CS350 - Tuesday, January 12, 2010

Proper Programming in C

There is a contest for Obfuscated C Programs — However we are expected to lose this contest.

Developed by two programmers at Bell Labs,
C was used to develop UNIX
They wanted to make a language that was machine portable

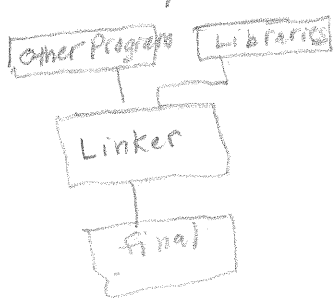
C Libraries
Use these instead of making your program

Differences between C / C++
C++ objected — g++ compiler)
C compile + libraries — later than C++ making it ideal for OS's

Create the file with an editor

* Editors: emac
preproccessing — # include <stdio.h>
#
* Compiler — gcc myProgram -O myProgram.o
* maybe Linker — link w/ libraries

DLL
Dynamic Linked Libraries

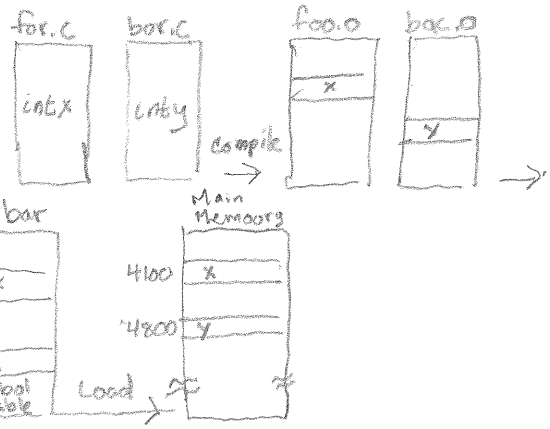


machine dependant
Loader — $ myProgram.exe — load into memory
Execute

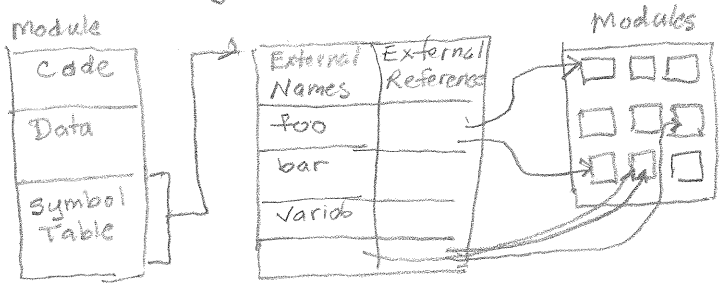
Thursday, January 14, 2010

www.ioccc.org/main - take a look

Dynamic Linking
Compile - Link - Load sequence



Nature of the object module



Ld (UNIX) code to Link both static + dynamic
gcc only static

static linking - all modules are put in object file as any updates would not be include

dynamic - place holder and

Advantage of dynamic
→ Space efficiency in main memory
↘ secondary memory
→ Allows update of individual modules, independent of the others
→ Security
→ Module Re-Use
⇒ Maintenance + Versioning

DLL Dynamic Linked Libraries

Problems

Generally slows down execution

DLL Hell! "Not available" changed name or path
The Hell of Dependency

Part II

Example C Program

Notes on

#include <stdio.h> < > standard directory ↺
same as import "my own.h"

int main (){ parameter
return 0; - Successfully completed otherwise -1

called functions, instead of method

Escape sequences

\n → new line
\a →
\\ backslash
\" double quotes
\t tab
\v vertical tab
\b backspace

name same rules

scanf ("%d", &x); & → address resolution operator
without it, you will get run-time error

scanf ("%s", myString); no & needed

Tuesday, January 19, 2010 Lecture #3

Topics – Binary and Relational Operators
Data Types in C
Promotion Rules
Structured Programs
Quiz

AND	OR	NOT
&&	||	!

