CS3157: Advanced Programming

Lecture #5

June 7

Shlomo Hershkop

shlomo@cs.columbia.edu

1

Announcements

- Wednesday is pizza day
  - bring your appetite
- When you leave me AIM's and I'm away from my desk, please identify yourself…
- How are the labs coming along ?
- Homework DUE!!!
  - lets talk!

2

Outline

- More c
  - Preprocessor
  - Bitwise operations
  - Character handling
  - Math/random
  - Pointers
  - Const
  - Typedef
  - Union
  - Enum
- Reading:
  - see website (Deitel chapter 7)
  - k&r ch chapter 4, 5.5-,6

3

Remember

- unlike high level languages, you are now getting much more control over the computer
  - but this also gives you much more chances to mess it up ☺
  - lots of control on how your code will work
- think of the difference between driving a car and repairing it

4

From last time

- the C pre-processor (cpp) is a macro-processor which
  - manages a collection of macro definitions
  - reads a C program and transforms it
- pre-processor directives start with # at beginning of line used to:
  - include files with C code (typically, "header" files containing definitions; file names end with .h)
  - define new macros (later – not today)
  - conditionally compile parts of file (later – not today)
- gcc -E shows output of pre-processor
- can be used independently of compiler

5

Pre-processor cont.

```
#define name const-expression
#define name (param1,param2,...) expression
#undef symbol
```

- replaces name with constant or expression
- textual substitution
- symbolic names for global constants
- in-line functions (avoid function call overhead)
- type-independent code

```
#define MAXLEN 255
```

6

Example

```
#define MAXVALUE 100
#define check(x) ((x) < MAXVALUE)
if (check(i)) { ...}
```

- becomes

```
if ((i) < 100) {...}
```

- Caution: don't treat macros like function calls

```
#define valid(x) ((x) > 0 && (x) < 20)
```

- is called like:

```
if (valid(x++)) {...}
```

- and will become:

```
valid(x++) -> ((x++) > 0 && (x++) < 20)
```

- and may not do what you intended...

7

- conditional compilation

- pre-processor checks value of expression
- if true, outputs code segment 1, otherwise code segment 2
- machine or OS-dependent code

- can be used to comment out chunks of code— bad!
- (but can be helpful for quick and dirty debugging :-)

- example:

```
#define OS linux
...
#if OS == linux
puts( "Wow you are running Linux!" );
#else
puts( "why are you running something else???" );
#endif
```

8

- ifdef
- for boolean flags, easier:

```
#ifdef name
code segment 1
#else
code segment 2
#endif
```

- pre-processor checks if name has been defined, e.g.:

```
#define USEDB
```

- if so, use code segment 1, otherwise 2

9

From last time

- We covered basic types

- int x = 100;
- int a[100];

- int b[20][34]

- int *c;

10

Function

- Declaration:
  - `Return-type function-name (parameters if any);`
- Definition:
  - `Return-type function-name (parameters if any){`

```
declarations


statements
}
```

- how does `main` fit in ?

11

Command Line Args

```
int main( int argc, char *argv[] )
```

- argc is the argument count
- argv is the argument vector
  - array of strings with command-line arguments
- the `int` value is the return value
  - convention: return value of 0 means success,
  - > 0 means there was some kind of error
  - can also declare as `void` (no return value)

12

- Name of executable followed by space-separated arguments

```
$ a.out 1 23 "third arg"
```

- this is stored like this:

```
1. a.out
2. 1
3. 23
4. “third arg”
•  argc  = 4
```

13

- If no arguments, simplify:

```
int main() {
printf( "hello world" );
exit( 0 );
}
```

- Uses exit() instead of return() — almost the same thing.

