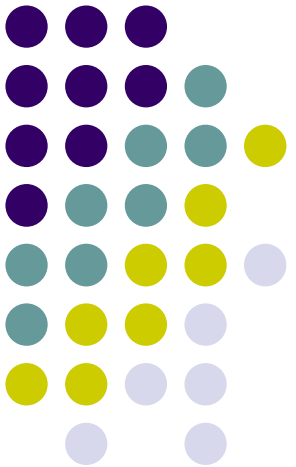
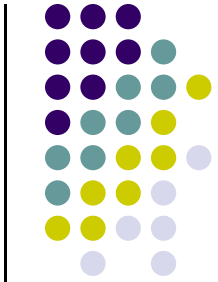
Day 01
Introduction to C

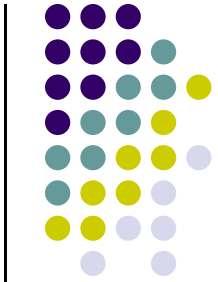


Saving Dyknow notes



- Save on the server.
  - Can access from anywhere.
    - Will have to install Dyknow on that machine.
    - Must connect to the network
- Bring a thumb drive and save it on the drive at the end of class.
  - Don’t have to connect to the network to access notes.
    - Will have to install Dyknow on that machine.
- Download from http://www.dyknow.com/download/
  - **DyKnow Client 5.0 (x86) , Version 5.0.70**
  - The dyknow server is **dyknow.cs.rose-hulman.edu**

Create a .cshrc file

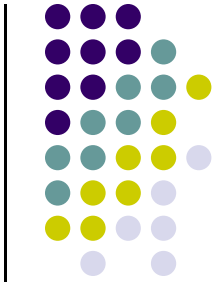


prompt> emacs .cshrc

- Create aliases for common commands
- After making changes, for the changes to take effect immediately:

prompt> source .cshrc

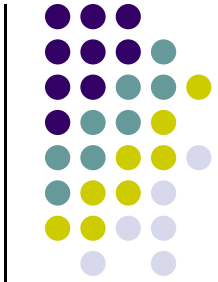
Why learn C (after Java)?



- Both high-level and low-level language
- Better control of low-level mechanisms
- Performance better than Java
- Java hides many details needed for writing OS code

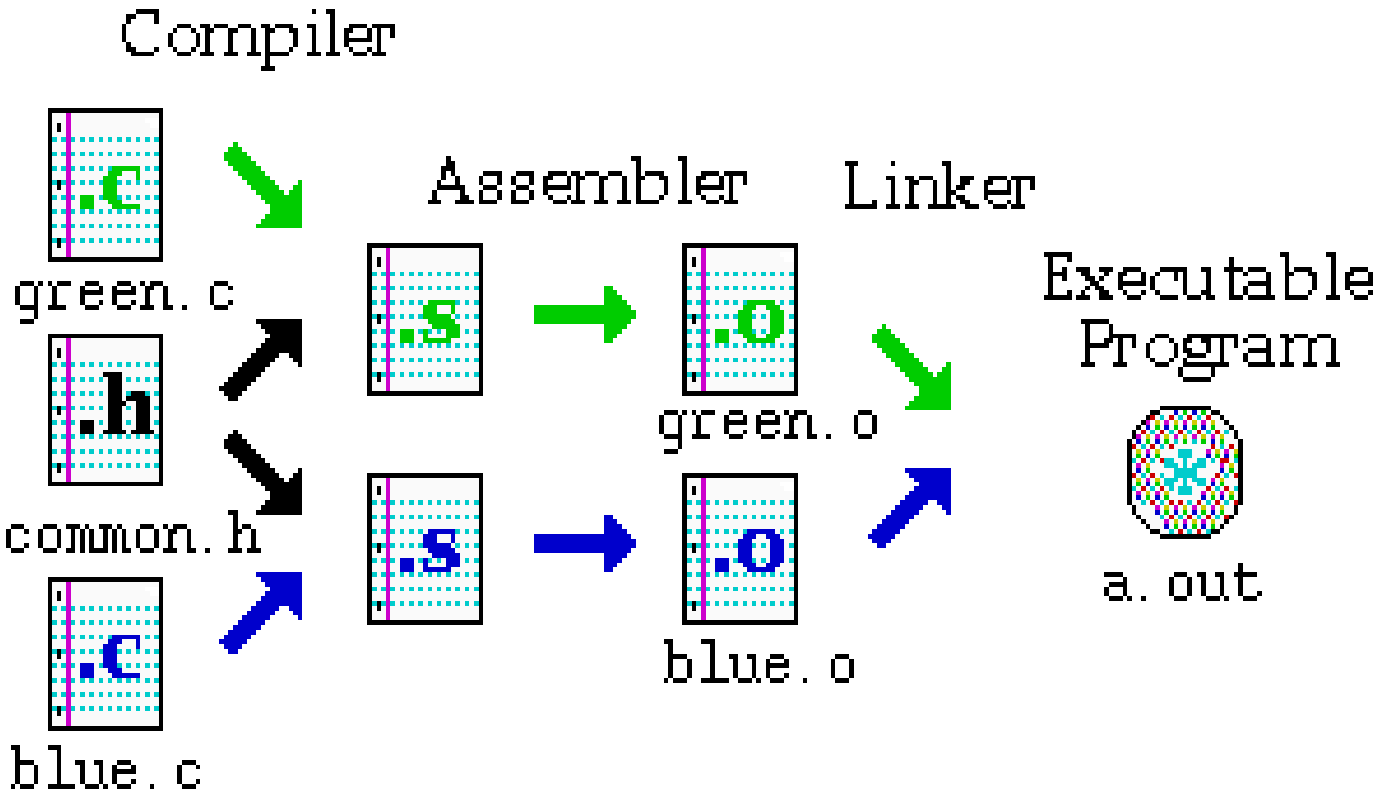
  But,….
- Memory management responsibility
- Explicit initialization and error detection
- More room for mistakes

Goals of this tutorial

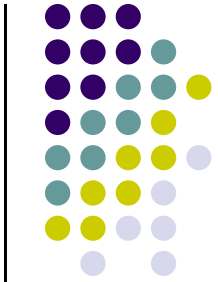


- To introduce some basic C concepts to you
  - so that you can read further details on your own
- To warn you about common mistakes made by beginners

Creating an executable

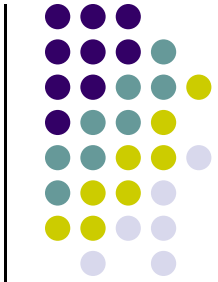


Types of files



- C source files (.c)
- C header files (.h)
- Object files (.o)
- Executable files (typically no extension – by default : a.out)
- Library files (.a or .so)