0 ⇒ false
non zero ⇒ true

A	B	A&&B	A||B	!A
F	F	F	F	T
F	T	F	T	T
T	F	F	T	F
T	T	T	T	F

The use of multiple

$(x^{10} > y^{5})$ && $(y^{5} > z^{3}) \Rightarrow T$

x > y && y > z ⇒ T > and < takes precendence over &&
use paranthesis anyway

Precedence & Associativity of C operators

Operator	Description	Precedence	Associativity
functions → ()	Parathesis		L → R
±	unary plus unary minus		R → L
*, /, %	arithmetic		L → R
±	addition/subtraction		L → R
==, <, <=, >, >=	Relational Ops		L → R
&&, ||	Binary Ops		L → R
!	Binary Not		R → L
=	assignment operation		R → L

Data Types	Size (bytes)	Range	Precision	printf
long double	≥ 8, 10, 12			%Lf
double	8 (64b)	$-2.2\,10^{+308} \rightarrow 1.79 \times 10^{+308}$	15	%f for scanf use %Lf
float	4	$-1.17 \times 10^{+38} \rightarrow 3.4 \times 10^{+38}$	6	%f
long int	≥ 4			%ld
int	4	-2^{31} to $2^{31}-1$		%d
short	2	-2^{15} to $2^{15}-1$		%hd
char	1	$0 \rightarrow 2^{8}-1$		%c

# of bits = $\log_2 n$

float represent
$\pm 9.999999 \times 10^{+35}$

+1 | 9 | .999999 | +1 | 38

1	4	20	1	6

runs slower

Promotion Rules

How to convert one data type to another (casting)

```
int x = 10;
float y = 3.5;
int z;
```

( ) ← promotion

```
z = x * (int) y;    explicit casting

z = (int) (x * y);

z = x * y = 35      implicit casting
```

Implicit casting

Direction of conversion for implicit casting

long double
↑
double
↑
float
↑
long int
↑
int
↑
Shorts

(int + double) ⇒ (double + double) ⇒ double

Thursday, January 21, 2010

Topics - Structured Programming
- control structures
- Flow charts
- selection structures
- Repetition
- C

implicit casting

```
double d;
int x,y;
d = 2/3;  = 0.0      Integer division
d = (double)(2/3) = 0.0   The parathesis indicates integer division
d = (double) 2/3 =
```

double * int ⇒ double * double = double

```
d = 2.5;
x = 3
y = d * x;      y = 7
```

Thursday, January 21, 200

"The Mythical Man Month"
non-structured programs
spaghetti codes due
to goto statements

Structured Programming
1) Understand the problem
2) Design the algorithm
3) Know the header files + library

Algorithm - step-by-step ordered sequences of instructions
Program control

Structure programming make sure that uses control statement
take care of how you enter and leave a section of code
Avoid goto statement like the plague

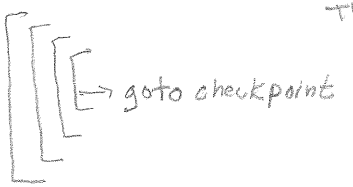
3 control structures
1) Sequence structures (in-built in our programming
3 Selections structures
if, if/else, switch
3 Repetition structures
while, do/while/for

Two ways to combine
1. control structure stacking

2. control structure nesting

Bohm/Jacopini wrote a paper
that proved you can write programs
without goto statements

When to use the goto statement
This allows the program to run faster if
you are in a deeply nested control structures



Use only if it makes yours programs
more efficient

Check points:
start with these statements

Flow Charts


start/stop


action


input/output


decision

flow of control


continuous tion

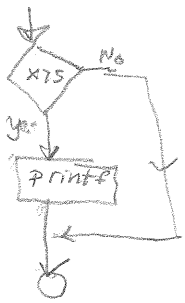
one entry/one exit
for first 3 symbols

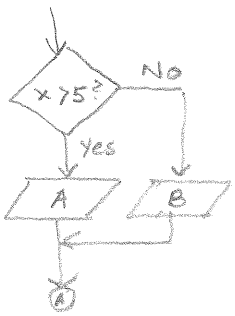
Thurs. January 21, 2010

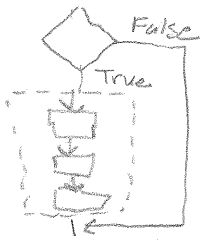
Selection

```
if (x > 5)
  printf ("x = %d \n");

if (x > 5)        A
  printf("x = %d \n");
else              B
  printf ("x is a small value");
```