14

booleans

- C doesn't have booleans
- emulate as int or char, with values 0 (false) and 1 or non-zero (true)

- allowed by flow control statements:

```
if ( n == 0 ) {
printf( "something wrong" );
}
```

- assignment returns zero -> false
- you can define your own boolean:

```
#define FALSE 0
#define TRUE 1
```

15

Booleans II

- This works in general, but beware:

```
if ( n == TRUE ) {
printf( "everything is a-okay" );
}
```

- if n is greater than zero, it will be non-zero, but may not be 1; so the above is NOT the
- same as:

```
if ( n ) {
printf( "something is rotten in the state of denmark" );
}
```

16

Logical operators

- in C logical operators are the same as in Java
- meaning C operator
- AND &&
- OR ||
- NOT !

- since there are no boolean types in C, these are mainly used to connect clauses in if and while statements
- remember that
  - non-zero == true
  - zero == false

17

Bitwise operators

- there are also bitwise operators in C, in which each bit is an operand:
- Meaning c operator
- bitwise AND &
- bitwise or |
- Example:

```
int a = 8; /* this is 1000 in base 2 */
int b = 15; /* this is 1111 in base 2 */
```

- a & b = $\frac{1000(8) \;\&\; 1111(15)}{1000(=8)}$ a | b= $\frac{1000(8) \;|\; 1111(15)}{1111(=15)}$

18

Question

- what is the output of the following code fragment?
- int a = 12, b = 7;
- printf("a && b = %d\n", a && b);
- printf("a || b = %d\n", a || b);
- printf("a & b = %d\n", a & b);
- printf("a | b = %d\n", a | b);

19

Implicit convertions

- implicit:

```
int a = 1;
char b = 97; // converts int to char
int s = a + b; // adds int and char, converts to int
```

- promotion: char -> short -> int -> float -> double
- if one operand is double, the other is made double
- else if either is float, the other is made float

```
int a = 3;
float x = 97.6;
double y = 145.987;
y = x * y; // x becomes double; result is double
x = x + a; // a becomes float; result is float
```

- real (float or double) to int truncates

20

explicit

- explicit:
- type casting

```
int a = 3;
float x = 97.6;
double y = 145.987;
y = (double)x * y;
x = x + (float)a;
```

- – using functions (in math library...)

1. floor() – rounds to largest integer not greater than x
2. ceil() - round to smallest integer not smaller than x
3. round() – rounds up from halfway integer values

21

Example

```
#include <stdio.h>
#include <math.h>
int main() {
int j, i, x;
double f = 12.00;
for ( j=0; j<10; j++ ) {
i = f;
x = (int)f;
printf( "f=%.2f i=%d x=%d
floor(f)=%.2f ceil(f)=%.2f round(f)=%.2f\n",
f,i,x,floor(f),ceil(f),round(f) );
f += 0.10;
} // end for j
} // end main()
```

22

Output

- f=12.00 i=12 x=12 floor(f)=12.00 ceil(f)=12.00 round(f)=12.00
- f=12.10 i=12 x=12 floor(f)=12.00 ceil(f)=13.00 round(f)=12.00
- f=12.20 i=12 x=12 floor(f)=12.00 ceil(f)=13.00 round(f)=12.00
- f=12.30 i=12 x=12 floor(f)=12.00 ceil(f)=13.00 round(f)=12.00
- f=12.40 i=12 x=12 floor(f)=12.00 ceil(f)=13.00 round(f)=12.00
- f=12.50 i=12 x=12 floor(f)=12.00 ceil(f)=13.00 round(f)=12.00
- f=12.60 i=12 x=12 floor(f)=12.00 ceil(f)=13.00 round(f)=13.00
- f=12.70 i=12 x=12 floor(f)=12.00 ceil(f)=13.00 round(f)=13.00
- f=12.80 i=12 x=12 floor(f)=12.00 ceil(f)=13.00 round(f)=13.00
- f=12.90 i=12 x=12 floor(f)=12.00 ceil(f)=13.00 round(f)=13.00

Be aware

- almost any conversion does something— but not necessarily what you intended!!
- – example:

```
int x = 100000;
short s = x;
printf("%d %d\n", x, s);
```

- – output is:

```
100000 -31072
```

- WHY?

math library

- Functions ceil() and floor() come from the math library
- definitions:
  - ceil(x): returns the smallest integer not less than x, as a double
  - floor(x): returns the largest integer not greater than x, as a double
- in order to use these functions, you need to do two things:

1. include the prototypes (i.e., function definitions) in the source code:
   #include <math.h>
2. include the library (i.e., functions' object code) at link time:
   unix$ gcc abcd.c -lm

- exercise: can you write a program that rounds a floating point?