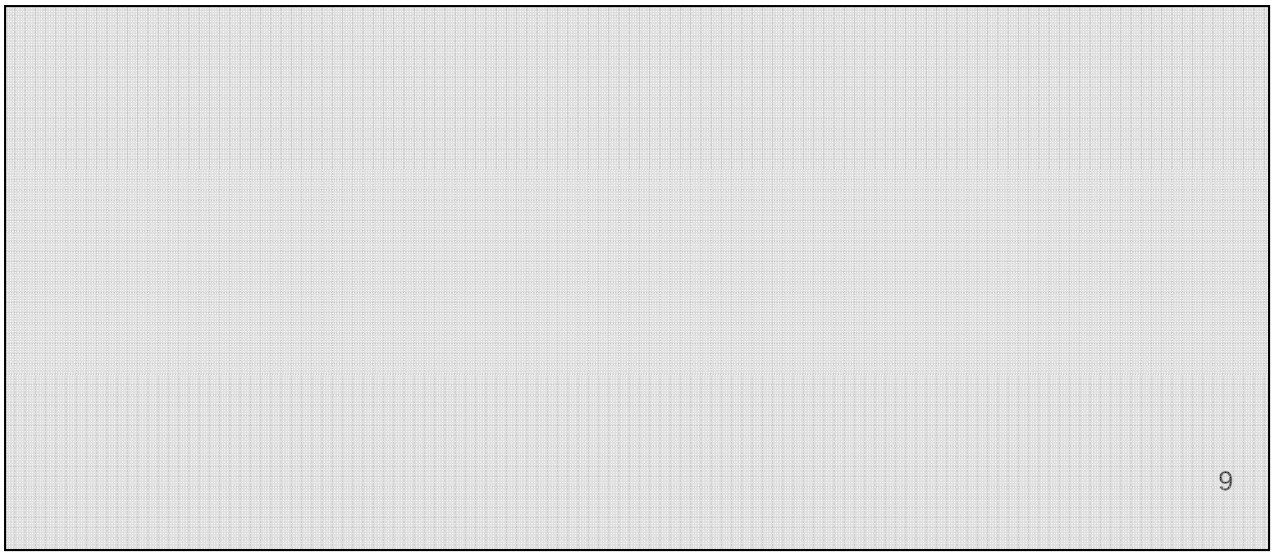
External library files lib*name*.a or lib*name*.so



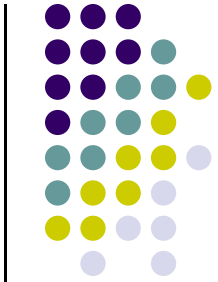
- Special functionality is provided in the form of *external libraries* of ready-made functions
- Ready-compiled code that the compiler merges, or *links*, with a C program during compilation
- For example, libraries of mathematical functions, string handling functions, and input/output functions
- Look for the library files under */usr/lib* and header files under */usr/include*

Example 1 – What is the output of this program?

```
#include <stdio.h>     //#include “myheader.h”

int
main()
{
   printf(“Hello World. \n \t and you ! \n ”);
     /* print out a message */
   return 0;
}
```

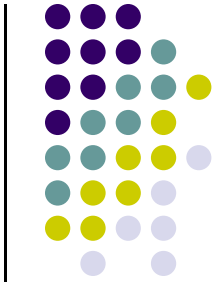


Summarizing the Example



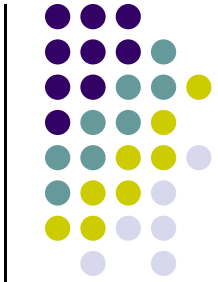
- `#include <stdio.h>` = include header file stdio.h
  - No semicolon at end
  - Small letters only – C is case-sensitive
- `int main(){ … }` is the only code executed
- `printf(" /* message you want printed */ ");`
- `\n` = newline `\t` = tab
- \ in front of other special characters within `printf` creates "escape sequences".
  - `printf("Have you heard of \"The Rock\" ? \n");`

Compiling and running



- >gcc eg1.c (Creates a.out)
- >./a.out (Runs the executable)

- >gcc eg1.c –o eg1 (Creates eg1 not a.out)
- >./eg1

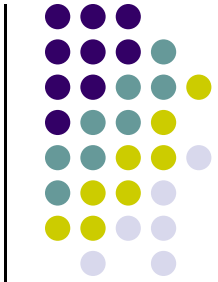


- To compile, use flag “l” and name i.e. –lname.

  Eg. gcc –o test test.c –lm

  where “m” in “lm” comes from libm.so i.e. the math library.

Using external library files



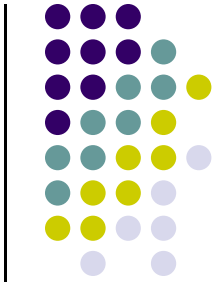
To use the library files, you must:

- include the library header files

You may also have to:

- link the library with a -l option to gcc

Pre-processor directives



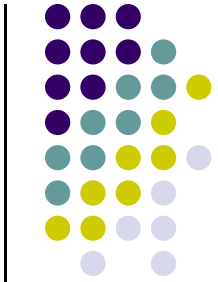
- A preprocessor is a program that examines C code before it is compiled and manipulates it in various ways.
- Two commonly used pre-processor directives
  - #include – to include library files.
  - #define - to define macros (names that are expanded by the preprocessor into pieces of text or C code)

Example of pre-processor directives

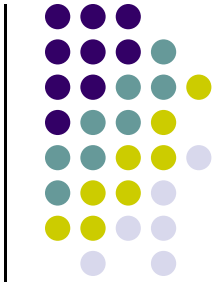
Example 2:

```
#include <stdio.h>
#define STRING1 "A macro definition\n"
#define STRING2 "must be all on one line!\n"
#define EXPRESSION1 1 + 2 + 3 + 4
#define EXPRESSION2 EXPRESSION1 + 10
#define ABS(x)    ((x) < 0) ? -(x) : (x)
#define MAX(a,b)   (a < b) ? (b) : (a)
#define BIGGEST(a,b,c)    (MAX(a,b) < c) ? (c) : (MAX(a,b))

int
main ()
{
  printf (STRING1);
  printf (STRING2);
  printf ("%d\n", EXPRESSION1);
  printf ("%d\n", EXPRESSION2);
  printf ("%d\n", ABS(-5));
  printf ("Biggest of 1, 2, and 3 is %d\n", BIGGEST(1,2,3));
  return 0;
 }
```

What is the output of the program?

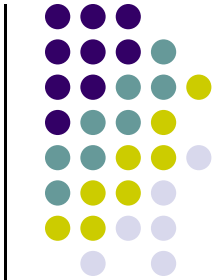


#define



- The expression is NOT evaluated when it replaces the macro in the pre-processing stage.
- Evaluation takes place only during the execution phase.

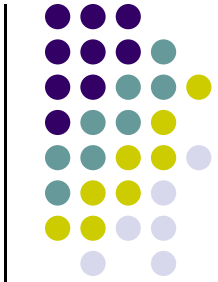
Simple Data Types



Data-type	# of bytes (typical)	Short-hand
int	4	%d %i
char	1	%c
float	4	%f
double	8	%lf
long	4	%l
short	2	%i

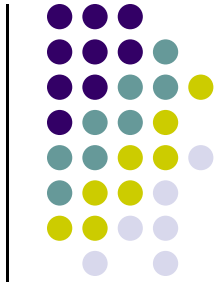
String - %s
address - %p(HEX) or %u (unsigned int)

Example 3



```
#include <stdio.h>

int
main()
{
int nstudents = 0; /* Initialization, required */
      float age = 21.527;

printf("How many students does RHIT have ?");
      scanf ("%d", &nstudents);  /* Read input */
printf("RHIT has %d students.\n", nstudents);
           printf("The average age of the students is %3.1f\n",age);
                 //3.1 => width.precision
return 0;
}
```

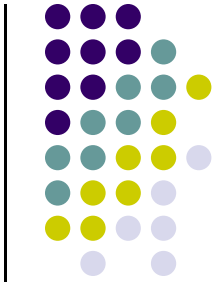
$How many students does RHIT have ?:2000 (enter)
RHIT has 2000 students.
The average age of the students is 21.5
$