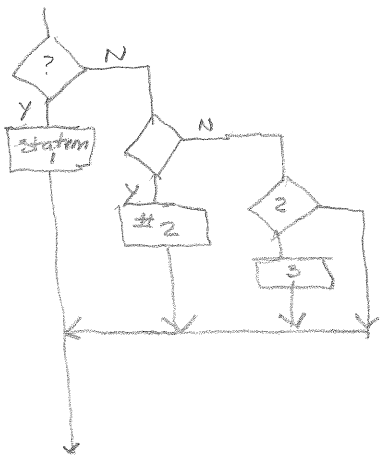


← these can be a block of statements including nested control statement blocks

switch

```
if (condition 1)
   statement 1
else
   if (condition 2)
         statement 2
   else if (condition 3)
         statement 3
```

```
switch (value)        ← integral type or character
{
 case '1':
    actions
 case '2':
    actions
 default :
    actions
}
```



CS 350 Lecture #5 Tuesday, January 25, 2010

Topics: Structured Programming
- Selection
- Repetition

Selection Structures
→ if
→ if else
→ switch case

switch value must be an integer
default can cover unusual situations but is not required.

case 'B': case 'b': colon is part of syntax
switch case are more useful if there are seven or more different situations

Errors in using selection Dork

Basic type of errors

→ Syntax
language error → caught by compiler

→ Runtime error
Divide by zero
Pointer error
> however don't count on C to help you
usually terminate the programs

→ Logic errors

```
if (grade == 'A')
    printf ("Move to the next class");
    printf ("You did very well");
if (grade == 'F')
    printf ("Move to next class");
    printf ("you did very well
```

this will print for everybody without braces →

lack of braces for multiple statements

← Meaning different than intended.

Another example

```
if (x>5)
    printf("A");
    printf("B");

else
    printf("C");
    printf("D");
```

← "The Dangling Else Problem"
This is syntax

← This is logic error

Another example

```
int a;
a = -5;
if(a)
  print ("D")
else
  print ("E")

if (a=5)
  printf("C");
else printf ("F");
```

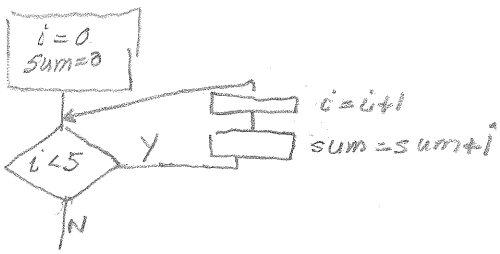
as long as a ≠ 0, it is considered true.
no boolean!

- Repetition structure
  - while
  - for
  - do/while

4 key components of a Repetition
- → initialization
- → loop condition
- → Update of the loop control variable
- ↪ Body of loop
- * termination of loop

while loop

```
int i=0;
int sum=0;
 while (i<5)
  {
  sum = sum + i;
  i=i+1;
  }
  printf ("i= %d\n", i);
```



Goes through the loop 5 times

i = 0, 1, 2, 3, 4

- Counter-controlled Loop
- Sentinel-controlled loop
  "special value" ⇒ use of sum in the above while loop

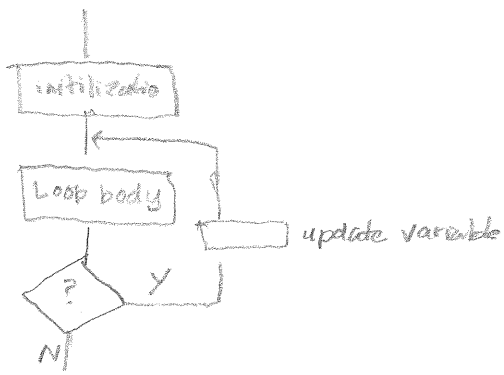
```
int score;
  :
scanf ("%d", &score);
  :
while (score != -99)
  {printf (.....)
   Scanf ("%d", &score)
  }
```

use while for sentinel control

use for loop for counter control

for loop

```
sum=0;
 for (i=0; i<5; i=i+1)
   sum=sum+i;
```




```
do
{ sum = sum+i;
   i=i+1;
} while (i<5);
```

do/while

menu system

Common error in repetition structures

→ one-off error → is your initialization correct?

