo.o -o stack

clean:

<tab> echo rm -f \*.o core

### C Examples

- A Simple C program
- Compute the average of x, y, and z
- Print the command line arguments of the program
- The above simple program with command line arguments

These are...

- statements which start with ‘#’
- tell the preprocessor how to edit the source file
- examples...

```
#include <filename>
#define constant <value>
#define macro_name
#ifdef ... #endif
#endif ... #endif
```

---

**Lectures 05 and 06 – An Example Program****14 Jan 2004**

This is an example C program which will draw a box and illustrate some of the problems which might arise in coding it...

```
include <stdio.h>
define TAB_SIZE 10

void printTab(void);
void printFullLine(int theWidth, char theChar);
void printBorderLines(int theWidth, int theHeight, char theChar);
void printOneBorderLine(int theWidth, char theChar);

int main(void){
    char aChar;
    int width, height, index0, index1, index2;

    /* Greetings */
    printf("Good day\n");

    /* Read in the width */
    printf("Enter the width of the rectangle.");
    scanf("%d", &width);

    /* Read in the height */
    printf("Enter the height of the rectangle.");
    scanf("%d", &height);

    /* Clear the \n character from the input queue */
    scanf("%d", &aChar);

    /* Read in the desired drawing character */
    printf("Enter the character you want to draw with.");
    scanf("%c", &aChar);

    /* Print the top line */
    printFullLine(width, aChar);

    /* Print the middle lines */
```

```

    printBorderLines(width,height,aChar);

    /* Print the bottom line */
    printFullLine(width,aChar);

    printf("Good bye...\n");

    return 0;
}

void printTab(void){
    int index;
    for (index=0;index<TAB_SIZE;index++)
        printf("%c",' ');
}

void printFullLine(int theWidth, char theChar) {
    int index;
    printTab();
    for(index=0; index<theWidth;index++)
        printf("%c", theChar);
    printf("\n");
}

void printBorderLines(int theWidth, int theHeight, char theChar) {
    int index;
    for(index=0; index<theHeight-2; index++){
        printTab();
        printOneBorderLine(theWidth, theChar);
    }
}

void printOneBorderLine(int theWidth, char theChar){
    int index;
    printf("%c", theChar);
    for(index=0; index<theWidth-2;index++)
        printf("%c", ' ');
    printf("%c", theChar);
    printf("\n");
}

```

- Do not use a *backhook*! Do not try to access a variable defined in the main program from a sub function.
- Don't forget to think about boundary conditions, ie: what your program will do with variable data that doesn't quite make sense.
- A dot h (header) file contains all the function signatures and global constants the program will use.

### The Make file

```

CC = gcc
CFLAGS = -Wall -g -ansi

all: draw

```

```
draw3.o: draw3.c draw3.h
      $(CC) $(CFLAGS) draw3.c -c
```

```
draw: draw3.o
```

---

## Lecture 07 – More C Basics

19 Jan 2004

*Questions? Concerns? Freaking out?*

*Get in touch with the CIS Success Team:*

Greg Klotz  
[gklotz@uoguelph.ca](mailto:gklotz@uoguelph.ca)

Jennifer Lyon  
[jlyon@uoguelph.ca](mailto:jlyon@uoguelph.ca)

---

### C Literals

- Strings are character arrays
  - NUL-terminated
  - enclosed in double quotes
  - `char str[] = "hello world"`
  - `str` is the memory location of the first character
  - automatically NUL terminated
- floating points (float double)

### Scope of Identifiers

- C supports free functions
- an identifier is a name of a variable, function, class or method (C++)
- identifiers are case sensitive
- variables (identifiers) may be:
  - local to a function (local or parameter)
  - local to a file (global)
  - shared with other files **extern**
- every identifier belongs to a scope.
- scope defines where the identifier can be used.
- scope is a logical block where identifiers are visible.
- a block is indicated by `{ ... }`.
- each function has its own scope.
- This means that a variable lives in between the two braces, once you exit the brace, the variable is invalid. You can declare variables inside loops, etc.

- In C, you can't mix declarations and executions together. You must do all the declarations first and all the executions next.
- You can "double-up" braces to prevent having this problem.
- data is always passed from one scope to another by parameter passing techniques. Do NOT use global variables in place of these techniques.

### Declaration of Variables

```
<type> <identifier> [=<value>] [, <identifier> [=<expression>] ...];
```

#### • Examples:

```
int a;
unsigned int a, b = 1, c = 0;
float a, b, c;
char a = 'a', b = 'b';
char str[10];
char str[] = "Hello world!";
```

- You must initialize a variable before you start using it. The value is not nothing, not null, not anything, just GARBAGE. You have NO IDEA what is in the variable when you declare it.
- In C, there is nothing to stop you from accessing memory that you shouldn't have access to!
- A segmentation fault occurs when you are accessing memory you shouldn't be accessing for any particular reason. The OS doesn't know, all it knows is that you messed up somewhere.

### Declaration of Functions

```
<return type> <identifier> ([<parameters>[,] ... ]);
```

- Also called function prototype or signature.
- usually written in the file where the function is being used or
- written in a header file which is included into the source file where the function is being used.
- see examples/intro\_c for examples.

### Variables and Functions

- upon entering a scope (or a function) declared variables are allocated automatically (memory is grabbed).
- upon exiting a scope or function, automatically allocated variables are deallocated (their memory is thrown away).
- **static** variables preserve their memory and values between calls (ie, they are not deallocated after exiting the scope).
- Dynamically allocated memory is a completely different shade of disaster.
- **global** variables are allocated in the scope of the file in which they are declared and are visible in any functions defined in that file.
- use extern to share variables between files.

- Every program has exactly one stack. A stack "frame" will give you the scope and will be discarded when it is complete.

### Outputting variables with printf()

```
printf("<preformatted string>" [,<variable> ...]);
```

- arguments to printf() may be:
  - text and escape characters
  - text, tags and variables
  - strings which have been NUL-terminated

- some tags:

%d for decimal number

%x for hexadecimal number

%p for a pointer

### Inputting Variables with scanf();

```
scanf("<formatted string>" [,<variable> ...]);
```

- arguments to scanf() may be:
  - tags and variables
  - strings
  - compare with scanf("%c", &name[4]); NOTE: No use of & and no white spaces.
  - compare with gets()
- same tags as printf()

### Logical operators and Expressions

0	FALSE
NOT 0	TRUE
&&	AND
	OR
==	EQUALITY
!=	INEQUALITY
> < >= <=	COMPARISONS

**Selection**

```
if (<logical expression #>) {
    /* statements for T */
}
else {
    /* statements for F */
}
```

- When using ifs and nested ifs, ensure which logical expression is being tested and which are independent and which are based on some other expression.

**Switch Statement**

```
switch(<identifier>) {
case <value 1>: /*statements */; break;
case <value 2>: /*statements */; break;
case <value 3>: /*statements */; break;
...
default: /*statements */;
```

- The break statements are required to stop the process from "dropping through" every subsequent statement.

**Loops**

```
while(<logical expression>) {
    /* statements */;
    ...
}

do {
    /* statements */;
    ...
} while (<logical expression>);

for ([<expr1>] ; [<expr2>] ; [<expr3>]) {
    /* statements in the loop */
    ...
}
```

*expr1 executes before starting first iteration.*  
*expr2 executes before starting any iteration.*  
*expr3 executes after the ending of each iteration.*

- A loop which will never be able to achieve its exit condition is called an infinite loop.  
Check your expressions before executing.
- There must be some kind of progress towards the stopping condition in the loop.

## Functions

- C has no procedures, only functions
- functions must return a value while procedures may not.
- execution starts at the function main() which may take parameters passed from the command line.
- functions may return the type void.
- function definitions cannot be nested.
- functions may be called recursively.
- function definition general syntax:

```
<type> <identifier> ([<parameter., ... ] ) {
    /* declare local variables */
    /* body of the function */
}
```

## Function Calls

1. allocates memory for parameter passing.
2. copies all parameters into allocated memory.
3. allocates memory for the returned value
4. transfers execution control to callee

*callee starts execution*

1. saves information needed by caller to resume execution afterwards.
2. allocates memory for local variables.
3. executes body of callee (may contain more function calls which may be recursive).
4. copies the returned value into memory allocated by caller
5. transfers execution control back to caller (using saved information) and deallocates memory allocated for parameter passing.

*callee terminates*

5. caller may (or not) retrieve returned value
6. caller deallocates memory of returned value
7. resume execution

For example...

```
int f(int a);
void main() {
    int x;
```

```

int a = 10;

x = f(a);
x = f(f(a));
}

```

- The program begins evaluating at the most nested expression and travels upward the recursive function path.
- There is only one way to pass parameters in C: by copying the data directly from one variable to another. However, knowing the memory location would render this irrelevant.

## Lecture 09 – More C Basics (Part III)

23 Jan 2004

### Statically Allocated Arrays

- Hold elements which all have the same type.
- stored in a contiguous block of memory.
- by definition, the name of the array is a pointer to the first element of the array.
- accessing an element is done by supplying an index (int) and using [].
- are always indexed from 0 to length -1 .
- cannot manipulate all elements in one step. Must manipulate the array one element at a time.

- general declaration syntax:

```
<type> <identifier> [d1][d1] ... [d1]
```

- Examples:

```
int num_list[10]; for first element, use: num_list[0]
```

```
char name[20]; for ith element, use char[i]
```

How much memory does this use?

```
int num_list [ 10 ]
```

```
4          10
```

- On the system stack, you will get a contiguous block of memory. This block is 4 bytes wide (because of int) and 10 bytes long.
- Do not have the program accessing data beyond this array block because you have NO IDEA what is there!! You will be LUCKY to get a segmentation fault.
- The actual name of the array is the location of the first element.
- Moving from one element to the next requires the program to skip by the size of the type.
- sizeof( ) will return the size of an identifier.
- the size of the block of memory must be known at compile time.

```
#define Size 100
int num_list [ size ]
int my.class [size][size]
```

- Count the number of entires you retrieve to prevent an over or under run error.