Complete the program below:

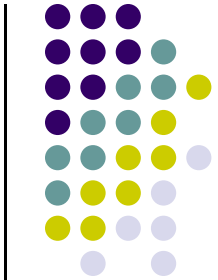
```
#include < stdio.h>

int main() {
    int nStudents, nFaculty;
    printf("How many students and faculty does RHIT have?\n");

     scanf(_____);
          /* You may use only one scanf statement to accept the two
             values. Assume that a space will be used by the user to
             delimit the two input values */

      printf(_____

                 _____);
           /* You may use only one printf statement to print the
              values. Use a meaningful statement to do so. */

       return 0;
}
```

Like Java

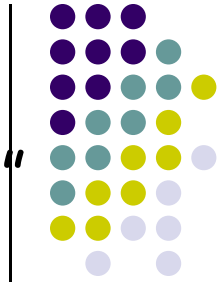


- Operators similar to those in Java:
  - Arithmetic
    - **int i = i+1; i++; i--; i *= 2;**
    - **+, -, *, /, %,**
  - Relational and Logical
    - **<, >, <=, >=, ==, !=**
    - **&&, ||, &, |, !**
- Syntax same as in Java:
  - **if ( ) { } else { }**
  - **while ( ) { }**
  - **do { } while ( );**
  - **for(i=1; i <= 100; i++) { }**
  - **switch ( ) {case 1: … }**
  - **continue; break;**

Example 4



```
#include <stdio.h>
#define DANGERLEVEL 5    /* C Preprocessor -
          - substitution on appearance */
               /* like Java 'final' */
int
main()
{
float level=1;
               /* if-then-else as in Java */
if (level <= DANGERLEVEL){ /*replaced by 5*/
          printf("Low on gas!\n");
     }
     else
          printf("On my way !\n");

return 0;
}
```

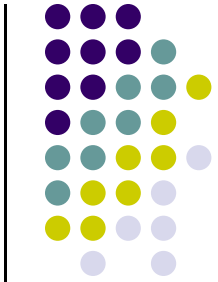


This problem is based on example 4. Change the "if" statement in the program to a "while", such that program will remain in the loop until "level" is greater than the DANGERLEVEL.

```
float level = 1;
```

```
{
     printf("Low on gas!\n");
    level = FillingGas();
     //Method or function that fills gas and
     //updates level.
}

printf(" I'm on my way!\n");
```

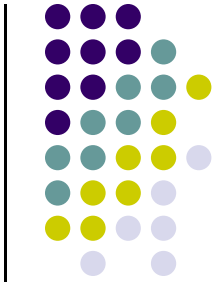
One-Dimensional Arrays



Example 5:

```
#include <stdio.h>

int
main()
{
   int number[12]; /* 12  numbers*/
   int index, sum = 0;
     /* Always initialize array before use */
   for (index = 0; index < 12; index++) {
number[index] = index;
   }
   /* now,number[index]=index; will cause error:why ?*/

   for (index = 0; index < 12; index = index + 1) {
sum += number[index];  /* sum array elements */
   }

   return 0;
}
```

String – an array of characters

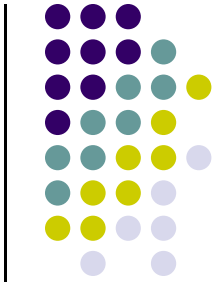


_____ **/* declare a string called name of 10 characters */**

/* Initialize "name" to "ALICE" */

/* Print "name" to the screen – Would you print all 10 characters or only the first 5 characters? *Hint: use %s* */

String – an array of characters

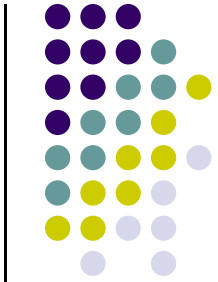


_char name[10];___ /*declare a string called name of 10 characters */

/* Initialize “name” to “ALICE” */
name[0] = ‘A’; name[1] = ‘l’; name[2] = ‘i’; name[3] = ‘c’;
name[4] = ‘e’;
name[5] = ‘\0’;

/* Print “name” to the screen – Would you print all 10 characters or only the first 5 characters? *Hint: use %s* */
printf(“%s”, name);

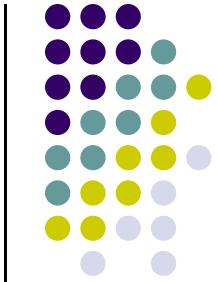
Strings – character array



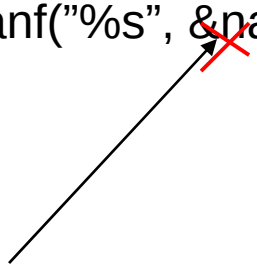
```
/* Declaring and initializing strings */
char name[ ] = {'A','l','i','c','e','\0'};
char name [ ] = "Alice";
char name [6] = {'A','l','i','c','e','\0'};
Char name[6] = "Alice";
```

```
/* Using scanf read an input string value and save it in "name"*/
scanf(_____, _____);
```

Strings – character array

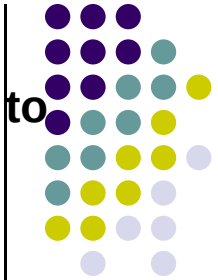


```
/* Declaring and Initializing strings */
char name[ ] = {'A','l','i','c','e','\0'};
char name [ ] = "Alice";
char name [6] = {'A','l','i','c','e','\0'};

/* Using scanf read an input string value and save it in "name"*/
scanf("%s", &name);
```



No ampersand

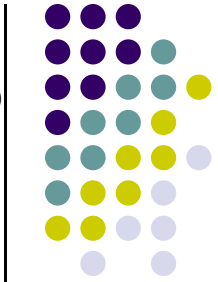
```
scanf("%s",name); //Initialization
                          // ERROR: scanf("%s",&name);
printf("%s", name); /* print until '\0' */
```



What is the output of the following code segment? Use the “man” pages to determine what “strlen” does?

```
char name[ ] = {‘A’,’l’,’e’,’x’,’\0’,’a’,’n’,’d’,’e’,’r’);
printf(“ The returned value is %d\n”,  strlen(name));
printf(“ And the string is %s\n”,name);
```

CAUTION: Strings lengths need not match the size of the array. So, always assign sufficient space to hold the longest string for that situation. No out-of-bounds exception!



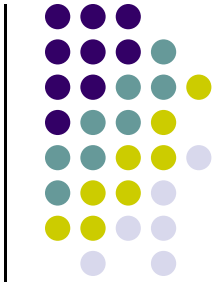
What is the output of the following code segment? Use the man pages to determine what “strlen” does?

```
char name[ ] = {‘A’,’l’,’e’,’x’,’\0’,’a’,’n’,’d’,’e’,’r’);
printf(“ The returned value is %d\n”,  strlen(name));
printf(“ And the string is %s\n”,name);
```

The returned value is 4.
And the string is Alex.

CAUTION: Strings lengths need not match the size of the array. So, always assign sufficient space to hold the longest string for that situation. No out-of-bounds exception!