→ no loop.

```
for (i=10, i<=5; i++)    i > 5
  {
  ...
  }
```

→ infinite loop

```
for(i=10; i>5; i++)
  {
  }
```

→ empty condition

for (i=i) Empty for is evaluated as positive

for (for ...); ← sits there forever

Thursday, January 28, 2009

Functions

→ Summary of Structured Programming Development
→ Functions of Black Box
→ Functions as a small program
→ Scope Rules

C-Idioms see 4.1

i++	++i
i--	--i

a = a+2 → a+=2

Functions (called Methods in Java)

Purpose is construct a program from smaller pieces

Functions as a black box

→ Saves time
→ programs may be easier to understand
→ Could save us from errors / bugs
→ Use over from programs

Library Files

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

format for calling functions

FunctionName (argument)

Math functions return double data type

Functions as a glass box

1. Prototype declaration
2. Function Call
3. Function Definition

Function Prototypes

```
float computeAverage (int n)
int computeProduct (int a, int b)
void myPrintf (void)
void myFn (int x, double y, int y)
with
```

Function definition:

```
int computeProduct (int a int b
  }
   int c;
     c = a*b;
     return c;
   }
```

Function Call

```
int main( )
  {
   int a,b,c;
   =
   c= computeProduct
   =
  }
```

N = number
O = order
T = type

Another example

```
<stdlib.h>
```

rand function:
Returns a value between 0 and Rand_MAX
up to 32767

Scaling
- to get a random # between 1 and n

```
1+ (rand () %n);
```

C produces "pseudo-random method
- rand(seed)

for the purists → there is no true random number generator

Storage Classes
storage duration - how long exist in memory
Scope - where object can be referenced

Automatic storage
Object created & destroyed within its block

```
auto double x, y;
```

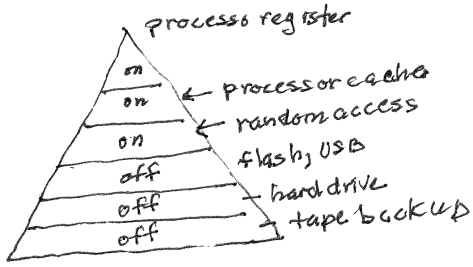
- registered
  tries to put variable into high-speed registers

Tuesday, February 2, 2010

$X_{n+1} = (a X_n + b) \% m$ (random number generator)

. "Random-like" sequence by numbers
pseudo-random numbers

bit-wise operations has a random number



auto: d.

register: tries to:
RAID ~ power off (mid + long term)
Permanence of storage
access speed
cost per unit of memory
memory size/capacity

static variable → memory last until programs finished
extern:

Storage classes - scope rules
global - known in any function

File scope - can reference anywhere in file → global variables
Function scope - local variable - only used inside the function
Block scope -

```
int fnz (int x).
  { int y;
    {
      int y;
    }
  printf ("%d", y)
```

Function prototypes

```
int fn (int x); ≡ in fn (int)
int fn1 (int x);
```

Example

```
int i;
void f(int i);
  { i = 1
  }
void g(void)
  { int = 2
    if (i > 0)
      {int i;
       i = 3;
      }
  }
```

function
block

```
global void h (void)
  {
   i = 5;
  }
```

```
int i;
void printRow (void)
  {
   for (i=1; i<=10; i++)
       printf ("*")
   }

  void printMatrix (void)
   {
    for (i=1, i≤10 ; i++)
        { printRow()
          printf ("\n")
         }
   }
```

Prints 1 row of 10**
Because of global variable

Recursive Functions

Function that calls itself
Can only solve a base case
Figure out the base case

4 Important Considerations for Recursive Methods

1. Ending condition - Base
2. The recursive call
3. Body of recursion
4. Update of recursion variable

$n! = n * (n-1) * (n-2) * \ldots 1$

$= n * (n-1)!$

Define base case

```
int myFactorial (int n)
  {
  if n==1 (or n==0)
      return 1

  else
      return n* myFactorial (n-1)   ← recursive call
  }
```

myFactorial (3)

n → myFactorial (2) → 2*1

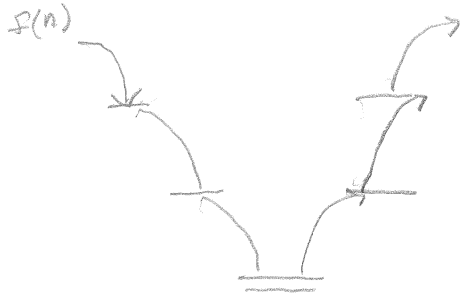
→ myFactorial (1) → 1*1

$f_n = f_{n-1} + f_{n-2}$

$f_0 = 0$
$f_1 = 1$ } base case

```
int myFibonacci(int n)
    if (n==0)
      return 0;
    else if
      (n==1)
        return 1;
    else
      return myFib(n-1) + myFib(n-2)
```