25

math

- some other functions from the math library (these are function prototypes):
  - double sqrt(double x);
  - double pow(double x, double y);
  - double exp(double x);
  - double log(double x);
  - double sin(double x);
  - double cos(double x);

- exercise: write a program that calls each of these functions

- questions:
  - can you make sense of /usr/include/math.h?
  - where are the definitions of the above functions?
  - what are other math library functions?

26

Random numbers

- with computers, nothing is random (even though it may seem so at times...)

- there are two steps to using random numbers in C:

1. seeding the random number generator
2. generating random number(s)

- standard library function:

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

- seed function:

```
srand( time ( NULL ));
```

- random number function returns a number between 0 and RAND_MAX (which is 2^32)

```
int i = rand();
```

27

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
int main( void ) {
int r;
srand( time ( NULL ));
r = rand() % 100;
printf( "pick a number between 0 and
  100...\n" );
printf( "was %d your number?", r );
}
```

28

Character handling

- character handling library

#include <ctype.h>

- digit recognition functions (bases 10 and 16)
- alphanumeric character recognition
- case recognition/conversion
- character type recognition

- these are all of the form:

```
int isdigit( int c );
```

- where the argument c is declared as an int, but it is interpreted as a char
- so if c = ’0’ (i.e., the ASCII value ’0’, index=48), then the function returns true (non-zero int)
  but if c = 0 (i.e., the ASCII value NULL, index=0), then the function returns false (0)

29

digits

- digit recognition functions (bases 10 and 16)

```
int isdigit( int c );
```

- returns true (i.e., non-zero int) if c is a decimal digit (i.e., in the range ’0’..’9’);
  returns 0 otherwise

```
int isxdigit( int c );
```

- returns true (i.e., non-zero int) if c is a hexadecimal digit (i.e., in the range ’0’..’9’,’A’..’F’); returns 0 otherwise

30

Alpha numeric

- alphanumeric character recognition

```
int isalpha( int c );
```

- returns true (i.e., non-zero int) if c is a letter (i.e., in the range ’A’..’Z’,’a’..’z’); returns 0 otherwise

```
int isalnum( int c );
```

- returns true (i.e., non-zero int) if c is an alphanumeric character (i.e., in the range ’A’..’Z’,’a’..’z’,’0’..’9’); returns 0 otherwise

31

Case

- case recognition

```
int islower( int c );
```

- returns true (i.e., non-zero int) if c is a lowercase letter (i.e., in the range ’a’..’z’); returns 0 otherwise

```
int isupper( int c );
```

- returns true (i.e., non-zero int) if c is an uppercase letter (i.e., in the range ’A’..’Z’); returns 0 otherwise

- case conversion

```
int tolower( int c );
```

- returns the value of c converted to a lowercase letter (does nothing if c is not a letter or if c is already lowercase)

```
int toupper( int c );
```

- returns the value of c converted to an uppercase letter (does nothing if c is not a letter or if c is already uppercase)

32

types

- character type recognition

```
int isspace( int c );
```

- returns true (i.e., non-zero int) if c is a space; returns 0 otherwise

```
int iscntrl( int c );
```

- returns true (i.e., non-zero int) if c is a control character; returns 0 otherwise

```
int ispunct( int c );
```

- returns true (i.e., non-zero int) if c is a punctuation mark; returns 0 otherwise