Other string functions



Functions to operate on strings

_____ compare two strings

_____ break a string into tokens

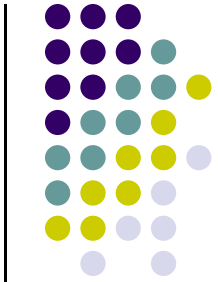
_____ finds the first occurrence of a character in a string

_____ make a copy of a string

What are the libraries that handle strings and characters?

Do you need a special flag when compiling a program that uses a function from either string library?

Other string functions



Functions to operate on strings

_strcmp, strncmp_____ compare two strings

_strtok_____ break a string into tokens

_strchr _____ finds the first occurrence of a character in a string

_strcpy_____ make a copy of a string

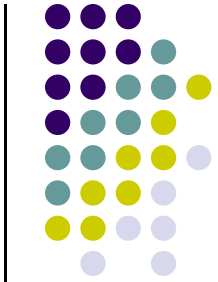
What are the two string libraries?

string.h strings.h stdlib.h ctype.h

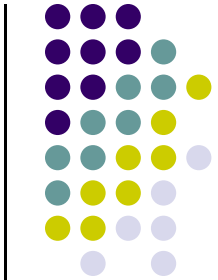
Do you need a special flag when compiling a program that uses a function from either string library?

We can get this information from the “man” pages. And, the answer is that a special flag is not needed.

Strings contd.



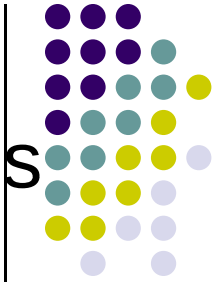
- Functions to operate on strings
  - strcpy, strncpy, strcmp, strncmp, strcat, strncat, substr, strlen,strtok
  - #include <strings.h> or <string.h> at program start
- CAUTION: C allows strings of any length to be stored. Characters beyond the end of the array will overwrite data in memory following the array.



- Multi-dimensional arrays

```
    int points[3][4];
    points [1][3] = 12;
/* NOT points[3,4] */
    printf("%d", points[1][3]);
```

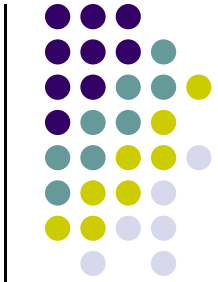
Structures



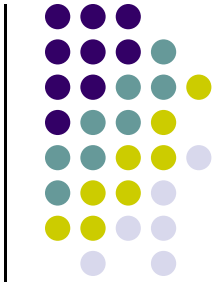
- Similar to Java's classes without methods

```
Example 6:
#include <stdio.h>

struct birthday{      /*"struct birthday" is the type*/
    int month;
    int day;
    int year;
  };                          //Note the semi-colon

int
main() {
  struct birthday mybday;  /* - no 'new' needed ! */
            /* then, it's just like Java ! */
  mybday.day=1; mybday.month=1; mybday.year=1977;
  printf("I was born on %d/%d/%d", mybday.day,
                              mybday.month, mybday.year);
}
```

More on Structures



```
struct person{
    char name[41];
    int age;
    float height;
    struct {          /* embedded structure */
      int month;
      int day;
      int year;
    } birth;
 };

struct person me;
me.birth.year=1977;……

struct person class[60];
/* array of info about everyone in class */
class[0].name="Gun"; class[0].birth.year=1990;……
```

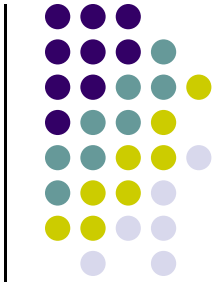


Define a structure called Inventory. It has the following attributes:

- item number
- quantity at hand
- price
- expiration date (month/year)

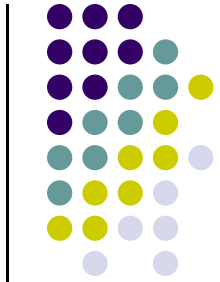
The expiration date must be defined as a struct within the Inventory struct.

Declare an array of 10 such Inventory items.



Initialize the expiration date for the 5th inventory item to January 2005.

Initialize the 5th item’s name to “papaya”.



Define a structure called Inventory. It has the following attributes:

- item number
- quantity at hand
- price
- expiration date (month/year)

The expiration date must be defined as a struct within the Inventory struct.

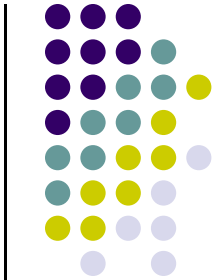
```
struct Inventory {
        int number;
        char name[20];
        int quantity;
        float price;
        struct {
              int year;
               int month;
        } expDate;
};
```

Declare an array of 10 such Inventory items.

```
struct Inventory myInv[10];
```

Initialize the expiration date for the 5th inventory item to March 2006.

```
myInv[5].expDate.month  = 1;
myInv[5].expDate.year = 2005;
```



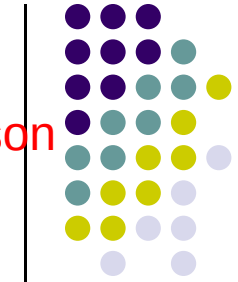
Initialize the 5th item's name to "papaya".

```
strcpy( myInv[4].name, "papaya");
```

typedef struct person myPerson

- Defines a new type myPerson as a synonym for the type struct person

```
int main(){
   myPerson me; //instead of struct person me;
   me.age = 6;
   …
}
```



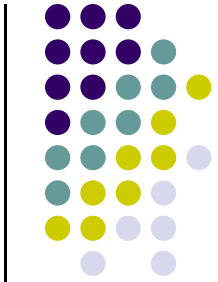
Use typedef to create a type called “Inv” which is a synonym for struct Inventory.

Declare an array of 5 such items.

typedef struct person myPerson

- **Defines a new type myPerson as a synonym for the type struct person**

```
int main(){
  myPerson me; //instead of struct person me;
  me.age = 6;
  …
}
```



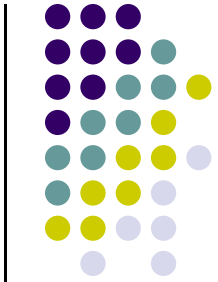
Use typedef to create a type called “Inv” which is a synonym for struct Inventory.

typedef struct Inventory Inv;

Declare an array of 5 such items.

Inv myInv[5];

User-defined header files

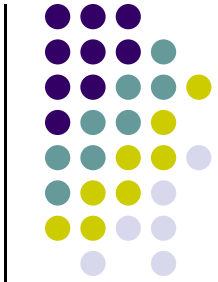


- Structures and other data structures may be defined in a header file, for better organization of the code.
- These are user-defined header files e.g. inventory.h
- To include it:

#include ``inventory.h''

at the start of the program file.

Command line arguments

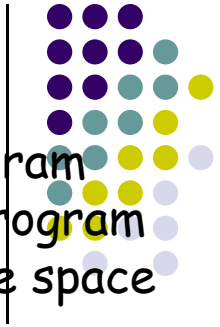


- Accept inputs through the command line.
- main(int argc, char* argv[])
  - argc – argument count
  - argv[] – value of each argument

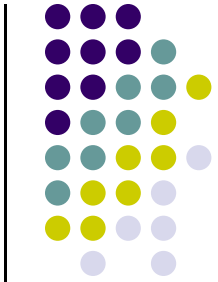
Example 7

```
#include <stdio.h>

int
main(int argc, char *argv[])
{
    int count = 0;
    if(argc < 2){
            printf("Must enter at least one argument\n");
            printf("Example: ./a.out this is program 7\n");
            exit(1);
    }
    printf(" The number of arguments is %d\n", argc);
    printf("And they are :\n");
    while(count < argc){
            printf("argv[%d]: %s\n",count,argv[count] );
            count++;
   }
   printf("\n");
   return 0;
}
```

Download the program example7.c from the class website. (Look under General Resources-> C Programs.) Do not make any changes to the program (unless the program does not compile). Compile the program. Run the program correctly at least once and then write the output of the program in the space provided below.