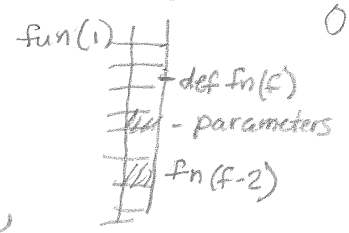
f(5)
↙




```
int f(int i)
{
  int i
  double x;
  double d[1000]
}
```

$O(2^n) \rightarrow O(n)$

$2^n * \frac{8K}{1\,bit}$



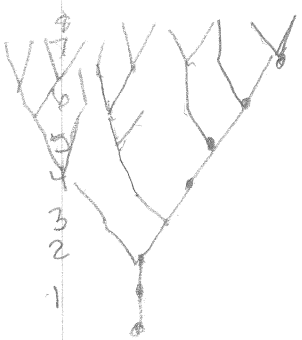
Thursday, February 4, 2010 Lecture # 08

Topics - More on recursion
Arrays:
→ definition + Initialization
→ array of characters
→ Arrays in functions
→ Sorting + Searching

Recursion vs Iteration:
Reproduction
- Iteration, explicit loop - save memory
- Recursion - repeated function calls - could require a lot of memory
Difficult to write iterative program for Towers of Hannoi

population growth, real estate

Study growth of plants → use of Fibonacci number
Growing trees with the Fibonacci
Two Rules:
1. At each step, each branch grows a new branch
2. However, no two siblings can grow branches at the same time



Review of quiz -

Arrays

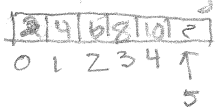
must be all of the same type
must remain the same size throughout the program

arrayname [ ] 0 → n-1 n = size of arrays

C allows you to have

Array declaration

int aa[5];
int bb[5] = {2,4,6,8,10} bb= 
i = bb[2]
x = bb[5]; ← Java would give compile error, but you will get a run-time error



Arrays can also be a global

#define N=5;

you can initialize

#define N=5;
int num [5] = {-3,2} all rightmost will be zero


```
int i, x;
int bb[5] = {2,4,6,8,10};
i = bb[2]; //i = 6
x = bb[i-3];
bb[i++];
```

you cannot put a double number into an int array

if you try to put a double → run-time error

Figure 6.8 was last printout

Topics
- Array of char
- Arrays in a function
- Sorting + Searching
- 2D Arrays
- Note on Assignment 2

Strings in Java are known as character arrays
To note the end of a string use the null character '\0'

```
char chArray[4] = "the";   ← the
char chArray2[4] = {'t', 'h', 'e', 'm'};   ← them
char chArray3[4] = {'t', 'h', 'e', '\0'}
```

```
chArray5[6] = {"This is OK"};
```

modify the Array

```
chArray5[6] = '\0';
printf("My array is %s\n", chArray);
```

Output will be This

```
scanf("%s", string);
```

Do not need to put & before arrayname ie

for characters you use "%c" and you must use the ampersand in scanf

Arrays in Functions

Prototype:
```
int sumInts(int a[], int N);
```

Function definition
```
int sumInts(int a[], int n)
{
 int i, sum;
   for(i=0; sum=0; i<n; ++i)
       sum = sum a[i];
   return sum
```

function call
```
int main ()
{
 int s;
 int aa[5];
 s = sumInts(aa, 5);
```
(no brackets)

when passing a single element

The element of an Array does not change permenantly.