```
int isprint( int c );
```

- returns true (i.e., non-zero int) if c is a printable character; returns 0 otherwise

```
int isgraph( int c );
```

- returns true (i.e., non-zero int) if c is a graphics character; returns 0 otherwise

33

Header files

- .h files usually used to define methods or centralize defintions

- public int calculateSomething(int []);

- Can either name the variables or not
- int[] vs int ar[]
- In .c file use; #include “something.h”

34

compilation

- Remember to make sure you have all your files when you split them between .c and .h
- You include the .c files for compilation and the compiler will find the .h files.
- Object files unchanged.

35

Outline

- Arrays
- Pointers
- Memory allocation
- functions
- function arguments
- arrays and pointers as function arguments

- Reading
  - Chapter 5,6-6.3

36

Arrays again

- Arrays and pointers are strongly related in C

```
int a[10];
int *pa;
```

- (remember that &a[0] is the address of the first element in a, that is the beginning of the array

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

- pointer arithmetic is meaningful with arrays:
- if we do

```
Pntr = &a[0]
```

- then

```
*(Pntr +1) =
```

- Is whatever is at a[1]

37

- There is a difference between
  - *(Pntr) + 1
  - and (*Pntr +1)

- Note that an array name is a pointer, so we can also do *(a+1) and in general: *(a + i) == a[i] and so are a + i == &a[i]
- The difference:
  - an array name is a constant, and a pointer is not
  - so we can do: Pntr = a and Pntr ++
- But we can NOT do: a = Pntr or a++ pr or Pntr = &a
- That is you can not reassign it as a pointer

38

Note

- When an array name is passed to a function, what is passed is the beginning of the array, that is passed by reference
- It is important, since this is an address, any changes to that memory location will stick when you come back from the function

39

From last time

- a pointer contains the address of an object (but not in the OOP sense)
- allows one to access object “indirectly”
- & = unary operator that gives address of its argument
- * = unary operator that fetches contents of its argument (i.e., its argument is an address)
- note that & and * bind more tightly than arithmetic operators
- you can print the value of a pointer with the formatting character %p

40

code

```
#include <stdio.h>
main() {
  int x, y;     // declare two ints
  int *px;      // declare a pointer to an int
  x = 3;        // initialize x
  px = &x;
  y = *px;
 printf( "x=%d px=%p y=%d\n",x,px,y );
}
```

41

Memory allocation

- One of the main advantage to c/cpp is that you can manipulate memory yourself (and are responsible to clean up after yourself.
- When you don’t it is called memory leaking…more on this later

42

Array vs memory allocation

- Arrays are great when you have a rough idea of how many items you will be dealing with
  - 10 numbers
  - 30 students
  - Less than 256 characters of input

43

Map of memory

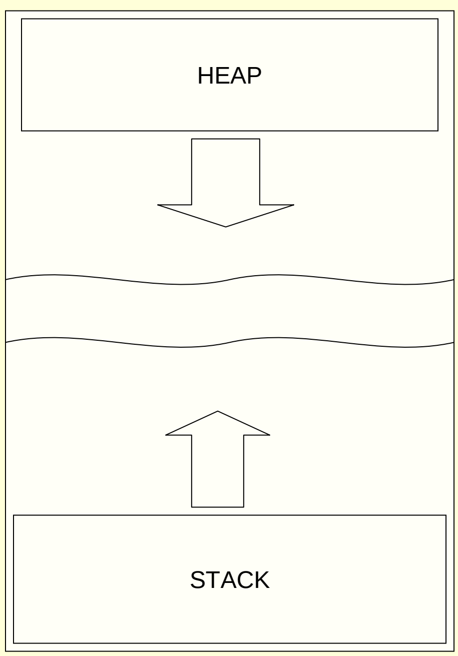
- Think of memory as a box
- Main is placed on the bottom and any variable on top of that
- Any function call gets placed on top of that
- This part of memory grows upward
- It is called the stack
- Your program is over when the stack is empty

44

heap

- The heap is the other side of memory
- Global variables, and allocated memory is created on the heap
- It grows downwards

45



46

Dynamic Memory Allocation

- pre-allocated memory comes from the “stack”
- dynamically allocated memory comes from the “heap”
- To get memory you allocated (malloc) memory, and to let it go, you free it (free)
- family of functions in stdlib, including:

```
void *malloc( size_t size );
void *realloc( void *ptr, size_t size
  );
void free( void * );
```

47

- malloc and realloc return a generic pointer (void *) and you have to “cast” the return to the type of pointer you want
- That is if you are allocation a bunch of characters, you say
- Ptr = (char*) malloc….

48

Malloc.c

```
#include <stdio.h>
#include <stdlib.h>
#define BLKSIZ 10
main() {
  FILE *fp;
  char *buf, k;
  int  bufsiz, i;
  // open file for reading
  if (( fp = fopen( "myfile.dat","r" )) == NULL ) {
    perror( "error opening myfile.dat" );
    exit( 1 );
  }
  // allocate memory for input buffer
  bufsiz = BLKSIZ;
  buf = (char *)malloc( sizeof(char)*bufsiz );
```

49

II

```
// read contents of file
  i = 0;
  while (( k = fgetc( fp )) != EOF ) {
    buf[i++] = k;
    if ( i == bufsiz ) {
      bufsiz += BLKSIZ;
      buf = (char *)realloc( buf,sizeof(char)*bufsiz );
    }
  }
  if ( i >= bufsiz-1 ) {
    bufsiz += BLKSIZ;
    buf = (char *)realloc( buf,sizeof(char)*bufsiz );
  }
  buf[i] = '\0';
  // output file contents to the screen
  printf( "buf=[%s]\n",buf );
  // close file
  fclose( fp );
} // end main()
```

50

Dynamic memory

- malloc() allocates a block of memory:

```
void *malloc( size_t size );
```

- lifetime of the block is until memory is freed, with free():

```
void free( void *ptr );
```

- example:

```
int *dynvec, num_elements;
printf( "how many elements do you want to enter? " );
scanf( "%d", &num_elements );
dynvec = (int *)malloc( sizeof(int) * num_elements );
```

51

Memory leaking

- memory leaks— memory allocated that is never freed:

```
char *combine( char *s, char *t ) {
u = (char *)malloc( strlen(s) + strlen(t) + 1 );
if ( s != t ) {
strcpy( u, s );
strcat( u, t );
return u;
}
else {
return 0;
}
} /* end of combine() */
```

- u should be freed if return 0; is executed
- but you don't need to free it if you are still using it!

52

Example 2

```
int main(void) {

  char *string1 = (char*)malloc(sizeof(char)*50);
  char *string2 = (char*)malloc(sizeof(char)*50);
  scanf("%s",string2);
  string1 = strong2;  //MISTAKE THIS IS NOT A COPY

  ...
  free(string2);
  free(string1); ///????

  return 0
  }
```

53

Memory leak tools

- Purify
- Valgrind
- Insure++
- Memwatch (will use it in lab)
- Memtrace
- Dmalloc

54

Dynamic memory

- note: malloc() does not initialize data, that is you have garbage there with whatever was there in memory
- you can allocate and initialize with “calloc”:

void *calloc(size_t nmemb, size_t size);

  - calloc allocates memory for an array of nmemb elements of size bytes each and returns a pointer to the allocated memory. The memory is set to zero.

- you can also change size of allocated memory blocks with “realloc”:

void *realloc(void *ptr, size_t size);

  - realloc changes the size of the memory block pointed to by ptr to size bytes. The contents will be unchanged to the minimum of the old and new sizes; newly allocated memory will be uninitialized.

- these are all functions in stdlib.h
- for more information: `man malloc`

55

Dynamic arrays

- “arrays” are defined by specifying an element type and number of elements
  - statically:

```
int vec[100];
char str[30];
float m[10][10];
```

  - dynamically:

```
int *dynvec, num_elements;
printf( "how many elements do you want to enter? " );
scanf( "%d", &num_elements );
dynvec = (int *)malloc( sizeof(int) * num_elements );
```

- for an array containing N elements, indeces are 0..N-1
- stored as a linear arrangement of elements
- often similar to pointers

56

Dynamic arrays II

- C does not remember how large arrays are (i.e., no length attribute, unlike Java)
- given:

```
int x[10];
x[10] = 5; /* error! */
```

- ERROR! because you have only defined x[0]..x[9] and the memory location where x[10] is can become something else...

- sizeof x gives the number of bytes in the array
- sizeof x[0] gives the number of bytes in one array element
- You can compute the length of x via:

```
int length_x = sizeof x / sizeof x[0];
```

57

Arrays cont.

- when an array is passed as a parameter to a function:
  - The size information is not available inside the function, since you are only passing in a start memory location
  - array size is typically passed as an additional parameter

```
printArray( x, length_x );
```

  - or globally

```
#define VECSIZE 10
int x[VECSIZE];
```

58

arrays

- array elements are accessed using the same syntax as in Java: array[index]
- C does not check whether array index values are sensible (i.e., no bounds checking)
- e.g., x[-1] or vec[10000] will not generate a compiler warning!
- if you're lucky, the program crashes with Segmentation fault (core dumped)

59

Dynamically allocated arrays

- C references arrays by the address of their first element
- array is equivalent to &array[0]
- you can iterate through arrays using pointers as well as indexes:

```
int *v, *last;
int sum = 0;
last = &x[length_x-1];
for ( v = x; v <= last; v++ )
sum += *v;
```

60

Code

```
#include <stdio.h>
#define MAX 12
int main( void ) {
int x[MAX]; /* declare 12-element array */
int i, sum;
for ( i=0; i<MAX; i++ ) { x[i] = i; }
/* here, what is value of i? of x[i]? */
sum = 0;
for ( i=0; i<MAX; i++ ) { sum += x[i]; }
printf( "sum = %d\n",sum );
} /* end of main() */
```

61

Code 2

```
#include <stdio.h>
#define MAX 10
int main( void ) {
int x[MAX]; /* declare 10-element array */
int i, sum, *p;
p = &x[0];
for ( i=0; i<MAX; i++ ) { *p = i + 1; p++; }
p = &x[0];
sum = 0;
for ( i=0; i<MAX; i++ ) { sum += *p; p++; }
printf( "sum = %d\n",sum );
} /* end of main() */
```

62

2 dimensional arrays

- 2-dimensional arrays
- int weekends[52][2];
- you can use indices or pointer math to locate elements in the array
  - weekends[0][1]
  - weekends+1
- `weekends[2][1]` is same as `*(weekends+2*2+1)`, but NOT the same as `*weekends+2*2+1` (which is an integer)!

63

swap

```
void swapNot( int a,int b ) {
  int tmp = a;
  a = b;
  b = tmp;
} // end swapNot()


void swap( int *a,int *b ) {
  int tmp = *a;
  *a = *b;
  *b = tmp;
} // end swap()
```

64

swap

```
int x, y;        // declare two ints
int *px, *py;    // declare two pointers to ints
x = 3;   // initialize x
y = 5;   // initialize y

printf( "before: x=%d y=%d\n",x,y );

swapNot( x,y );
printf( "after swapNot: x=%d y=%d\n",x,y );

px = &x; // set px to point to x (i.e., x's address)
py = &y; // set py to point to y (i.e., y's address)

printf( "the pointers: px=%p py=%p\n",px,py );

swap( px,py );
printf( "after swap with pointers: x=%d y=%d px=%p py=%p\n",x,y,px,py );