File handling



- Open a file using “fopen”
  - -Returns a file pointer which is used to access the file.

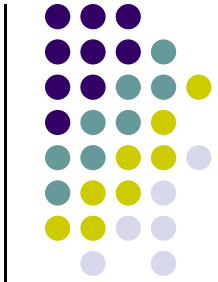
- Use the man pages, to answer the following:

What is the type of the value returned by the fopen function?

- Match the operation with the file open mode argument value?

w	error if file does not already exist. Else, file pointer at the beginning of file.
a	delete contents of existing file or create a new file. File pointer at the beginning of file.
r	create a new file if file does not exist. Preserve the contents if file does exist and place file pointer at the end of the file.

File handling

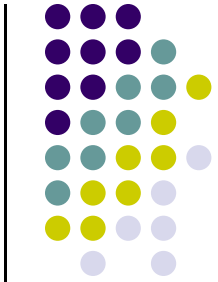


- A few more file handling functions:
- fprintf

- fscanf

- fclose

- Do they need any special library files?



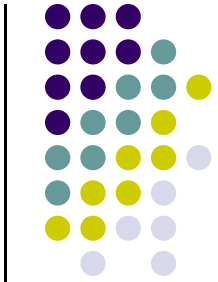
- **Open a file using “fopen”**
  - Returns a file pointer which is used to access the file.

Use the man pages, to answer the following:
What is the type of the value returned by the fopen function?
FILE *

What are the modes in which a file can be opened?
__r___ – error if file does not already exist. Else, file pointer at the beginning of file.
__w___ – delete contents of existing file or create a new file. File pointer at the beginning of file.
__a___ – create a new file if file does not exist. Preserve the contents if file does exist and file pointer at the end of the file.

A few more file handling functions:
fprintf - write to the stream in the format specified
fscanf - read from the stream
fclose - close the file and return the file pointer.

Do they need any special library files?
Yes, stdio.h



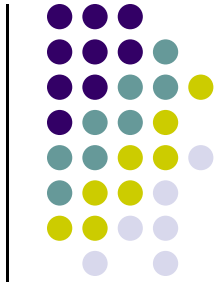
```
/* Declare a file pointer called in_file */
```

```
/* open a file called “test.txt” for writing */
```

```
 /* No exception handling – so check if file opened successfully */
   if(in_file == NULL){
        exit(1);  /* exit program – don’t return to calling function */
   }

/* Write a string “Hello there” to the file */
 /* Function prototype:
  int fprintf(FILE *stream,  const  char  *format,  /*  args*/  ...);
*/
```

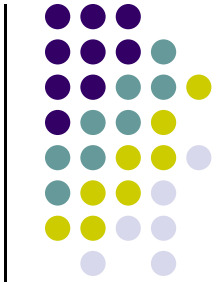
```
/* Write the value of the int variable “count” followed by the value of
the float variable “price” to the file. Separate the two values with
space */
```

```
/* Don’t forget to release file pointer */
```



```
/* Declare a file pointer called in_file */
   FILE *in_file;

/* open a file called “test.txt” for writing */
  in_file = fopen(“test.txt”, “w”);

 /* No exception handling – so check if file opened successfully */
   if(in_file == NULL){
        exit(1);  /* exit program – don’t return to calling function */
   }

/* Write a string “Hello there” to the file */
 /* Function prototype:
  int fprintf(FILE *stream,  const  char  *format,  /*  args*/  ...);
*/
  fprintf(in_file, “Hello there”);

/* Write the value of the int variable “count” followed by the value of
the float variable “price” to the file. Separate the two values with
space */
    fprintf(in_file, “%i %f”, count, price);

/* Don’t forget to close the file and release the file pointer */
    fclose(in_file);
```

Reading till end of file

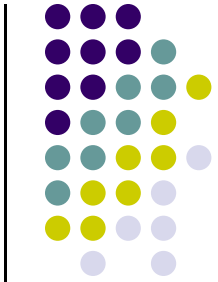


int feof(FILE *) – The function is defined in stdio.h

- **Returns a non-zero value if end of file has been reached, and zero otherwise.**

/* Using a while loop, read data from a file that has only integers until there is no more data to read */

Reading till end of file



int feof(FILE *) – The function is defined in stdio.h

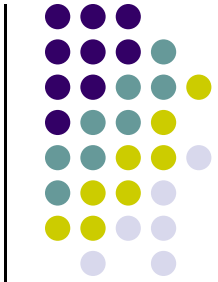
- Returns a non-zero value if end of file has been reached, and zero otherwise.

/* Using a while loop, read data from a file that has only integers until there is no more data to read */

```
int count = 0;
 fscanf(in_file, "%d", &int1);            // Try to read first item
 while (feof(in_file) == 0 ){              // If there is data, enter loop
     count++;
     printf("%d \n", int1);               // Do something with the data
     fscanf(in_file, "%d", &int1);        // Try reading again for next item
 }                                            // Go back to while to test if more
                                        // data was read
```

Functions – C methods

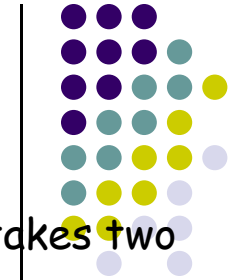
Why do we use functions in programs?



- **Passing arguments to functions**
  - **pass the value of the variable**
  - **pass the address of the variable (use of the phrase “by reference” is incorrect in C.)**

- **Returning values from functions**
  - **return the value of the variable**
  - **use the address passed to the function to modify the value**

Functions



```
#include <stdio.h>
/* Write the function prototype for function "sum" that returns an int value and takes two
ints as parameters */
```

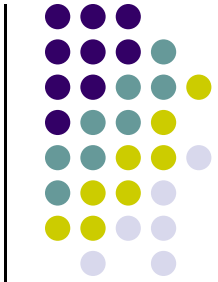
```
int
main(void){
   int total, a = 5, b = 3;
/* Call the function "sum". The parameters are a and b and the returned value must be saved
in total */
```

```
    return 0;
}

/* The function implementation */
int sum(int a, int b){    /* arguments passed by value */
       return (a+b);      /* return by value */
}
```

Functions

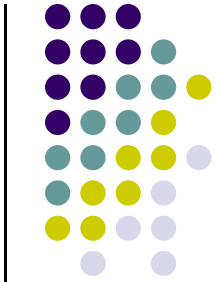
```
#include <stdio.h>
/* Write the function prototype for function "sum" that returns an int value and takes two
ints as parameters */
   int sum (int x, int y);



int main(void){
   int total, a = 5, b = 3;
/* Call the function "sum". The parameters are a and b and the returned value must be saved
in total */
   total = sum(a,b);
   return 0;
}



/* The function implementation *.
int sum(int c, int d){       /* arguments passed by value */
       return (c + d);       /* return by value */
}
```