### Statically Allocated Multidimensional Arrays

- Two-dimensional: int matrix[rows][cols]
  - a column is a done-dimensional array of int.
  - a ow is an array of row values
  - matrix [i][j] is an int element.
  - for matrix(0,0) use matrix [0,0]
  - for matrix(i,j) use matrix [i-1,j-1]
  - matrix[0] is int array of the first row values.
  - matrix[i] is int array of the ith row.
- Three dimensional int space[x][y][z]
  - x, y and z are the dimensions of the array
  - space [i] is a two dimensional array of int.
  - space [i][j] is one-dimensional array of int
  - space [i][j][k] is an int element
- binding is the process of binding a type to an array element.
- if you ever reference the array all by itself, you will get the first element.

Example:

```
printf("%d\n", sizeof(x));
```

This will actually give you the size of the block, 40.

Check to see what happens if you do this with a string. It will be on the exam.

Initialization of Statically Allocated Arrays

- statically allocated arrays can be declared and initialized in one statement.

- examples:

```
- int numlist[8] = {1,2,3,4,5,6,7,8};  
- char name[12] = {'B','e','n',' ','  
  ','B','o','s','t','r','u','m','\0'};  
- int matrix[3][3] = { {11,12,13} , {11,12,13} , {11,12,13} };
```

- we may also omit a dimension:

```
- int num_list[] = {1,2,3,4,5,6,7,8};  
- char name[] = {'B','e','n',' ',' ','B','o','s','t','r','u','m','\0'};  
- compare with char name[] = "Ben Bostrum";  
- int matrix[][3] = { {11,12,13} , {11,12,13} , {11,12,13} };
```

- arrays of strings!

*Consult 2D\_arrays\_1.c for examples*

- When you pass in an array to a function, you must specify n-1 dimensions or the n-1th dimension.
- Use an ampersand to calculate the location in memory of a value.

```
int y = 10;  
y = f(y);  
  
int f(int a) {  
  int x;  
  x = a;  
  return a;  
}
```

- Three steps when you have a function call
  - worry about the parameters
  - worry about the calculations within the function.
  - worry about the data that goes out.

- If we were to write...

```
int *f(int a) {  
  ...  
  return &a;  
}
```

...Then we are passing in a memory location and returning the memory location back again.

- variables of different types can be grouped into Structures (like records in Pascal).
- each of the variables is called a field.
- fields are accessed by using the dot operator.
- the following declares a structure Person:

```
struct Person {
    char name[50];
    int age;
} troy, mary;

troy.name = "Troy Bayliss";
troy.age = 534;
mary.name = "Mary Moor";
mary.age = 34;
```

- The size of the structure is 54 bytes: 50 for the name and 4 for the integer.
- You cannot declare any more Person s after the semicolon after mary. It is not a generalized or known type.

### User Defined Structures

- use typedef to create new types.
- the following creates a new type Person:

```
typedef struct {
    char name[50];
    int age;
} Person;
```

- Now we can create new Person variables:

```
Person troy, mary;
troy.name = "Troy Bayless";
troy.age = 534;
mary.name = "Mary Moor";
mary.age = 34;

Person Joe = {"Joe, 100}
```

- We can also create an array of person. You can also nest structures within structures.
- You cannot compare variables of user defined types because you can only compare the primitive data types contained within the structure.
- It does not make sense to see if `mary == troy`, UNLESS you are parameter passing and you want to see if you are comparing a memory location to the original data type. (Ie, deep or shallow copy testing).

## Lecture 12 – Parameters in Structures

30 Jan 2004

- Write a C function which will print out to screen 1 record (structure) of Person
    - We need to print out a string and an integer

```
void printPerson(Person aPerson) {
    printf("The name is: %s\n", aPerson.name);
    printf("The age is: %d\n", aPerson.age);
}
```
    - To use this function simply pass a Person to the function `printPerson`
    - `printPerson` would be inside `person.c`
    - the definitions would be in `person.h`
    - We would use the definition and function in `main.c`
    - To compile, use a makefile and compile the two `.c`'s together
      - Remember that the structure passed to the function is a static copy
    - If it is changed in the function, it is only changed in the local function!
    - We can also use arrays, but we have to change `printPerson` to accept an array, or create a new function
    - Pass the array, and print each record separately
- 

## Lecture 13 – Arrays of Structures

2 Feb 2004

- Consider the following C code fragment:

```
typedef struct {
    char name[50];
    int age;
} Person;

Person people[100];
Person x = {"William", 35};
```

- Write a C function which returns the index position of the record whose name equals to "William".

```
int findPerson(Person arrPerson[], char name[]) {
    int result = -1;
    int index = 0;

    while( (index < SIZE) && (result == -1) ) {
        if (strcmp(arrPerson[index].name, name) == 0)
            result = index;
        index++;
    }
}
```

```
    }  
    return result;  
}
```

- Before you return anything, document somewhere what the convention of the function is.
- ie: What will it return if nothing was found? Negative one. That is a convention. Document it!

---

## Lecture 14 and 15 – Introduction to Pointers

4 Feb 2004

### What is a Pointer?

- Simply stated, a pointer is an address.
- A running program consists of three parts: execution stack, code, and data. They are stored in main memory.
- Each cell of the main memory has an address. A pointer variable is a variable whose value is either NULL (0) or a legal address.

**Memory map:** A map of how many things you need to count. Bytes? Bits? Intersections?

**Legal Address:** Do not try to obtain data from a pointer which points nowhere. Do not attempt to dereference a NULL.

- A problem that could happen is if you try to reference memory which is outside of your allocated program memory. Illegal address, but valid.

### Why Pointers?

- Pointers are used in programs to access memory and manipulate addresses.
- To get the effect of call-by-reference in C, we must use pointers in the parameter list of a function definition and pass addresses of variables as arguments in the function call.
- To hold the memory location of a chunk of dynamically allocated memory block.

```
int **pp;
```

This is a pointer to int \*, which itself is a pointer to int.

```
Object x;
```

x doesn't exist yet.

```
x = new Object;
```

now x exists.

- A pointer is always 4 bytes in size, regardless of what type of variable it points to.  
Example: A float is 8 bytes, but a pointer to that float is 4 bytes.

#### How to declare pointers:

- a pointer must be defined to point to some type of variable
- following a proper definition, it cannot be used to point to any other type of variable or it will result in a "type incompatibility" error.

- Declaration examples:

```
int *p, **p;           //How to read?
char *cp;
float *fp1, *fp2, *fp3;
```

---

## THE MIDTERM

### **Content:**

Very heavy on structures, arrays, parameter passing, debugging, problem decomposition (Take a bigger problem and break it down into terms of smaller problems), memory maps (what they look like), arrays of structures, structures of arrays, manipulating and indexing arrays. Will not go too crazy on pointers. I am hoping to not have coding questions.

### **Format:**

- it is going to be a big one.
  - T/F
  - Multiple choice
  - Debugging question (code, run, intended output - what is the problem?)
  - Identifying the problem is worth marks, not as much as the WHY and the FIX.
- One humongous question: A big piece of code, the specs of each component is already designed for you. Follow the code, part by part and fill in the giant blanks. You will also be provided with a sample run of the entire system compiled together.
- Every answer is short. The size of the space is very generous. If you are writing more than five lines, you're doing too much. The space allocated is twice what you should need.

### **Notes:**

- If you get stuck, put your hand up. If you're asking a question that you should know, don't worry about it. Ask anyway.
- I would prefer that you use a pen and not a pencil. If you write in pencil, you cannot ask for a remark.
- No calculators permitted at all.

## Lecture 16 – More on Pointers

9 Feb 2004

```
int x = 5, y = 6, z;  
int *px, *py, *pz;
```

```
px = &x;  
py = &y;  
pz = &z;
```

```
*pz = *px + *py; // z = x + y (11)  
y = 2 * *px + y; // y = ?
```

### • Exercises:

```
int i = 3, j = 5, *p = &i, *q = &j, *r;  
double x;
```

<u>Expression</u>	<u>Equivalence</u>	<u>Value</u>
p == &1	p == (&i)	1
p = i + 7	p = (i + 7)	<i>illegal</i>
**&p	*(&p)	3
r = &x	r = (&x)	<i>illegal</i>
7 **p/*q+7	((7*(*p)))/(*q)+7	11
*(r = &j)*=*p	*(r=(&j))* = (*p	15

### • Example:

```
i = 0;  
while(i < N) {  
    x = msg[i++];  
}  
x= Arr[i++]++;
```

### **Pointers to void:**

#### • void \* is a generic pointer type (in ANSI C)

```
int x = 6, *px = &x;  
void * vp;  
double *qx;
```

```
qx = px; //Illegal  
vp = px; //ok
```

## Memory Map:

```
int x = 10;           10
```

```
char ch = 'a';      a
```

```
void *p;
```

```
p = &x;
```

```
*p = something;
```

```
something = *p;
```

- The trick is to cast the pointer before you use it.
- You CAN force the double pointer to point to the int, but you CANNOT force the double pointer to equal the integer pointer.

```
void swap(int *p, int *q) {  
    int tmp;  
    tmp = *p;  
    *p = *q;  
    *q = tmp;  
}
```

```
int main(void) {  
    int a = 3, b = 7;  
    swap(&a, &b);  
}
```

### before swap:

```
p --> a=3
```

```
q --> b=7
```

### after swap:

```
p --> a=7
```

```
q --> b=3
```

```
a[3]
```

```
b[7]
```

```
--> swap (&a, &b)
```

```
-----  
swap [p] [q]
```

```
p[ ]  
q[ ]
```

```
*p = 10;
```

```
tmp = p;
```

```
p = q;
```

```
q = tmp;
```

- Never reference a pointer without testing for NULL.

```
if (p != NULL) {  
    tmp = *p;  
    *p = *q;  
    *q = tmp;  
}
```

---

## **Lecture 17 – Even more on Pointers**

**11 Feb 2004**

- Return multi-values through pointers:

```
void minmax(int table[], int n, int *minp, int *maxp) {  
    int i, min = INT_MIN, max = INT_MAX;  
    for (i = 0; i < n; i++) {  
        if (table[i] < min)  
            min = table[i];  
        else if (table[i] > max)  
            max = table[i];  
    }  
    *minp = min;  
    *maxp = max;  
}
```

```
int main() {  
    int table[] = {10, 15, 38, 27, -21, 88, 69};  
    int min, max;  
    minmax(table, 7, &min, &max);  
    // print min and max  
}
```

- There is a problem with this code. The code does not check to see if it NULL first. Just test to see if it is pointing SOMEWHERE and not NOWHERE.