Sorting

Bubble sort - Small elements "bubble" to the top

	Pass 1	Pass 2
3	3	2
4	2	3
2	4	4
6	6	6
7	7	7

Call-by-reference

```
void swapA (int a, int b)
{
 int temp;
 temp = a;
  a = b;
  b = temp
 // printf (a,b)

}
int main ()
 {
  int aa [5] = {1,2,3,4,5}
     swapA (aa[1], aa[3]);
  // Result aa = [1][2][3][4][5]
     swapB (aa, 1, 3);
  // Result : aa [1][4][3][2][5]
 }
```

```
void swapB (int a[], i, j)
 {
 int temp;
 temp = a[i];
 a[i] = a[j];
 a[j] = temp;
  // printf (ai, aj)
 }
```

Read up on linear search and binary search on your own

2-D Arrays

m x n rows x columns

Declaration

```
int num [3][4];
int values [3][4] = { {1,2,3,4}, {5,6,7,8}, {9,10,11,12} };
int newArrary [3][4] = { {1,2}, {3,4}, {5} }
int x;
x = newArray [1][2];
newArray [2][2] = **20;
```

1	2	3	4
5	6	7	8
9	10	11	12

1	2	0	0
3	4	0	0
5	0	0	0

Arrays in functions

Define cSIZE 4

Prototype: int sum2D (int aa [][cSIZE], int n);

Definition int sum2D (int aa[ ][cSIZE], int n);

Function call

```
int main ( )
 {
  int bb [N][M];
  int s;
   s = sum2D (bb, N);
 }
```

saved in row-order

1	2	3	4
0	5	6	7
3	2	1	8

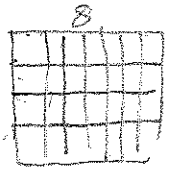
→

define K ...
define N ...
define M ...

int aa[N][M][K];
x = aa[i][j][k];

Knight's Movement Problem

heuristic



64 positions

Knight will moves either 2 to right
1 to down

Fill up a table with the number of different moves able to move from in that space

Accessibility

Have the knight move to every square
start at the positions with lowest number of moves availability

once you are in the new position - move to the block with least number of available moves

Brute force

how to compile

$ gcc MyProgram.c -o myProgram

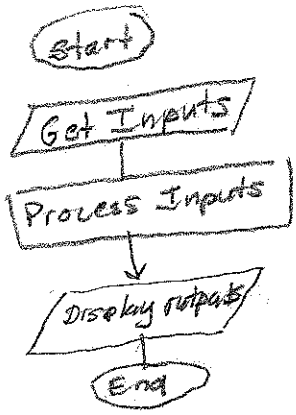
$ ./myProgram

Tuesday, February 16, 2008

** Test on Thurs next week

Pointers - Intro
Pointer Distributor
Pointer Operations
Call by Reference
Const Quantifiers
Quiz

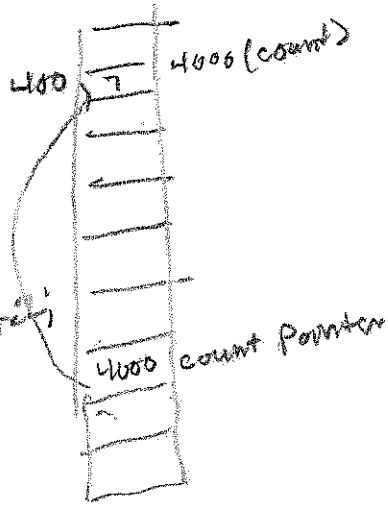
Documentation for assignment
1. Problem definition/specification
2. Problem analysis
   a. Identify the inputs
   b. Identify the outputs
   c. Identify other constraints
   d. relate inputs to outputs
3. Algorithm Design
   - stepwise refinement
   Flowchart (not necessary
   otherwise - write
   second level of refinement
   1. Get Inputs
      1.1 ...
      1.2 ...



you do not need to use #flow chart

Pointers

The efficiency of C comes from the use of pointers
1. Powerful - but difficult to master
   Simulate call-by-reference
   Close relationship w/ arrays & strings
The value you get will be the pointer
Pointers contain address



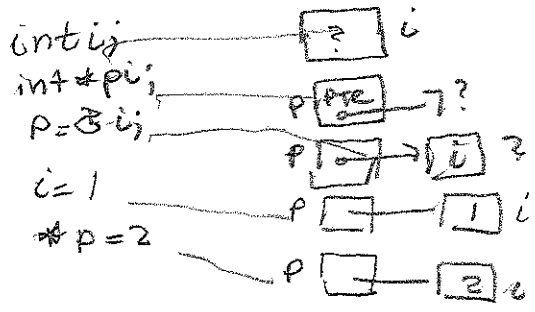
int count;
int *countPointer;
count = 7;
countPtr = &count;
dcount ptr = count
Pointers (* name ptr)
&yptr → * (yptr) →
returns address of y