// you can also do this directly, without px and py:
swap( &x,&y );
printf( "after swap without pointers: x=%d y=%d\n",x,y );
```

65

Pointers

- Make sure you feel comfortable with the idea of what is happening inside pointer
- Will try to use more examples today to make specific points

66

```
int main(){
  int number = 10;
  foo(&number);
  return 0;
}

void foo(int *p){
  *p = 30;
  }
```

67

Question

- Whats the advantage of passing in by pointer reference ?
- What is the problem?
- How would we solve it?

68

const

- Allows the compiler to know which values shouldn’t be modified
- Added in to c later

- Example:

```
const int a = 5;

void foo(const int x) {   }
```

69

const

- Better than #define since error message will be easier to understand since preprocessor not involved
- Very useful in functions to either return const or make sure a pointer doesn’t alter the original object

70

Const pointer to non-const

- This is a pointer which always points to same location, but the value can be modified
- int * const ptr = &x;

```
*ptr = ??
can't say
ptr = & ??
```

- Example2: array name

71

Const pointer to const data

- Int x = 200;
- const int * const ptr = &x;

72

- Some confusion
  - int const * X
  - const int * X //variable pointer to const
  - int * const Y //const pointer to int
  - int const * const Z //const point to const

73

Pointers to functions

- C allows you to also pass around a pointer to a function
- `void foo (int , int (*) (int , int) );`
- `int example1(int x, int y) { return x+y; }`
- `foo(5, example1);`

74

- ```
  void foo(int a, int (*A)(int,int)){

  if((*A)(5,10) > 0){

  }
  else {

  }

  }
  ```

75

Creating your own types

- Equivalent to a class idea in other programming languages, you can define your own types in c

```
struct name {

  types
  }
```

76
```

example

```
struct point {
   int x;
   int y;
  }
```

- Usage:

```
struct point a;
a.x = 5;
a.y = 10;
```

77

Anonymous structs

- Can also create anonymous structs

```
struct {
   int x;
   int y;
  } a, b;
```

78

Nesting

```
struct rect {
    struct point pt1;
    struct point p2;
  }
```

- Use:
  struct rect largeScreen;

79

Making space

- Remember in the proceeding examples, simple types so memory is automatically allocated (in a sense).
- struct student {
      char * name;
      int age;
  }

  struct student a;
  a.name = (char*)malloc(sizeof(char)*25));
  …

80

Use in functions

```
struct point makePoint(int x, int y)
  {
      struct point temp;
      temp.x = x;
      temp.y = y;
      return temp;
  }
```

81

Operations

- Copy
- Assignments
- & (addressing)
- Accessing members

- How do we compare 2 structs

82

Structs and pointers

- ```
  struct point *example
  = (struct point *)malloc(sizeof(struct
  point));
  ```

- `(*example).x`

  what does
  `*example.x mean?`

  Shortcut:
  example->x

83

typedef

- defining your own types using typedef (for ease of use)

```
typedef short int smallNumber;
typedef unsigned char byte;
typedef char String[100];

smallNumber x;
byte b;
String name;
```

84
```

enum

- define new integer-like types as enumerated types:

```
enum weather { rain, snow=2, sun=4 };
typedef enum {
Red, Orange, Yellow, Green, Blue, Violet
} Color;
```

- look like C identifiers (names)
- are listed (enumerated) in definition
- treated like integers
  - start with 0 (unless you set value)
  - can add, subtract — e.g., color + weather
  - cannot print as symbol automatically (you have to write code to do the translation)

85

enum

- just fancy syntax for an ordered collection of integer constants:

```
typedef enum {
Red, Orange, Yellow
} Color;
```

- is like

```
#define Red 0
#define Orange 1
#define Yellow 2
```

- here's another way to define your own boolean:

```
typedef enum {False, True} boolean;
```

86

Usage

```
enum Boolean {False, True};

...
enum Boolean shouldWait = True;
...
if(shouldWait == False) { .. }
```

87

struct