Given:

```
int table[]
```

- Know the difference between:

```
table
```

*The address of the first block of the contiguous memory allocated for the array*

```
&table
```

*The address of the thing holding the address of the integers that we are looking for.*

```
*table
```

*Stars do not always mean dereference, sometimes they mean type - check if it is a declaration statement or an execution statement.*

Array and Pointers:

```
#define N 100
int a[N], *pa, sum, i;
// suppose array a has been initialized
pa = a;
pa = &a[0];    //these 2 are the same
pa = a + 10;
pa = &a[10];   //these 2 are the same
sum = 0;

for (pa = a; pa < &a[N]; pa++)
    sum+= *pa;
sum = 0;
for (i = 0; i < N; i++)
    sum += a[i];
```

NEVER WRITE THIS EVER: `a == &a`  
UNLESS YOU REALLY KNOW WHAT YOU'RE DOING.

```
a == &a[0];
```

This is equivalent.

```
a == &a;
```

This is the ugly one.

```
pa != &pa;
```

### **Efficient code - using pointers**

```
void stat(double a[], int n, double *p_ave, double *p_max,  
double *p_sum) {  
    int i;  
    *p_max = *p_sum = a[0];  
    for (i = 1; i < n; i++) {  
        *p_sum += a[i];  
        if (*p_max < a[i])  
            *p_max = a[i];  
    }  
    *p_ave = *p_sum/(double) n;  
}
```

---

**Lecture 18 – THE MIDTERM**

**13 Feb 2004**

- No notes today.

**Review of Variable:**

```
int v = 0; /* declared at some place in your program */
```

*What can you see from the declaration?*

storage class : auto, register, static, extern  
type : value domain  
value : current value E, value domain  
name : symbolic identifier  
location : memory address  
size : how many bytes it occupies  
scope : where it can be accessed

**Storage class and Scope:**

**Auto:** declared inside a block, exists only when the block is entered and disappears when execution leaves the block.

```
{ int x, y; ... }  
  x and y alive  
  Accessible in  
  this block
```

**Static:** accessible in the block where it is declared, exists and retains its value in whole program cycle.

<pre>void f(){   static int x = 0;   printf("%d\n", x++); }</pre>		<pre>void f(){   int x = 0;   printf("%d\n", x++); }</pre>
<pre>int main(){   f(); // 0   f(); // 1   f(); // 2 }</pre>		<pre>int main(){   f(); // 0   f(); // 0   f(); // 0 }</pre>

**Extern:** accessible and exists in whole program file and program cycle. Define a global variable exactly once and use external declarations everywhere else.

<pre>/* f1.c */ int x; void f(){   x++; } void g(){   x++; }</pre>		<pre>/* f2.c */ extern int x; void h(){   x++; }</pre>
--	--	--

**Register:** frequently used variables for efficiency purpose.

Restrictions:

- (1) can not take the address of a register variable,
- (2) can not declare global register variables,
- (3) a register variable must fit into a single machine word,
- (4) the compiler may ignore register declaration.

```
/* search the given table to find the given key;
return the index if found or -1 otherwise */

int table_search(int a[], register int n, register int key){
    register int j;
    for (j = 0; j < n && a[j] != key; j++);
    return (j != n) ? j : -1;
}
```

Review of pointer:

```
int x = 4;
int *a = &x;
int **b = &a;
```

x	4	a	addr(x)	b	addr(a)
&x	addr(x)	&a	addr(a)	&b	addr(b)
*x	illegal	*a	4	*b	addr(x)
*(&x)	4	**a	illegal	**b	4
				***b	illegal

```
*b == a == &x
**b == *a == x
```

Relationship of -> and .

```
typedef struct {
    char name[20];
    int grade;
} student;

student s;
student *sp = &s;

s.grade = 97;           // direct structure field selection
sp -> grade = 97;       // indirect structure field selection
(*sp).grade = 97;      // (*sp).grade == sp->grade
(&s) -> grade = 97;     // s.grade == (&s)->grade
```

## Structures and Operators

- `sizeof` operator determines the number of bytes used by a structure.  
`sizeof(s) // using a student variable`  
`sizeof(student) // or using a type`
- Don't assume the size of a structure is the sum of the size of its fields.  
`sizeof(s.name) + sizeof(s.grade) ≠ sizeof(s)`

## Structures and Operators

- Assignment operator: applies to structures which copies the contents of one structure into another.  
`student new_student = s;`
- No operators for comparing structures.  
`(new_student == s) // wrong`

## Runtime Memory Allocation

<b>All external and static variables</b>	Global	System Control
<b>All dynamically allocated variables</b> <i>malloc, realloc, calloc</i>	Heap	User Control
	<i>Free Area...</i>	
<b>All auto variables and function (call control information)</b>	Stack	System Control

### *Example:*

- Suppose we want to design a program for handling student information:

```
typedef struct {  
    char name[20];  
    int grade;  
} student;
```

- *Question:* how to create a table of student records?
  - a) static array: `student stable[MAX_STUDENTS];`
  - b) dynamic: Table? List?

- C requires the number of items in an array to be known at compile time. Too big or too small?
- Dynamical memory allocation allow us to specify an array's size at run time.
- Two important library functions are `malloc`, which allocates space from HEAP, and `free`, which returns the space (allocated by `malloc`) back to HEAP for reuse later.

*Example:*

```
/* allocate and free an array of students, with error check */
#include <stddef.h> // definition of NULL
#include <stdlib.h> // definition for malloc/free

student *table_create(int n){
    student* tp;
    if ((tp = malloc(n*sizeof(student))) != NULL)
        return tp;
    printf("table_create: dynamic allocation failed.\n");
    exit(0);
}

void table_free(student *tp){
    if (tp != NULL) free(tp);
}
```

*Comments:*

- Don't assume `malloc` will always succeed.
- Don't assume the dynamically allocated memory is initialized to zero.
- Don't modify the pointer returned by `malloc`. free only pointers obtained from `malloc`, and don't access the memory after it has been freed.
- Don't forget to free memory which is no longer in use (garbage).

Allocate and free dynamic memory

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

```
void* malloc(size_t n)
```

allocates n bytes and returns a pointer to the allocated memory, the memory is not cleared

```
void* realloc(void* p, size_t n)
```

changes the size of the memory block pointed to by p to n bytes. The contents will be unchanged to the minimum of the old and new sizes.

**void\* calloc(size\_t m, size\_t n)**  
allocates memory for an array of m  
elements of n bytes each, and returns a  
pointer to the array.

**void free(void\* p)**  
frees the memory block pointed to by p.

*Example:* student table

- To create a student table dynamically.
  - (1) do not know how many students,
  - (2) keyboard input
- Requirement: table holds exact number of pointers to student records;

```
/* creates a student record dynamically and returns the pointer to the  
student record */
```

```
student* make_student(char* name, int grade){  
    student* sp;  
    if ((sp = malloc(sizeof(student))) == NULL){  
        printf("make_student: dynamic allocation failed.\n");  
        exit(0);  
    }  
    strcpy(sp->name, name);  
    sp->grade = grade;  
    return sp;  
}  
  
/* copy s to t, suppose t long enough*/  
void my_strcpy(char* t, char* s){ while (*t++ = *s++); }
```

```
#define CHUNK 5  
student** make_table(int* num){  
    int j = 0, maxs = CHUNK, grade;  
    char name[20];  
    student** stp;  
  
    stp = (student**) malloc(maxs*sizeof(student*));  
  
    while (2 == scanf("%s%d\n", name, &grade)){  
        if (j >= maxs){  
            maxs += CHUNK;  
            stp = (student**) realloc(stp,maxs*sizeof(student*));  
        }  
        stp[j++] = make_student(name, grade);  
    }  
  
    if (j < maxs)  
        stp = (student**) realloc(stp, j*sizeof(student*));  
    *num = j;  
  
    return stp;  
}
```

```

int main(){
    student** cis265;
    int num;

    cis265 = make_table(&num);
    printf("The total number of students: %d\n", num);
    // other processing...
}

```

j: counter of students  
maxs: number of entries in the table

- If the table is not big enough, Add another 5 entries;  
Upon complete, if the table has unused entries, then return them.

- Note: we only expand or truncate the pointer array (stp).  
student records are not changed.

### Nested dynamic memory allocation

- *suppose:*

```

typedef struct {
    char *name; // instead of char name[20]
    int grade;
} student;

```

- *then:*

```

student* make_student(char* name, int grade){
    student* sp;
    if ((sp = malloc(sizeof(student))) != NULL && (sp->name = malloc(strlen(name) + 1)) != NULL){
        strcpy(sp->name, name);
        sp->grade = grade;
        return sp;
    }

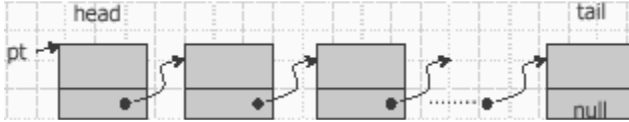
    printf("make_student: dynamic allocation failed.\n");
    exit(0);
}

void free_student(student* sp){
    free(sp->name); // must release name field first
    free(sp);
}

```

### Linked list

- A linked list represents a sequence:
- Every node but one has a predecessor, and every node but one has a successor.



```
/* recursive definition */
typedef struct node {
    int data;                // whatever useful in the node
    struct node* next;     // link to the next node
} node;
```

### *Example: student list*

- data structure:

```
typedef struct student{
    char name[20];
    int grade;
    struct student* next;
} student;
```

- **make\_student function not changed**

```
student* make_list(int* num){
    int j = 0, grade;
    char name[20];

    student *sp = NULL, *ep = NULL;

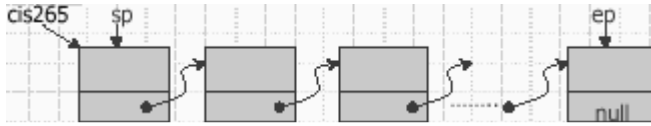
    while (2 == scanf("%s%d\n", name, &grade)){
        if (sp == NULL)
            sp = ep = make_student(name, grade);
        else {
            ep->next = make_student(name, grade);
            ep = ep->next;
        }
        j++;
    }

    if (ep != NULL) ep->next = NULL; // last node of the list
    *num = j;
    return sp;
}
```

```

int main(){
    student* cis265;
    int num;
    cis265 = make_list(&num);
    printf("The total number of students: %d\n", num);
    // other processing
}