Complete the program



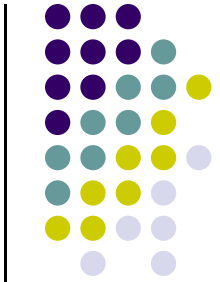
```
/* Function: Product -   The function returns the product of the two input parameters. */
    int Product (int input1, int input2);

/* Function: WriteToFile -      The function writes to the file pointed to by filePtr, the  values of input1,
      input2, and output.   */
 void WriteToFile(FILE *file_ptr, int input1, int input2, int output1);

 int
   main( ){
       FILE *f_ptr;
       int inp1=3;
       int inp2 = 4;
       int out_value;

       /* Call the Product function with inp1 and inp2 as arguments and save the return value
       in out_value. */

    ____________________________________________________________

    //Call the function WriteToFile with f_ptr, inp1, inp2 and out_value as arguments.

    ____________________________________________________________

       return 0;
}
```

Complete the program

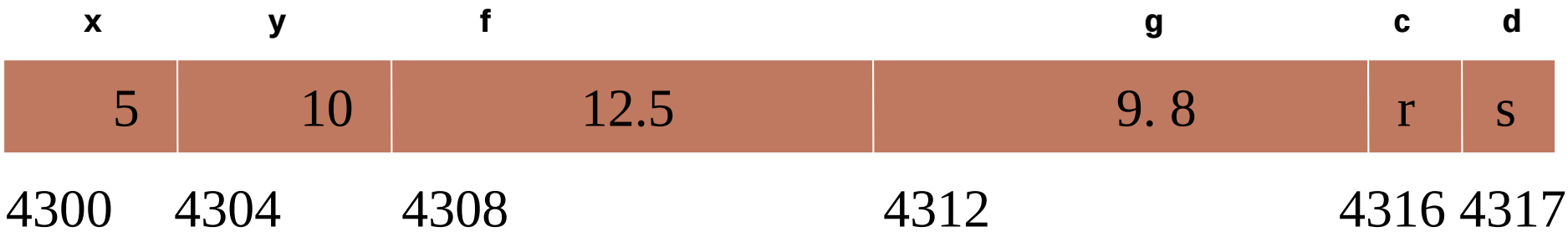


```
/* Function: Product -   The function returns the product of the two input parameters. */
    int Product (int input1, int input2);

/* Function: WriteToFile -      The function writes to the file pointed to by filePtr, the  values of input1,
      input2, and output.   */
 void WriteToFile(FILE *filePtr, int input1, int input2, int output1);

 int main( ){
       FILE *fPtr;
       int inp1=3;
       int inp2 = 4;
       int outValue;

       /* Call the Product function with inp1 and inp2 as arguments and save the return value
        in outValue. */
        outValue = Product(inp1,inp2);

      //Call the function WriteToFile with fPtr, inp1, inp2 and outValue as arguments.
       WriteToFile(fPtr, inp1, inp2, outValue);

      return 0;
}
```

Memory layout and addresses



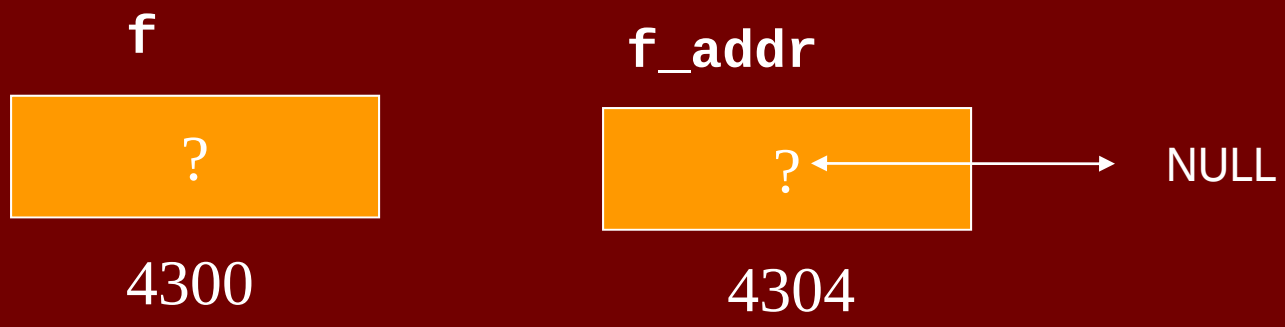
```
int x = 5, y = 10;
float f = 12.5, g = 9.8;
char c = ‘r’, d = ‘s’;
```



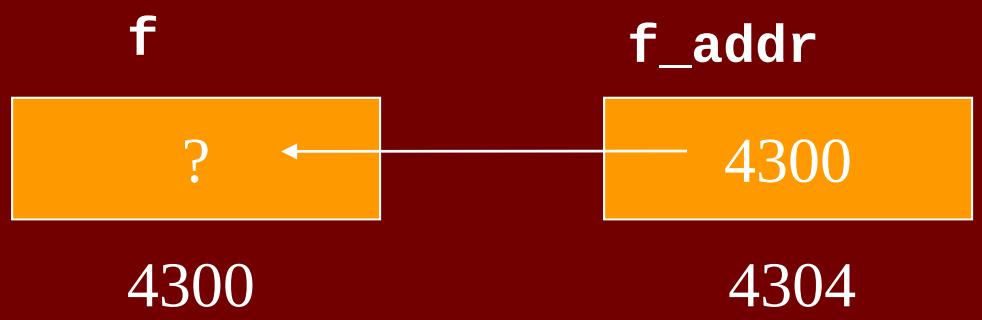
Pointers made easy - 1

float f; // data variable - holds a float

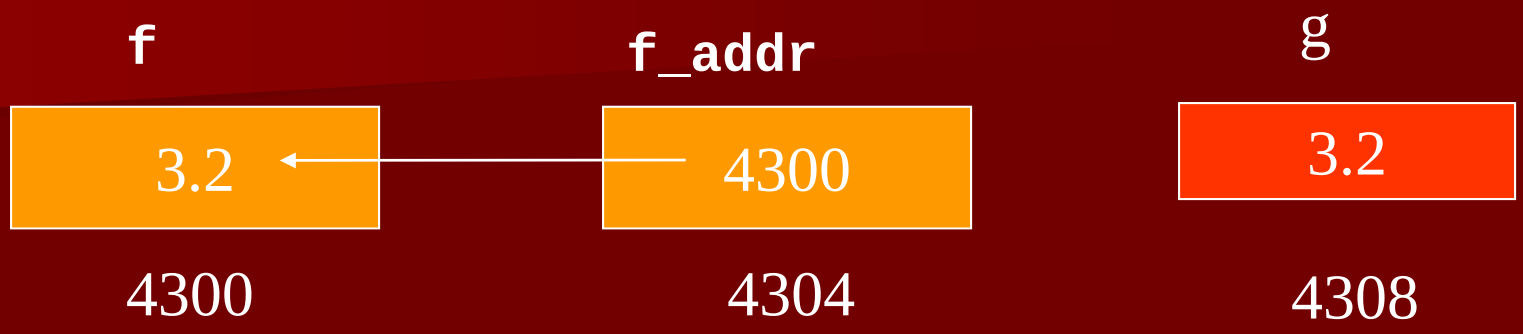
float *f_addr; // pointer variable – holds an address to a
// float