Two pointers
int *myPtr1, *myPtr2;

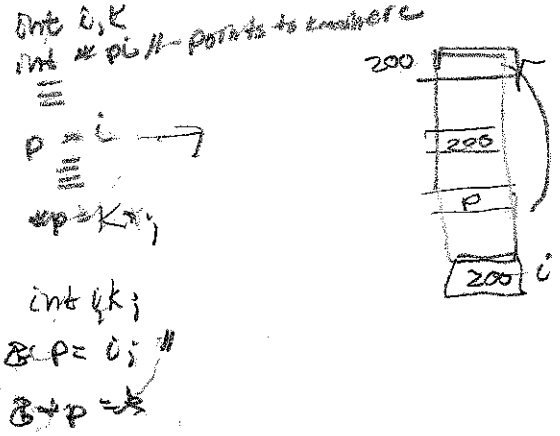
Can declare
int y = 5
int *yPtr;

↑ previous
yptr = &y / get address of y

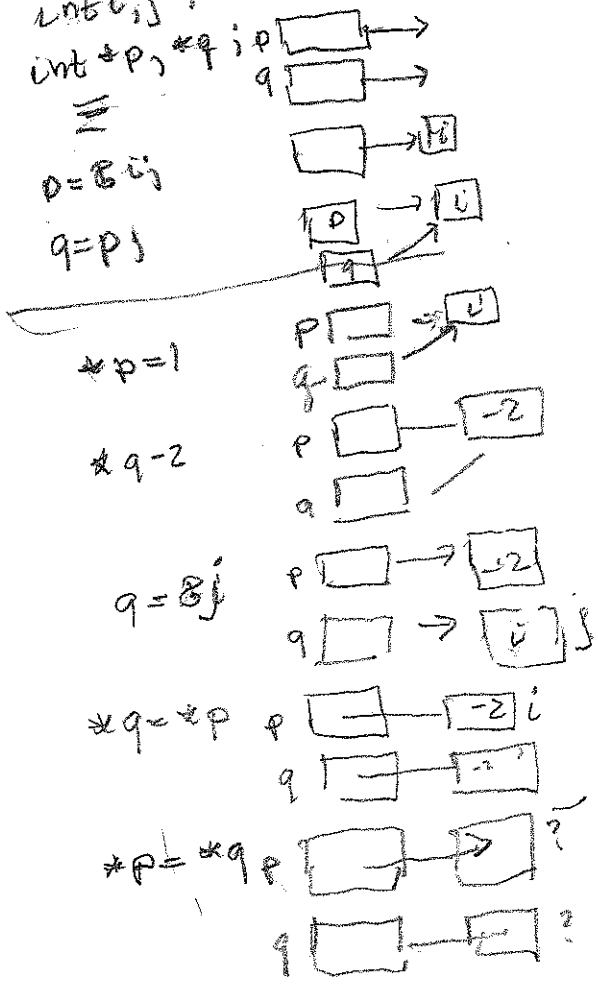
Pointer examples



Example



Pointer assignment



Notes

p=q; ≠ *p=*q;

q=&p=p ← this doesn't work

Call by reference with pointers

```
void swap (int *b, int *a)
{int temp;
 temp = *a
 *a = *b
 *b = temp;
```

Call in main

```
int main()
{
 int x, y;
swap (&x, &y);
```

To Compile

```
$ gcc MyProgram.c -o myProgram
```

To Run

```
$ ./MyProgram
```

Pointers (con'td)
- const Qualifiers
Pointers + Arrays
Pointer expressions
2D Arrays
Array of Pointer
Pointers to Functions

To run program
$ gcc MyProgram.c -o myProgram
$ ./myProgram

```
int main ()
{
 int x=7
 =
 doubleX = ( )
 =
return
```

prototype → void double (double *b);

Const qualifier - variable can not be changed
Attempted to change a
You must initialize it when you declare it

4 ways to use a const qualifier

1. const int i=30; constant integer Keeps value the entire program
2. const int * p = &i;
   pointer to a constanst
   b can be changed, but not via *p
3. int * const p = &a;
   Constant pointer to an int change value, but not location
   *p can be changed, but P cannot
4. const int * const p = &c

   const pointer to constant int
   c can be changed, but no via *p
   p cannot be changed or re-directed

sizeof (int) = 4

bubbleSort (a, size)