```



### Other useful functions

**/\* find the student record wrt name \*/**

```

student* find(student* sp, char* name){
    while (sp && (strcmp(sp->name, name) != 0))
        sp = sp->next;
    return sp;
}

```

**/\* print student list \*/**

```

void print_list(student* sp){
    while (sp){
        printf("Name: %s Grade: %d \n", sp->name, sp->grade);
        sp = sp->next;
    }
}

```

### Recursive versions of print\_list

**/\* print student list recursively (from head to tail)\*/**

```

void print_list_1(student* sp){
    if (sp){
        printf("Name: %s Grade: %d \n", sp->name, sp->grade);
        print_list_1(sp->next);
    }
}

```

**/\* print student list recursively (from tail back to head)\*/**

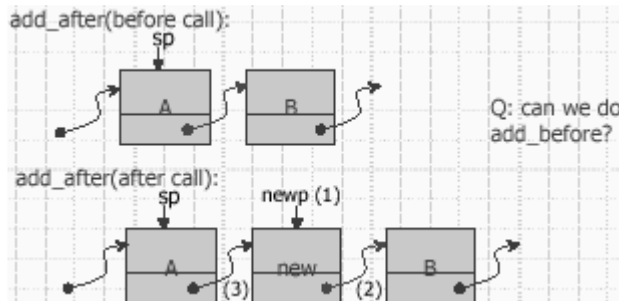
```

void print_list_2(student* sp){
    if (sp){
        print_list_2(sp->next);
        printf("Name: %s Grade: %d \n", sp->name, sp->grade);
    }
}

```

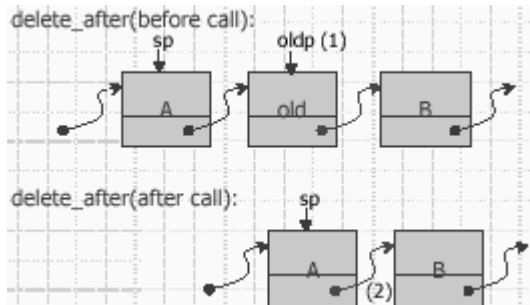
**/\* add a new student record after sp \*/**

```
void add_after(student* sp, char name*, int grade){  
    student* newp = make_student(name, grade); // (1)  
    newp->next = sp->next; // (2)  
    sp->next = newp; // (3)  
}
```



**/\* delete the student record after sp \*/**

```
void delete_after(student* sp){  
    student* oldp;  
    if (!sp || !sp->next) return;  
    oldp = sp->next; // (1)  
    sp->next = oldp->next; // (2)  
    free(oldp);  
}
```



## Linked list vs Array

### • Array:

- static storage allocation
- storage is contiguous
- random access (index)
- insert and delete must shift existing data

### • Linked List:

- dynamic storage allocation
- storage is not contiguous

- sequential access only
- insert and delete do not change existing data

### Ordered Linked List

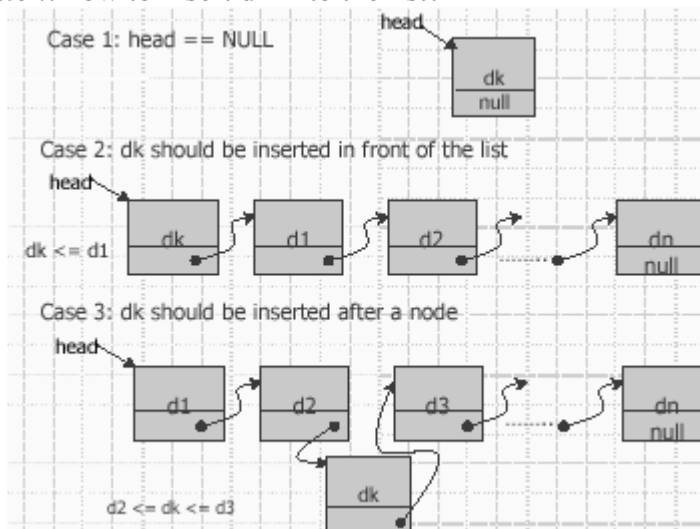
- Insert nodes in their sorted position.

```
typedef struct node {
    int data;
    struct node* next;
} node;
```



where  $d1 \leq d2 \leq d3 \leq \dots \leq dn$

- *Question:* how to insert **dk** into the list?



- can we declare:

```
void insert(node * hd, int data){ ... }
```

- suppose we have the code:

```
node* head = NULL;
insert(head, 2);
insert(head, 5);
...
```

- NO, because list head will be modified in both Case 1 and Case 2.

```
void insert(node ** hp, int data){ ... }
node* head = NULL;
insert(&head, 2);
insert(&head, 5);
...
```

**/\* Version 1: insert a new node (data) into a list pointed to by \*hp \*/**

```
void insert(node** hp, int data){
    node* new, *prev == NULL, *curr;
    new = make_node(data);           // suppose we have this function
    curr = *hp;                       // get head pointer

    while (curr && data > curr->data){ // find position
        prev = curr;
        curr = curr->next;
    }

    if (prev == NULL){                // or if (!prev)
        new->next = *hp;               // insert in front
        *hp = new;
    }
    else {                             // insert after prev
        prev->next = new;
        new->next = curr;
    }
}
```

**/\* Version 2: insert a new node (data) into a list pointed to by \*hp \*/**

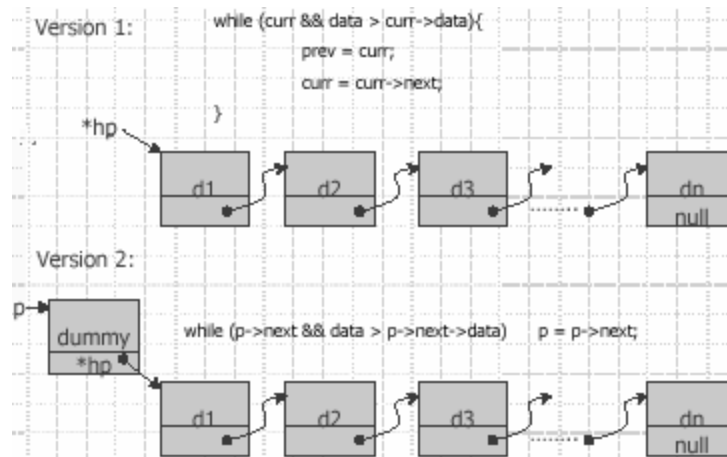
```
void insert(node** hp, int data){
    node dummy, *new, *p;

    new = make_node(data);           // suppose we have this function
    p = &dummy;
    dummy.next = *hp;                // set head in dummy

    while (p->next && data > p->next->data) // find position
        p = p->next;

    new->next = p->next;              // always insert after p
    p->next = new;
    *hp = dummy.next;
}
```

*See illustration on next page...*



**/\* delete the node (data) from a list pointed to by \*hp \*/**

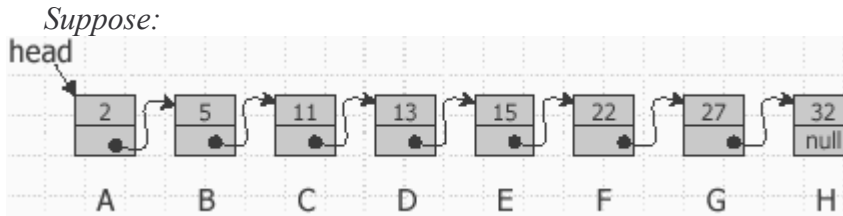
```

void delete(node** hp, int data){
    node dummy, *old, *p;
    p = &dummy;
    dummy.next = *hp;           // set head in dummy

    while (p->next && data != p->next->data) // find position
        p = p->next;

    if (p->next){
        old = p->next;           // get the node to be deleted
        p->next = p->next->next;
        free(old);              // free the node
    }
    *hp = dummy.next;
}

```



```

node *head, *p, *q, *temp;
int data;
p = head;
q = NULL;

```

```

(1) temp = p-> next-> next-> next;           temp == addr_of(D)
(3) data = 22;
    while (p && data > p-> data)
        p = p-> next;                         p == addr_of(H)
(2) while (p && p-> next) p = p-> next;
(4) data = 29;
    while (p && data > p-> data){
        q = p;                               p == addr_of(F)
        p = q-> next;                         p == addr_of(H)
        q = p;                               q == addr_of(G)
    }

```

```

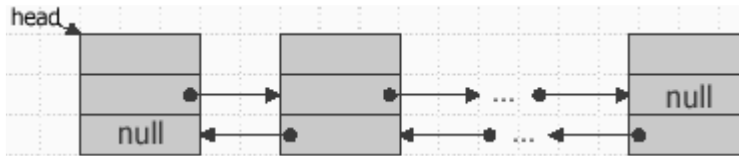
/* delete the node (data) from a list pointed to by *hp */
void delete( node** hp, int data){
    node dummy, *old, *p;
    p = &dummy;
    dummy.next = *hp;           // set head in dummy
    while (p-> next && data != p-> next-> data) // find position
        p = p-> next;
    if (p-> next){
        old = p-> next; // get the node to be deleted
        p-> next = p-> next-> next;
        free( old); // free the node
    }
    *hp = dummy.next;
}

```

## Doubly Linked List

A list which can be traversed either forward or backward:

```
typedef struct node{
    int data;
    struct node* next;
    struct node* prev;
} node;
```



### **insert\_after:**

```
new-> next = p-> next;
new-> prev = p;
p-> next = p-> next-> prev = new;
```

### **insert before:**

```
new-> next = p;
new-> prev = p-> prev;
p-> prev = p-> prev-> next = new;
```

*Example:*

Suppose we have a doubly linked list:

- (1) find the mid node,
- (2) along mid's prev link, decrease each node value by 2, along mid's next link (include mid node), increase each node value by 5.

```
#define NEXT 0
#define PREV 1
typedef struct node{
    int data;
    struct node *link[ 2];
} node;

void traverse( node* p, int dir, void (* fp)( node*)){
    while (p){
        (* fp)( p); // call function fp with argument p
        p = p-> link[ dir];
    }
}

void dec2( node* p){ p-> data -= 2; }
void inc5( node* p){ p-> data += 5; }
```

```

void processing( node* p){
    node* mid = p;

    while (p && p-> link[ NEXT] && p-> link[ NEXT]-> link[ NEXT]){
        mid = mid-> link[ NEXT];
        p = p-> link[ NEXT]-> link[ NEXT];
    }

    traverse( mid, PREV, dec2); // any problem here?
    traverse( mid, NEXT, inc5);
}

```

**Function Pointer:**  
**type (\*fp) (parameter\_list);**  
**Function name is the pointer to that function.**

## Lecture 23 – The Stack

3 March 2004

### The Stack:

- Linked lists can be used to implement stack data
- structure: add and remove node from the “top”.
- Property: Last In First Out (LIFO).
- Operations: is\_ empty, push, pop

*Example...*

$\pi * 10^{128} + e * 10^{128}$  Each operand has 128 digits

Read an operand(from left to right):	3141592653.....
	27182818.....
Calculate the sum(from right to left):	xx...xxxxx
	+ yy...yyyyy
	= ss...sssss

- We need three stacks - two for operands and one for the sum.

```

typedef struct node node;
struct node{
    int n;
    node* next;
};

```

```

#define empty( s) (!( s))

void push( node** top, int n){
    node* new = malloc( sizeof( node)); // create a new node
    if (! new) exit(- 1);
    new-> n = n;
    new-> next = *top;
    *top = new; // set up stack top
}

int pop( node** top){ // return a value (not a node)
    int n;
    node* temp;
    if (empty(* top)) return 0;
    temp = *top; // save top node
    *top = temp-> next; // set up stack top
    n = temp-> n;
    free( temp);
    return n;
}

node* get_ operand(){ // read an operand and store on a stack
    node* s = NULL;
    int n;
    while (1){
        n = getchar();
        if (n < '0' || n > '9') return s; // return if input char
is not a digit
        push(& s, n - '0');
    }
}

main(){
    node *a, *b, *sum = NULL;
    int sumdig, carry = 0;
    printf("\n input two operands\ n");
    a = get_ operand();
    b = get_ operand();
    while (! empty( a) || !empty( b)){ // perform addition
        sumdig = pop(& a) + pop(& b) + carry;
        push(& sum, sumdig% 10);
        carry = sumdig/ 10;
    }
    if (carry != 0) push(& sum, carry);
    printf("\n the sum is:");
    while (! empty( sum)) printf("% c", pop(& sum)); // output result
}

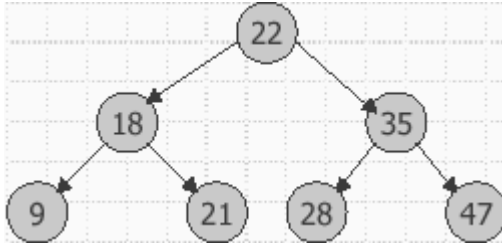
```

## Lecture 24 – Binary Trees

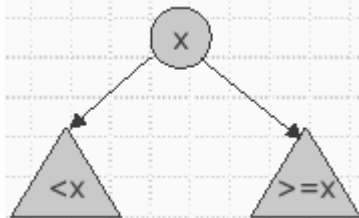
5 March 2004

### A Tree:

- A binary tree is either empty or a node whose left and right children are binary trees.



- A binary search tree is a binary tree such that for any node in the tree, we have:



```
typedef struct node node;
struct node {
    int data;
    node* left;
    node* right;
};

#define empty_tree( t ) (!( t))

node* make_node( int d){
    node* new = malloc( sizeof( node));
    if (! new) {
        printf(" make_node: memory allocation failed\n");
        exit(- 1);
    }
    new-> data = d;
    new-> left = new-> right = NULL;
    return new;
}

void insert( node** rp, int d){ // a recursive version
    if (empty_tree(* rp))
        *rp = make_node( d);
    else{
        if (d < (* rp)-> data) insert(&((* rp)-> left), d);
        else insert(&((* rp)-> right), d);
    }
}
```

```

void insert( node** rp, int d){ // a non- recursive version
    node *curr, *prev, *np;
    np = make_node( d);
    curr = *rp;
    prev = NULL;
    while (curr){ // walk down the tree until reach a NULL pointer
        prev = curr;
        curr = (curr-> data > d) ? curr-> left : curr-> right;
    }
    if (! prev) *rp = np; // if the tree was empty
    else if (prev-> data > d) prev-> left = np;
    else prev-> right = np;
}

int find( node* root, int d){
    while (root && root-> data != d)
        root = (root-> data > d) ? root-> left : root-> right;
    if (root) { // d has been found }
    else { // not found }
}

void print_tree( node* root){
    if (root){
        print_tree( root-> left);
        printf("% d\ n", root-> data);
        print_tree( root-> right);
    }
}

```

## Lecture 25 – Directives

**8 March 2004**

### Preprocessor Directives:

- Preprocessor reads a source program (file), performs various actions, and passes the resulting output to compiler.

- Actions are defined by preprocessor directives, such as:

#define	#include	#undef	#if
#else	#elif	#endif	#ifdef
#ifndef	#line	#error	#pragma

### #define

<b>#define macro</b>	<b>replacement_text</b>
#define PI	3.1415926
#define TWOPI	(PI + PI)
#define ALLOC(n,t)	(malloc(( n) * sizeof( t)))
#define MAX(x,y)	(( x) > (y) ? (x) : (y))

- To define symbolic constants; to define simple expression (macro looks like a function call);
- The preprocessor performs macro expansion, replacing the macro with replacement text (include replacing parameters)

### Using macros:

```
#define TWOPI (3.1415926* 2)
#define MAX( x, y) (( x) > (y) ? (x) : (y))

int main() { /* find the max circumference */
    double c1, c2, radius1, radius2;
    printf(" Enter two radius:");
    scanf("% lf %lf", &radius1, &radius2);
    c1 = TWOPI * radius1;
    c2 = TWOPI * radius2;
    printf(" The max circumference is %lf\ n", MAX( c1, c2));
}
```

- Regardless of how complex a macro is, the preprocessor simply substitutes the expression for its defined name.
- Always parenthesize the replacement text and parameters.

### **Wrong:**

```
#define SQ( x) (x* x)
int m = 6, n = 7, h;
h = SQ( m + 2, n - 3); // m + 2 * n - 3
```

### **Right:**

```
#define SQ( x) (( x)*( x))
int m = 6, n = 7, h;
h = SQ( m + 2, n - 3); // (m + 2)*( n - 3)
```

## Lecture 26 – Directives II

10 March 2004

### #include

- Instructs the preprocessor to replace the current line with the entire contents of the specified file.

```
#include <filename>
#include "filename"
```

- File inclusion allows us to create a header file for (1) useful definitions; (2) useful (macro) operations; and (3) useful new data types.

```
/* default definitions: defs. h */
#include <stdio. h>
#include <stddef. h>
#include <stdlib. h>
#include <string. h>
```

```

#define MIN( x, y) (( x) < (y) ? (x) : (y))
#define MAX( x, y) (( x) > (y) ? (x) : (y))
#define INRANGE( x, y, v) (( v) >= (x) && (v) <= (y))
/* hide ugliness of strcmp */
#define SEQ( x, y) (strcmp( x, y) == 0)
#define SLT( x, y) (strcmp( x, y) < 0)
#define SGT( x, y) (strcmp( x, y) > 0)

```

---

## Lecture 27 – Directives III

12 March 2004

### Conditional Compilation (#if)

- allow us to compile a program into different versions, depending on our needs.

```

    #if constant_expression
        first_group_of_statements
    #else
        second_group_of_statements
    #endif

/* table averaging function: tavg. c */
double table_avg( int * p, int n){
    int sum = 0;
    int *ep = p + n;
    #if defined( DEBUG)
        printf(" In function tavg: p = %d, ep = %d\ n", p, ep);
    #endif
    for ( ; p < ep; p++) sum += *p;
    return (n != 0) ? (double) sum/ n : 0.0;
}

/* using table_avg function */
#include <stdio. h>
int main(){
    int table[] = {91, 50, 77, 28, 11, 62, 83};
    int n = sizeof( table)/ sizeof( table[ 0]);
    #if defined( DEBUG)
        printf(" In main. c: Average %d values.\ n", n);
    #endif
    printf(" Table average is: %f\ n", table_avg( table, n));
}

    $ gcc -o test1 -DDEBUG tavg. c main. c
    $ gcc -o test2 tavg. c main. c

```

External Files:

- Abstractly, a file can be thought of as a stream of data (either char or binary).
- C has two groups of files: standard files, such as **stdin**, **stdout**, **stderr**, and external files.
- External files have several important properties: they have a name, they must be opened and closed, they can be written to, or read from, or appended to.
- Before accessing an external file, we have to open it.  
`FILE *fp = fopen("mydata", "r");`
- where FILE is a structure (stdio.h) containing the current state of a file and **fopen** is the standard I/O function, which takes two string parameters – a file name and a mode – and returns a pointer to a FILE.
- The mode specifies how to use the file. There are a number of possibilities for the mode.
- Each of these modes can end with a + character which means the file is to be opened for both reading and writing.
- “r+” means open text file for reading and writing

<u>Meaning</u>	<u>Mode</u>
open binary file for appending	“ab”
open binary file for writing	“wb”
open binary file for reading	“rb”
open text file for appending	“a”
open text file for writing	“w”
open text file for reading	“r”

Text Files versus Binary Files:

- A text file contains a sequence of characters. The standard input and out files are text files. (We have to convert back and forth between a data type’s internal representation and its char representation, such as 22.45 and “22.45”).
- A binary file is simply a stream of bytes. There is no conversions needed. We may use binary files to move data directly from memory to disk, exactly as is.
- Binary files are more efficient to process, however, they are not portable and not easily viewable.