```
int main() {
struct {
int x;
char y;
float z;
} rec;
rec.x = 3;
rec.y = 'a';
rec.z = 3.1415;
printf( "rec = %d %c %f\n",rec.x,rec.y,rec.z
  );
} // end of main()
```

88

struct

```
int main() {
struct record {
int x;
char y;
float z;
};
struct record rec;
rec.x = 3;
rec.y = ’a’;
rec.z = 3.1415;
printf( "rec = %d %c %f\n",rec.x,rec.y,rec.z );
} // end of main()
```

89

```
int main() {
typedef struct {
int x;
char y;
float z;
} RECORD;

RECORD rec;
rec.x = 3;
rec.y = ’a’;
rec.z = 3.1415;
printf( "rec = %d %c %f\n",rec.x,rec.y,rec.z );
} // end of main()
```

90

- note the use of malloc where “sizeof” takes the struct type as its argument (not the pointer!)

```
int main() {
typedef struct {
int x;
char y;
float z;
} RECORD;
RECORD *rec = (RECORD *)malloc( sizeof( RECORD ));
rec->x = 3;
rec->y = 'a';
rec->z = 3.1415;
printf( "rec = %d %c %f\n",rec->x,rec->y,rec->z );
} // end of main()
```

91

Important to understand

- overall size of struct is the sum of the elements, plus padding for alignment (i.e., how many bytes are allocated)
- given previous examples: sizeof(rec) -> 12
- but, it depends on the size and order of content (e.g., ints need to be aligned on word boundaries, since size of char is 1 and size of int is 4):

```
struct {
char x;
int y;
char z;
} s1;
/* x y z */
/* |----|----|----| */
/* sizeof s1 -> 12 */
```

```
struct {
char x, y;
int z;
} s2;
/* xy z */
/* |----|----| */
/* sizeof s2 -> 8 */
```

92

Reminder

- pointers to structs are common — especially useful with functions (as arguments to functions or as function type)
- two notations for accessing elements: (*sp).field or sp->field
- (note: *sp.field doesn't work)

```
struct xyz {
int x, y, z;
};
struct xyz s;
struct xyz *sp;
...
s.x = 1;
s.y = 2;
s.z = 3;
sp = &s;
(*sp).z = sp->x + sp->y;
```

93

Arrays of structs

- notations for accessing elements: arr[i].field

```
struct xyz {
int x, y, z;
};
struct xyz arr[2];
...
arr[0].x = 1;
arr[0].y = 2;
arr[0].z = 3;
arr[1].x = 4;
arr[1].y = 5;
arr[1].z = 6;
```

94

unions

- union
- like struct:

```
union u_tag {
int ival;
float fval;
char *sval;
} u;
```

- but only one of ival, fval and sval can be used in an instance of u (think container)
- overall size is largest of elements

95

Example

```
#define NAME_LEN 40

struct person {
  char name[NAME_LEN+1];
  float height;
};

int main( void ) {
  struct person p;
  strcpy( p.name,"suzanne" );
  p.height = 60;
  printf( "name   = [%s]\n",p.name );
  printf( "height = %5.2f inches\n",p.height );
} // end of main()
```

96

Files

- so perl makes working with files a 3 line process

```
open (FH,"a.txt");
while(<>){
chomp;
print splice (split / / ) 1 1;
}
```

97

File Handling

- `File *log_file;`

- any ideas what this look like ?

98

- use function fopen to open handle
- pass in arguments to fopen to set type
  - r read
  - w write
  - a append
- need to check if not null

99

```
if( (log_file = fopen(“some.txt”, “w”)) == NULL)
  fprint(stderr,”Cannot open %s\n”, “log_file”);


/*****
do your cool stuff here

*****/

fclose(log_file);
```

100

moving characters

- can move characters using putchar(c) and getchar()

- if no handle supplied
- putchar(c,stdout)
- getchar(stdin)

101

strings

- fgets
- fputs

102

Next lab

- work with pointers
- create a small puzzle
- Play games ☺

103