Arithmetic Expressions

p++

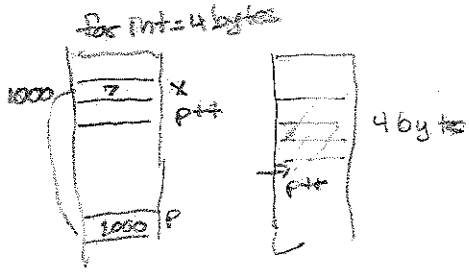
pointer comparison

<, ==, >

int x;
int *p = &x;

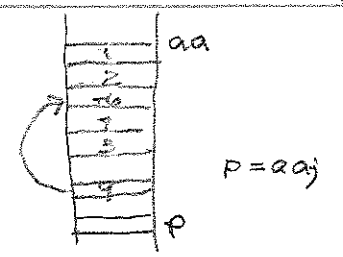
int aa[5];
p = aa

for int = 4 bytes



Arrays vs Pointers

```
int aa[5] = {1, 2, -6, 7, 3}
int i, *p, *q
aa = &i
q = &aa[2]
p = aa;
if (*q == *p)
```

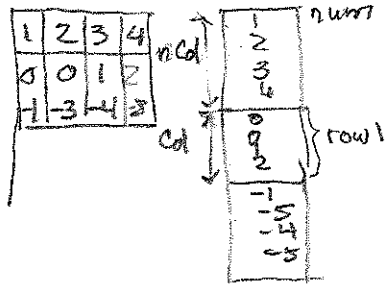


q[0] = -6

x = q = [1] // x = 7
y = p = [2] // y = -6
x = q-- // y = x = -6

2-D Arrays + Pointers

```
int num[3][4] = {{1,2,3,4},{0,0,1,2},{-1,-3,-4,-5}}
```




```
p = num; // same as p = &num[0]
y = num[i][j]; y = p[i*4+j]
x = num[0][1] // x = p[1]
```

CS 350 L#12 Tuesday, February 23, 2010

Topics - More on pointers
- Array of Pointers
- Pointers and Functions
- Pointer and strings
- Review for Test 1

Some Confusing Pointer Problems

int *p;

*p++ ≡ *(p++) → value of expression is *p but increment; increment p later

*++p ≡ *(++p) → increment p first; value of expression is *p after increment

(*p)++ → value of expression is next value 30 → 31, but doesn't change p

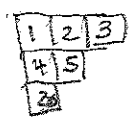
Arrays of Pointers

jagged arrays →

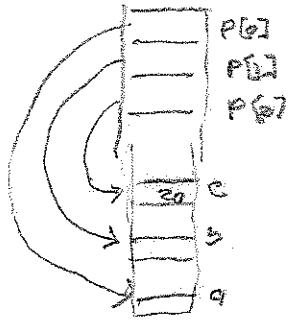
Strings are Array of Characters

char *suit[4] = {"Hearts", ...}

```
int a[] = {1,2,3}, b[] = {4,5};
    c[] = {20};
int *p[3]
≡
p[0] = a; // p[0] = &a[0]
p[1] = b; // p[1] = &b[0]
p[2] = &c[0] //
```



p[0][1]
↳ a=[1]



$\int_a^b f(x)$

integrate(f, x)

integrate(sine, x)

Also use in sorting

Pointers to Functions

Prototype declaration ⇒ void bigFunction(int bb[], const int n,
≡ int (*smallFunction)(int i, int j));
int smallFunction1(int a, int b);
int smallFunction2(int a, int b);

```
#define N 5
≡
int main()
{
  int x, y;
  int aa[N]
```

```
bigFunction(aa, N, smallFunction1);
bigFunction(aa, N, smallFunction2);
≡
return 0;
```

⇒

Function Definition

```
void bigFunction (int b[], const int n, int (*smallFunction)(int i, int j))
{
  int a,b,c,d;
  =
  c= (*smallFunction)(i,j);
  d=(*smallFunction)(a,b);
```

* Read chapter 8 on your own *

- Basic intro to C
  - identifiers, data types, etc
  - Structured Programming in C
    - sequential
    - selection
    - repetition
  - Functions
  - Arrays
  - Pointers
  - Flowcharts

Content

→ conceptual / theory
→ Results of Program Fragments
→ Errors in Programs / or Program segments
→ Program writing
→ T/F