Open an File:

- `fopen("filename", "r")` returns NULL if the file does not exists.
- `open("filename", "w")` causes the file to be created if it does not exists, or overwritten if it does.
- `fopen("filename", "a")` causes the file to be created if it does not exists, or causes writing to occur at the end of the file if it does.

## A Simple Example:

```
#include <stdio.h>

int main(void){
    int sum = 0, val;
    FILE *inf, *outf;

    inf = fopen("my_data", "r");
    outf = fopen("my_sum", "w");

    while (fscanf(inf, "%d", &val) == 1) sum += val;

    fprintf(outf, "The sum is %d. \n", sum);
    fclose(inf);
    fclose(outf);
}
```

## Character File I/O:

- Access an open file a character at a time, using **getc** and **putc**.
- **getc** takes a file pointer and returns the integer representation of the next character in the file, or EOF if it encounters an end of file.
- **putc** takes the integer representation of a character and a file pointer and writes the character to the file.
- Both **getc** and **putc** return EOF if an error occurs.

## How to Check End of File?

- Check EOF during accessing a file:

```
int c;
while ((c = getc(inf)) != EOF) {
    // do something
}
```

- Call standard file I/O functions:

```
while (!feof(inf)){
    c = getc(inf);
    // do something
}
```

- EOF could indicates either an error or end of file. To detect an error from end of file, you may use feof and ferror.

**Example: Double-spacing a File:**

```
#include <stdio.h>
#include <stdlib.h>
void double_space(FILE*, FILE*);
void print_info(char*);

int main(int argc, char* argv[]){
    FILE *ifp, *ofp;
    if (argc != 3){
        print_info(argv [0]);
        exit(1);
    }

    ifp = fopen(argv[1], "r");
    ofp = fopen(argv [2], "w");
    double_space(ifp, ofp);
    fclose(ifp);
    fclose(ofp);
    return 0;
}

void double_space(FILE *ifp, FILE *ofp){
    int c;
    while ((c = getc(ifp)) != EOF){
        putc(c, ofp);
        if (c == '\n') putc('\n', ofp);
    }
}

void print_info(char* pgm){
    printf("\n%s%s%s\n\n%s%s\n\n", "Usage: ", pgm, " infile outfile",
        "The contents of infile will be double-spaced ", "and written to
        outfile");
}
```

## Example: File-copying:

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

long file_copy(FILE*, FILE*);

int main(int argc, char * argv[]){
    FILE * ifp, * ofp;
    long count;
    int status = EXIT_FAILRE;
    if (argc != 3)
        printf ("Usage: %s source_file dest_file\n", argv[0]);
    else if (( ifp = fopen(argv[1], "rb")) == NULL)
        printf ("Can't open %s for reading\n", argv[1]);
    else if ((ofp = fopen(argv[2], "wb")) == NULL)
        printf ("Can't open %s for writing\n", argv[2]);
    else {
        if ((count = file_copy(ifp, ofp)) == -1L)
            printf ("Copy failed\n");
        else {
            printf ("Copied % li bytes\n", count);
            status = EXIT_SUCCESS;
        }
    }
    return status;
}

/* file_copy.c */
#include <stdio.h>
#include <stdlib.h>

long file_copy(FILE *ifp, FILE *ofp){
    int c;
    long cnt ;
    for (cnt = 0L; (c = getc(ifp)) != EOF; cnt ++ )
        putc(c, ofp);
    if (ferror(ifp) || ferror(ofp))
        cnt = -1L;
    fclose(ifp);
    fclose(ofp);
    return cnt ;
}
```

## Formatted File IO:

- Both `printf` and `scanf` have counterparts that do formatted I/O on files:

```
fprintf(file_pointer, format_string, ...)
fscanf(file_pointer, format_string, ...)
```

- *Example:*

```
int a[20], i, j;
FILE* ofp;
for (i = 0; i < 20; i++)
    a[i] = i;
ofp = fopen("my_output", "w");
for (i = 0; i < 4; i++) {
    for (j = 0; j < 5; j++)
        fprintf(ofp, "%i ", a[i*5 + j]);
    fprintf(ofp, "\n");
}
fclose(ofp);
```

## Line-oriented File I/O:

- The standard I/O library provides functions that do line-oriented I/O: `fgets` and `fputs`.
- `fgets` takes three parameters: a char array, its size, and a file pointer. It reads chars into the array, stopping when it meets a newline (this newline is included in the array) or find that the array is full. It terminates the array with a NULL.
- `fputs` takes a string (with a `'\n'`) and a file pointer and writes the string to the file.

- *Example:*

```
#include <stdio.h>
#include <string.h>
#define MAXLEN 80
FILE *gfopen(char*, char*);

long file_copy(char* dest, char* source){
    FILE *sfp, *dfp;
    char line[MAXLEN + 1];
    long cnt;
    sfp = gfopen(source, "rb");
    dfp = gfopen(dest, "wb");
    for (cnt = 0L; fgets(line, sizeof(line), sfp) != NULL; cnt += strlen(line))
        fputs(line, dfp);
    fclose(dfp);
    fclose(sfp);
    return cnt;
}
```

*/\* A graceful version of fopen \*/*

```
FILE *g fopen(char *fname, char* mode){
    FILE *fp;
    if ((fp = fopen(fname, mode) == NULL){
        fprintf(stderr, "Can't open %s - BYE!\n", fname);
        exit(1);
    }
    return fp;
}
```

### Random File Access:

- Accessing a random file is similar to an array, indexing any byte in the file as we would an array element.

- *For example:*

The Kth element of an array is located at position:

`starting_address + k * sizeof(array_element)`

The kth record in a file is located at position:

`start_of_the_file + k * size_of_record`

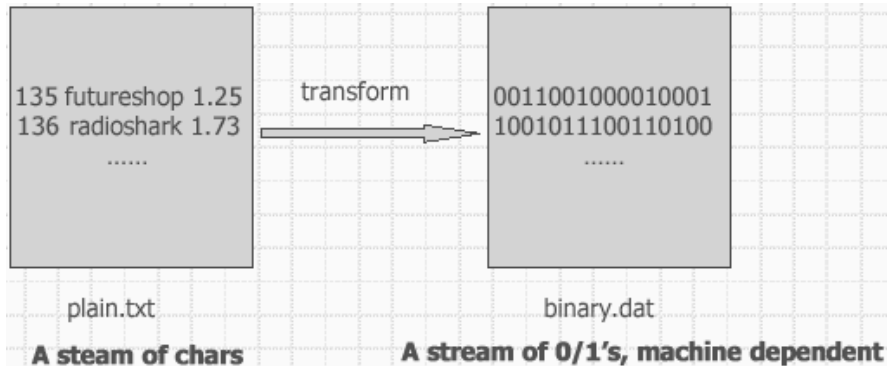
### Block I/O:

- C provides two functions to read and write arrays of bytes: **fread** and **fwrite**.
- Both of them takes four parameters:
  - a pointer to the array's first element (void\*)
  - the size (in bytes) of an element
  - the number of elements in the array
  - a file pointer

- *Example:*

```
#ifndef STORE_H
#define STORE_H
#define INFILE "plain.txt"
#define OUTFILE "binary.dat"
struct store{
    int id;
    char name[15];
    float price;
};
#endif
```

- *What we want:*



- *The Code:*

```
#include <stdio.h>
#include "store.h"
FILE* g fopen(char*, char*);

int main(){
    struct store srec;
    FILE *ifp, *ofp;
    ifp = g fopen(INFILE, "r");
    ofp = g fopen(OUTFILE, "w");
    while (fscanf(ifp, "%d%d%f", &srec.id, srec.name, &srec.price) != EOF)
        fwrite(&srec, sizeof(struct store), 1, ofp);
    fclose(ifp);
    fclose(ofp);
}
```

### Locating a Position:

- Three standard I/O functions support random file access: `fseek`, `rewind`, and `ftell`.
- `fseek` is used to move the file pointer to a specific byte in the file:

```
int fseek(FILE *fp, long offset, int from);
```

where `from` must be one of three values:

```
SEEK_SET // beginning of the file
SEEK_CUR // current position
SEEK_END // end of the file
```

returns `-1` if there is an error and `0` otherwise.

- *Examples: `fseek`*

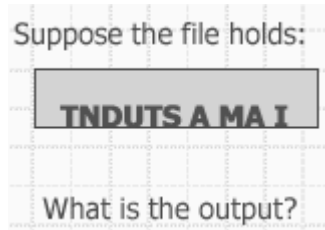
```
fseek(fp, 0L, SEEK_SET); // go to the start of the file
fseek(fp, 0L, SEEK_CUR); // don't move
fseek(fp, 0L, SEEK_END); // go to the end of the file
fseek(fp, n, SEEK_SET); // go to the nth byte in the file
fseek(fp, n, SEEK_CUR); // skip ahead n bytes
fseek(fp, -n, SEEK_CUR); // go backward n bytes
fseek(fp, -n, SEEK_END); // go to n bytes before the end
```

### rewind and ftell:

- `rewind` is a special case of `fseek` that moves the file pointer to the start of the file, i.e., `fseek(fp, 0L, SEEK_SET)`, except it returns no value and clears the internal EOF and error indicators.
- `ftell` takes a file pointer and returns a long containing the current offset in the file (the position of the next byte to be read/written).
- Typically, programs use `ftell` to save their current position in a file so that they can easily return to it later.

#### • *Example:*

```
#include <stdio.h>
int main(){
    int c, n;
    long offset, last;
    FILE *fp;
    fp = fopen("rev.dat", "r");
    fseek(fp, 0L, SEEK_END);
    last = ftell(fp);
    for (offset = 0; offset <= last; offset++){
        fseek(fp, -offset, SEEK_END);
        c = getc(fp);
        if (c != EOF) printf("%c", c);
    }
    printf("\n");
    fclose(fp);
}
```



---

#### • *Example: Random Access:*

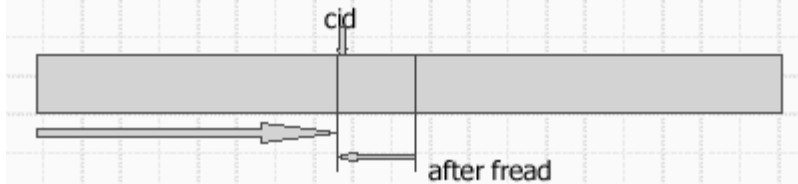
```
#include <stdio.h>
struct customer {
    int id;
    float balance;
};

void init_database(FILE *ofp, int num){
    struct customer temp = {0, 0.0};
    int k;
    for (k = 0; k < num; k++){
        temp.id = k;
        fwrite(&temp, sizeof(struct customer), 1, ofp);
    }
}
```