f_addr = &f; // & = address operator

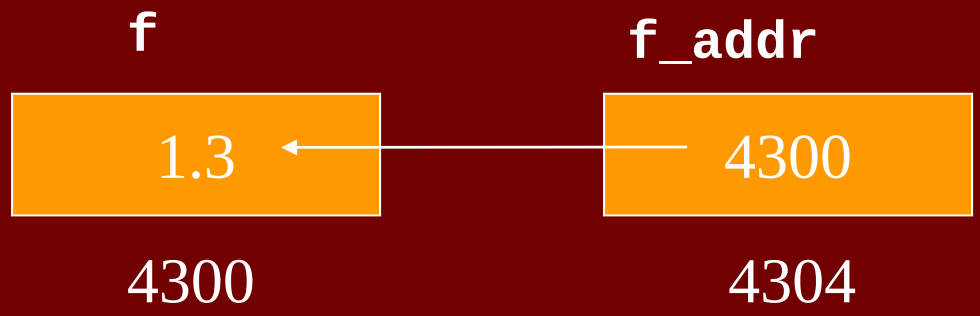


***f_addr = 3.2; // indirection operator or dereferencing**

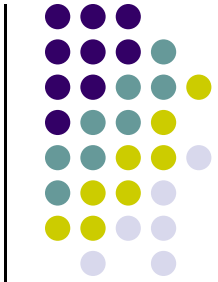


float g=*f_addr; // indirection: g is now 3.2

f = 1.3;



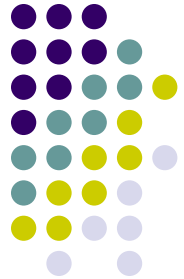
Pointer operations



- Creation
  - int *ptr;
  - Pointer assignment/initialization
  - ptr = &i; (where i is an int and &i is the address of i)
  - ptr = iPtr; (where iPtr is a pointer to an int)
- Pointer indirection or dereferencing
  - i = *ptr; (i is an int and *ptr is the int value pointed to by ptr)

Example 10

Use blocks as shown earlier, to explain pointer creation, pointer assignment and so on. The statements that you must graphically represent have been numbered.



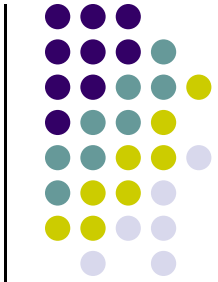
```
#include <stdio.h>

int main(void) {
    int *ptr, j; //j is not a pointer.

    ptr=&j;   /* initialize ptr before using it */  ------------ Line 1
          /* *ptr=4 does NOT initialize ptr */
       *ptr=4;                                        -------------- Line 2
       j=*ptr+1;                                      -------------- Line 3
       return 0;
}
```

Line 1

Line 2

Line 3

Pointers and arrays



```
int p[10], *ptr; // Both p and ptr are pointers
                 // i.e.  hold addresses.
                  // p is already pointing to a fixed location and
                 // cannot be changed.
                  // ptr is still to be initialized.
```

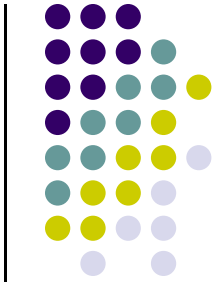
p[i] is an int value.

p, &p[i] and (p+i) are addresses or pointers.

*p = 5; ⇔ p[0] = 5;

*(p+i) = 5; ⇔ p[i] = 5;

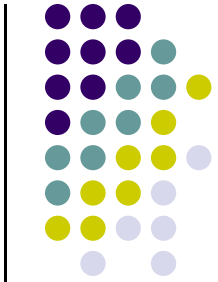
Pointer arithmetic



```
int *ptr;
int p[10];
ptr = p;          // or ptr = &p[0]
ptr +=2;          //Assume ptr = 3000 before this
                   //statement
```

What is the value of ptr?

Pointer arithmetic



```
int *ptr;
int p[10];
ptr = p;          // or ptr = &p[0]
ptr +=2;          //Assume ptr = 3000
```

What is the value of ptr?

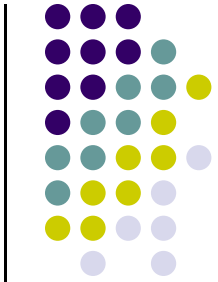
ptr = ptr + 2 * sizeof(int) = ptr+8 bytes

ptr = 3000 + 8 = 3008

=> ptr = &(p[2]);

ERROR: p = ptr; because “p” is a constant address, points to the beginning of a static array.

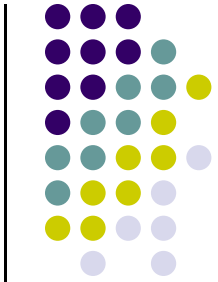
Dynamic memory allocation



- **Explicit allocation and de-allocation by user using malloc() and free().**
- **(void *)malloc(size_t size);**
- **void free(void *ptr);**
- **bytes sizeof(type)**

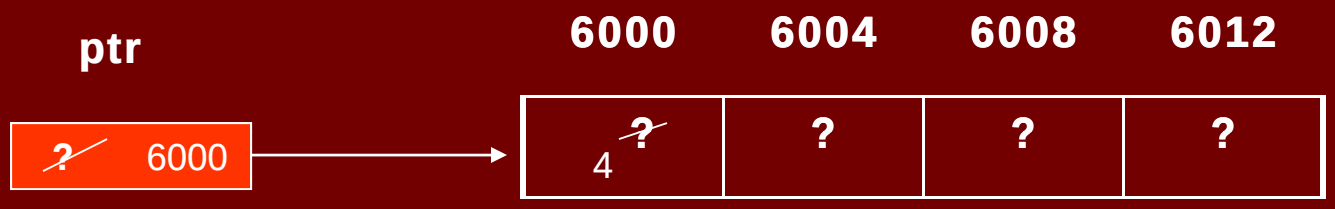
```
#include <stdio.h>
int main() {
    int *ptr;
      /* allocate space to hold 4 ints */


    /* do stuff with the data */
    *ptr=4; //ptr[0] = 4;

    /* free up the allocated space */


      return 0;
}
```

Dynamic memory allocation



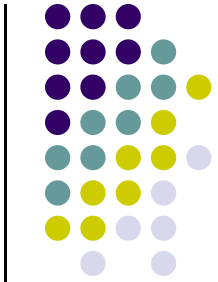
```
#include <stdio.h>

int main() {
   int *ptr;
       /* allocate space to hold 4 ints */
   ptr = (int*)malloc(4 * sizeof(int));

      /* do stuff with the space */
   *ptr=4; //ptr[0] = 4;

   free(ptr);
      /* free up the allocated space */
      return 0;
}
```

```
int *ptr;
ptr = (int*)malloc(4 * sizeof(int));  //Address 6000 on the heap is allocated
*ptr=4;
```




```
free(ptr);
```