```

int update(int cid, FILE* fp, int amount){
    struct customer temp;
    if (fseek(fp, cid*sizeof(struct customer), SEEK_SET) < 0)
        return -1;
    fread(&temp, sizeof(struct customer), 1, fp);
    temp.balance += amount;
    fseek(fp, cid*sizeof(struct customer), SEEK_SET);
    /* fseek(fp, -sizeof(struct customer), SEEK_CUR); */
    return fwrite(&temp, sizeof(struct customer), 1, fp);
}

```




---

```

int print_database(FILE *fp){
    struct customer temp;
    rewind(fp);
    while (fread(&temp, sizeof(struct customer), 1, fp) > 0){
        printf("ID: %4d BALANCE: %.2f\n",
            temp.id, temp.balance);
    }
}

int main(){
    // ask for database name
    // show options : init, update, print, or quit
    // perform the selected option
    // repeat
}

```

Generic Functions:

- A generic function is one that can work on any underlying C data type.
- Generic functions allow us to reuse programs by adding a small piece of type-specific code.
- For example, standard C library provides two generic functions: **bsearch** searches an arbitrary array, and **qsort** sorts an arbitrary array.

Review of Pointers to Functions:

- A pointer to a function contains the address of the function in memory.
- Similar to an array name which is the starting address of the first array element, a function name is the starting address of the function code.
- Pointers to functions can be passed to functions, returned from functions, stored in an array, and assigned to other function pointers.

Pointers to Functions:

```
int compare(int a, int b){
    // implementation
}

int main(){
    int cr1, cr2;
    int (*pf)(int, int); // declare a pointer to function
    pf = compare; // assign a function pointer to pf
    cr1 = (*pf)(2, 3); // call the function
    cr2 = pf(2, 3); // same as above but not recommended
}
```

Generic Binary Search:

```
void * bserach(const void *target, const void *table, size_t n,
size_t size, int (*cmpfp)(const void *, const void*));
```

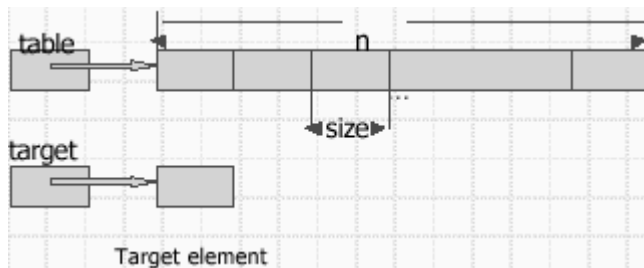
target: a pointer to the element we are searching for

table: the array we are searching

n: the size of the array

size: the size of an array element

cmpfp: a pointer to a function to compare the target and an element



### bsearch Function:

```
void * bsearch(const void *target, const void *table, size_t n,
size_t size, int (*cmpfp)(const void *, const void*)) {
    void *min = table, *max = table + (n - 1)*size;
    void *mid;
    int k = n/2;
    int cr;

    while (min < max) {
        mid = min + k*size
        cr = (*cmpfp)(target, mid) ;
        if (cr == 0) return mid;
        else if (cr > 0) min = mid + size;
        else max = mid - size;
        k /= 2;
    }
    return NULL;
}
```

### Invoking bsearch:

- bsearch returns a pointer to the matching array element if it finds one, or NULL otherwise.
- Suppose we have an int table **intTable** and a string table **stringTable**, we might call:  
int key = 78;  
int\* ip = bsearch(&key, intTable, ISIZE, sizeof(int), cmpInt);  
char\* sp = bsearch("Tom Wilson", stringTable, SSIZE,  
sizeof(stringTable[0]), cmpStr);

---

## Lecture 32 – Type-Specific Functions

24 March 2004

### Type-specific Functions:

```
int cmpInt (const void *tp, const void *ep){
    int it = *(int*) tp; // get target value
    int ie = *(int *) ep; // get array element value
    return (it == ie) ? 0 : ((it < ie) ? -1 : 1);
}

int cmpStr(const void *tp, const void *ep){
    char *ctp = (char*) tp; // casting target to char pointer
    char *cep = *(char**) ep; // get array element value (a char
pointer)
    return strcmp(ctp, cep);
}
```

- *Another Example: // In-place insertion-sort an array of integers:*

```
void isort_int(int table[], int n){
    int j, k, temp;
    for (j = 1; j < n; j++){
```

```

        temp = table[j];
        for (k = j; k > 0; k--){
            if (temp >= table[k-1]) break;
            table[k] = table[k-1];
        }
        table[k] = temp;
    }
}

```

### ***// Generic insertion-sort:***

```

#include <stddef.h>
#include <stdlib.h>
#include <string.h>

void isort_generic(void *table, int n, size_t size,

int (*cmp)(const void*, const void*)){
    char *sp = table, *pj, *pk;
    char *ep = sp + n*size;
    void * temp = malloc(size);
    for (pj = sp + size; pj < ep; pj +=size){
        memcpy(temp, pj, size);
        for (pk = pj; pk > sp; pk -= size){
            if ((*cmp)(temp, pj - size) >= 0) break;
            memcpy(pk, pk - size, size);
        }
        memcpy(pk, temp, size);
    }
    free(temp);
}

void isort_int (int table[], int n){
    int j, k, temp;
    for (j = 1; j < n; j++){
        temp = table[j];
        for (k = j; k > 0; k--){
            if (temp >= table[k-1]) break;
            table[k] = table[k-1];
        }
        table[k] = temp;
    }
}

```

### **Call isort\_generic:**

- Suppose we have along table **longTable**, and a string table **stringTable**, we might call:

```

isort_generic(intTable, TSIZE, sizeof(longTable[0]), cmpLong);
isort_generic(stringTable, SSIZE, sizeof(stringTable[0]), cmpStrs);

```

- How to implement cmpLong and cmpStrs?

```

int cmpLong(const void *p1, const void *p2){
    long lp1, lp2;
    lp1 = *(long*) p1;
    lp2 = *(long*) p2;
    return lp1 - lp2;

    // or even better: return *(long*) p1 - *(long*) p2;
}

int cmpStrs(const void *p1, const void *p2){ // for isort_generic
    return strcmp(*(char**) p1, *(char**) p2);
}

/* what is the difference of cmp functions for isort and bsearch? */
int cmpStr(const void *tp, const void *ep){ // for bsearch
    char *ctp = (char*) tp; // casting target to char pointer
    char *cep = *(char**) ep; // get array element value (a char
pointer)
    return strcmp(ctp, cep);
}

```

### Example: A Stack-based Calculator

- Use command-line arguments as input and calculate basic arithmetic wrt the given data and operations.
- The program interprets the following operations:
  - +: pop two numbers, add them and push the result back
  - : pop two numbers, subtract them and push the result back
  - mul: pop two numbers, multiply them and push the result back (\*\*)
  - /: pop two numbers, divide them and push the result back
  - xchg: exchange the top two elements
  - print: pop and print the top element
  - dup: duplicate the top element
- *Example:* \$scal 4 dup dup mul xchg 2 / - print  
4 \* 4 - 4/2

(\*\*) Why don't use \* instead of mul?

- Because different shells have different meanings if \* is used as the command line argument. For example, Bash interprets \* as file names of the current directory.
- Suppose there are three files under the current dir:

```
$prog *  
argc = 4  
argv[0] = "prog"  
argv[1] = "f1.c"  
argv[2] = "f1.o"  
argv[3] = "f1.h"
```

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

```
typedef struct node node;  
struct node{  
    double n;  
    node* next;  
};
```

```
#define empty() !(stk)
```

```
node* stk = NULL;
```

```
void push(double n){  
    node * new = (node*) malloc(sizeof(node));  
    if (!new) exit(-1);  
    new->n = n;  
    new->next = stk;  
    stk = new;  
}
```

```
double pop(){  
    double n;  
    node* t;  
    if (empty()) return 0.0;  
    t = stk;  
    n = stk->n;  
    stk = stk->next;  
    free(t);  
    return n;  
}
```

```
void add(){  
    double b = pop();  
    push(pop() + b);  
}
```

```

void sub(){
    double b = pop();
    push(pop() - b);
}

void fmul(){
    double b = pop();
    push(pop() * b);
}

void fdiv(){
    double b = pop();
    push(pop() / b);
}

void xchg(){
    double a = pop();
    double b = pop();
    push(a);
    push(b);
};

void print(){
    printf("Result = %f\n", pop());
}

void dup(){
    double b = pop();
    push(b);
    push(b);
}

char* operation[] = { "+", "-", "mul", "/", "xchg", "print", "dup"};
void (*action[])() = {add, sub, fmul, fdiv, xchg, print, dup};

void exec(char * cmd){
    int i;
    for (i = 0; i < sizeof(operation)/sizeof(char*); i++){
        if (strcmp(cmd, operation[i]) == 0){
            (*action[i])();
            return;
        }
    }
    printf("Illegal operation %s\n", cmd);
}

int main(int argc, char* argv[]){
    int i;
    for (i = 1; i < argc; i++){
        if (isdigit(argv[i][0]))
            push(atof(argv[i]));
        else
            exec(argv[i]);
    }
    exit(0);
}

```

Constants:

```
? #define ONE 1
? #define TEN 10
#define INPUT_MODE 1
#define BUFSIZE 20
```

Name:

- Use descriptive names for globals, short for locals.
- Be consistent when naming functions, types, variables, and constants.

```
? for (theElementIndex = 0;
    theElementIndex < numberOfElements;
    theElementIndex++)
    elementArray[ theElementIndex] = 0;
```

```
for (i = 0; i < n; i++) a[ i] = 0;
```

Convention:

- Do not use id's that contains two or more underscores in a row.
- Do not use id's that begin with an underscore.

```
int i_ _j = 11;      // illegal
int _ _m = 20;      // illegal
int _k = 10;        // not recommended
```

- Do not change a loop variable inside a for loop block.
- Update loop variables close to where the loop condition is specified.
- All flow control primitives (if, else, while, for, do, switch, and case) should be followed by a block, even if it is empty.
- Statements following a case label should be terminated by a statement that exits the switch statement.
- All switch statements should have a default case.
- Use **break** and **continue** instead of **goto**.
- Do not have overly complex functions.
- Indent to show program structure (better readability).
- Parenthesize to resolve ambiguity.

```
for (j = 0; j < n; j++){
    a[ j] = j;
    for (k = j ; k < n; k++){
        if (a[ j] < 5)
            a[ k] = a[ j];
        else
            a[ k] = k;
    }
}
```

```

? for (j = 0; j < n; j++){
    a[ j] = j;
    for (k = j ; k < n; k++){
        if (a[ j] < 5)
            a[ k] = a[ j];
        else
            a[ k] = k;
    }
}

```

```

? Leapyear = y % 4 == 0 && y % 100 != 0 || y % 400 == 0;
Leapyear = (( y % 4 == 0) && (y % 100 != 0) ) || (y % 400 == 0);

```

- Be careful with **side effects**.

Operators like ++ have side effects: besides returning a value, they also modify an underlying variable. Side effects can be extremely convenient, but they can also cause trouble because the actions of retrieving the value and updating the variable might not happen at the same time. In C the order of execution of side effects is undefined.

```

? str[ j++] = str[ j++] = 'a';
str[ j++] = 'a';
str[ j++] = 'a';
? a[ j++] = j;
? scanf("% d% d", &x, &a[ x]);

```

### Break up complex expressions:

```

? *x += (* xp = (2 * k < (n - m) ? c[ k+ 1] : d[ k--]));

```

```

if (2 * k < n - m)
    *xp = c[ k+ 1];
else
    *xp = d[ k--];
*x += *xp;

```

```

? child = (! LC && !RC) ? 0 : (! LC ? RC : LC);

```

```

if (LC == 0 && RC == 0)
    child = 0;
else if (LC == 0)
    child = RC;
else
    child = LC;

```

### Use idioms for consistency:

```
?j = 0;
  while (j <= n - 1) a[ j++] = 1;
? for ( j = 0; j < n; ) a[ j++] = 1;
? for ( j = n; -- j >= 0; ) a[ j] = 1;

/* idiomatic form in C */
  for (j = 0; j < n; j++) a[ j] = 1;

/* standard loop for walking along a list */
  for (p = list; p != NULL; p = p-> next) ...
```

### Problem with function- like macros:

```
#define isupper( c) (( c) >= 'A' && (c) <= 'Z')

? while (isupper( c = getchar())) ...
while ( (c = getchar() != EOF && isupper( c )) ...
#define round( x) (( int) (( x) + (( x) > 0) ? 0.5 : -0.5))
? size = round( sqrt( dx* dx + dy* dy));

/* not an error, but low performance */
```

### Number 0:

```
? char *str = 0;
? name[ n- 1] = 0;
? double x = 0;

/* reverse o for a literal integer zero */
char *str = NULL;
name[ n- 1] = '\ 0';
double x = 0.0;
```

### List:

```
typedef struct node node;
struct node {
    int data;
    node* next;
};

/* addfront: add newp to front of listp (version 1) */
node* addfront( node *listp, node *newp){
    newp-> next = listp;
    return newp;
}
```

• *Usage:*

```
node* list = NULL;
node* newp = make_node( 234);
list = addfront( list, newp);
```

```
/* addfront: add newp to front of the list through lpp( version 2)*/
void addfront( node ** lpp, node * newp){
    newp-> next = *lpp;
    *lpp = newp;
}
```

• *Usage:*

```
node* list = NULL;
node* newp = make_node( 234);
addfront(& list, newp);
```

```
/* apply: execute fn for each element of listp */
void apply( node* listp, void (* fn)( node* , void*), void* arg){
    while( listp){
        (* fn)( listp, arg);
        listp = listp-> next;
    }
}

void print_node( node* p, void * arg){
    printf(( char*) arg, p-> data, (int) p-> next);
}
```

• *Usage:*

```
apply( listp, print_node, "data = %d, next = %d\ n");
```

## Binary Search Tree:

```
typedef struct node node;
struct node {
    int data;
    node* left;    // lesser
    node* right;   // greater or equal
};
```

```

/* insert newp in tp (recursive version 1) */
node* insert( node* tp, node* newp){
    if (tp == NULL) return newp;
    if (newp-> data < tp-> data)
        tp-> next = insert( tp-> left, newp);
    else
        tp-> right = insert( tp-> right, newp);
    return tp;
}

```

```

/* insert newp through tpp (recursive version 2) */
void insert( node ** tpp, node* newp){
    if (!(* tpp))
        *tpp = newp;
    else {
        if (newp-> data < (* tpp)-> data)
            insert(&((* tpp)-> left), newp);
        else
            insert(&((* tpp)-> right), newp);
    }
}

```

```

/* insert newp through tpp (non- recursive version) */
void insert( node ** tpp, node *newp){
    node *curr, *prev,;
    curr = *tpp;
    prev = NULL;
    while (curr){
        prev = curr;
        curr = (newp-> data < curr-> data) ? curr-> left: curr->
right;
    }

    if (! prev) *tpp = newp;
    else if (newp-> data < prev-> data)
        prev-> left = newp;
    else
        prev-> right= newp;
}

```

### Debugging:

• **Bug:** A defect or fault in a machine, plan, or the like. Oxford English Dictionary

- (1) do not blame your computer
- (2) do not blame your compiler
- (3) do not blame the standard library

→ experienced programmers know that, realistically, most problems are their own fault.

- Use debugger, or write self-checking code:

```
void check( char* s){
    printf("% s\ n", s);
    fflush( stdout);
    abort();
}

int main() {
    ...
    check("..."); // before suspect
    // suspect code
    check("..."); // after suspect
}
```

---

### **Lecture 35 – Sample Final “Self-Test”**

**31 March 2004**

- For the following multiple choice questions **bold all** the answers for each question that are correct. You will be penalized for circling answers that are incorrect. (NOTE: ON OUR FINAL, THERE WILL ONLY BE ONE CORRECT ANSWER).

**(1) Accessibility of a variable, i. e., its lifetime, is defined by its**

- a) data type      **b) storage class**  
c) name            d) memory address

**(2) If a local variable has the same name as a global variable, what will happen?**

- a) A compiler error gets generated.  
b) Both variables will share the same memory location.  
**c) The local variable will supersede the global variable, i. e. hide it.**  
d) Both are visible to the entire program.

**(3) Given the declaration, which scanf would successfully read in a float value into x?**

```
float x, *y = &x, ** z = &y;
```

- a) scanf("% f", &x);**            b) scanf("% f", \*y);  
c) scanf("% f", \*z);            d) scanf("% f", y);

**(4) Address of array declared as int x[1] is indicated by**

- a) \*(& x)      b) &x[ 0]  
c) &>(\* x)      d) none of the above

**(5) What would be the value of t after the following code executes?**

```
int t = 0;
char *s = "Hello world!";
char *u = s + 2;
for ( ; *u; ++ u, ++ t);
```

- a) 10**    b) 12  
c) 13    d) none of the above

**(6) What is the definition of or concept behind unions?**

- a) They allow you to put two things in the same place at the same time.
- b) They allow the same memory location to be accessed using different identifiers.**
- c) They allow merging of different structures.
- d) They are identical to structures.

**(7) Given the following expression which of the following is not true?**

`s[ 3]. x[ 0]-> t`

- a) s is an array.
- b) x[ 0] is a pointer to a structure.
- c) t is a member of s[ 3].**
- d) x is a member of s[ 3].

**(8) Text files are nice because they**

- a) are more efficient than binary files
- b) are portable among different platforms.**
- c) take up less space than binary files, typically.
- d) can be accessed randomly.

**(9) Given that x is an unsigned char,  $x \wedge 0xFF$  will always give**

- a) all zeros
- b) all ones
- c) x
- d) none of the above**

• Short Answer:

**(1) Write a recursive function `unsigned long sum( int n)` which returns all of the nonnegative integers summed up from 1 to n.**

**(2) Rewrite the following code using a for loop instead of a while loop.**

```
int sum = 0, i, n;
printf(" Enter i and n: ");
scanf("% d %d", &i, &n);
while (i < 2* n){
    if (i% 2 == 0)
        sum += i;
    i = i + 3;
}
```

**(3) Using the following declaration for each question, identify the output for each part.**

```
int x = 20, y = 5, *px = &x, *py;
int a[]={ 2, 4, 8, 1, 5, 3}, *pa = a;
```

```

a) printf("% d", (* px) + 3);

b) py = px;
   *py += *px* y;
   printf("% d\ n", x);

c) printf("% d\ n", *( pa + 3));

d) pa++;
   printf("% d\ n", *pa + 3);

```

**(4) Trace the program and give the output**

```

#include <stdio. h>
int main( void){
    char s[] = " bskbmshd";
    int i = 0;
    while( s[ i]){
        if (s[ i] < 'f')
            s[ i++] -= 1;
        else
            s[ i++] += 1;
    }
    printf(" The resulting string is %s.\ n", s);
    return 0;
}

```

**(5) Write a function `node * duplicate( node * np)` which creates a new copy of the node pointed by `np` such that the returned `node*` should point to a different block of memory. The following code shows the definition of node as well as the usage of the function:**

```

typedef struct { char* name, int len} node;
node n = {" a node", 6};
node *np1 = &n, *np2;
np2 = duplicate( np1);

```

**(6) An integer is said to be prime if it is divisible only by 1 and itself. For example, 2, 3, 5 and 13 are primes but 4, 6 and 8 are not. Write a C function, `int is_prime( int n)` which returns 1 if n is a prime or 0 if not.**

• More:

^ is (xor)

Switching vars:

a = a ^ b;

b = a ^ b;

a = a ^ b;

```
unsigned long sum(int n) {  
    if (n == 1)  
        return 1;  
    else if (n <= 0)  
        return 0;  
    else  
        return sum(n-1) + n;  
}
```

*Identify the output...*

---

**Lecture 36 – No Lecture Today**

**2 April 2004**

• Class cancelled. Good luck on exams!