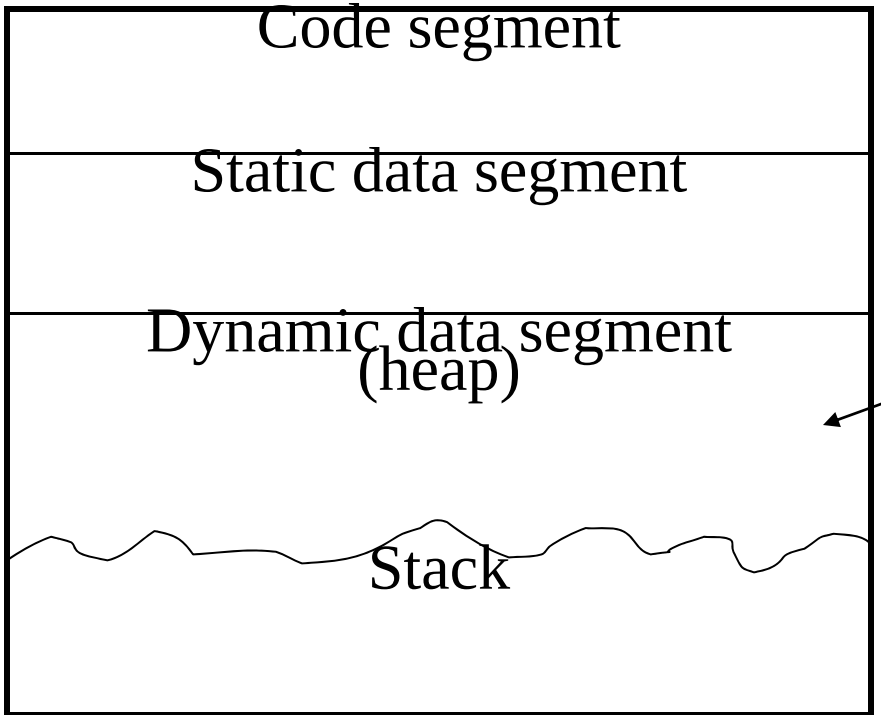
Man pages – Section 3



>>man –s 3 free

>>man –s 3 malloc

Memory allocation for a process.

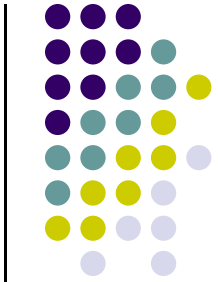


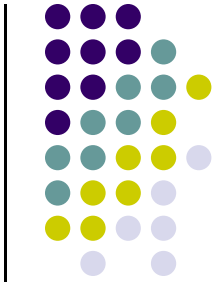
int *p; //p is created in the static data segment

p = (int *)malloc(4 * sizeof(int)); //Space for 4 ints i.e. contiguous 16

//bytes is allocated on the heap

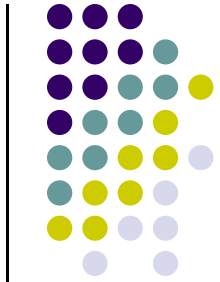
See video





- How you would write the value “25” to address 6008
  - Using pointer arithmetic

  - Using “ptr” as the address of an array.



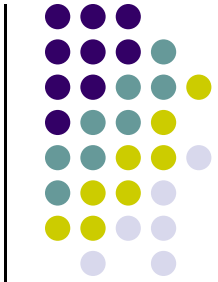
- How you would write the value “25” to address 6008
  - Using pointer arithmetic

    *(ptr + 2)= 25;

  - Using “ptr” as the address of an array.

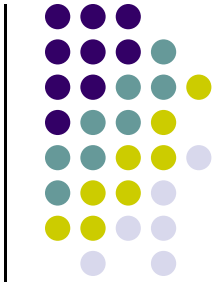
    ptr[2] = 25;

Dynamic array



```
int *ptr, i,size;
printf("Enter the size of the array");
scanf("%d",&size)


//Create a dynamic array of "size" ints.
ptr = (int*)malloc(_____________________________);

for(i=0; i<size; i++){
     ptr[i] = i;
}
```

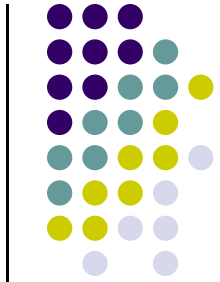
Dynamic array



```
int *ptr, i,size;
printf("Enter the size of the array");
scanf("%d",&size)


//Create a dynamic array of "size" ints.
ptr = (int*)malloc( size * sizeof(int) );

for(i=0; i<size; i++){
    ptr[i] = i;
}
```

What is the output of the following program?

```
 #define SIZE 4


int
 main(){
    int i = 0;
    int c[SIZE]  = {0,1,2,3};     //c is assigned the address 5000.

    int *c_ptr = c;

   printf(" The value of c is %u\n",c);
   printf(" The value of c_ptr is %u\n",c_ptr);

  while(i < SIZE){
               printf("%i\t",*c_ptr);
              c_ptr++;
              i++;
  }
     printf("\n c_ptr is %u\n",c_ptr);
}
```



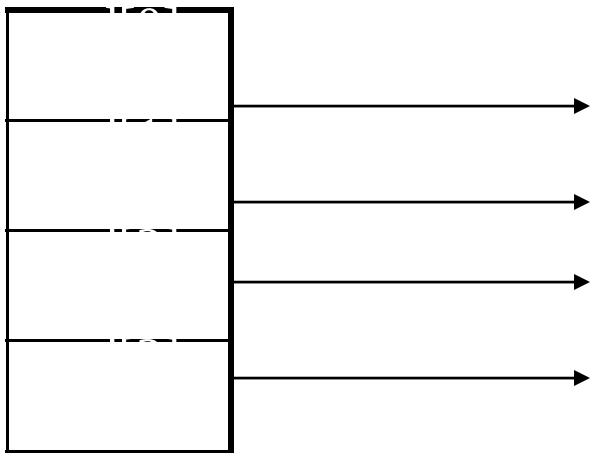
What is the output of the following code segment?

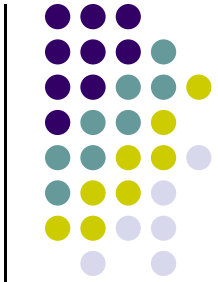
```
int i = 0;
char c[5]  = "game";      //c is assigned the address 5000.
char *cPtr = c;

while(i < strlen(c)){
   printf("%c",*cPtr);
   cPtr++;
   i++;
}

printf("\n cPtr is %u\n",cPtr);
```

Array of Pointers

- Variable length strings

```
char *card[4];        // card[4] => array of 4 elements
                        //char* => element is a pointer to a character.
                        // card[4] => array of 4 char pointers
```





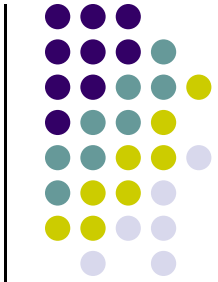
card[0] = (char*)malloc(6*sizeof(char));

card[1] = (char*)malloc(3*sizeof(char)); and so on.

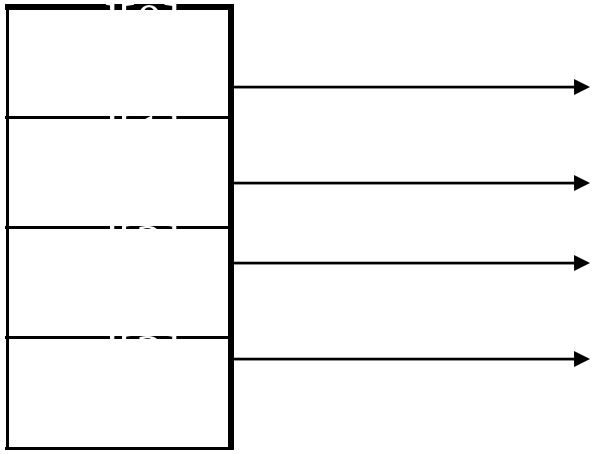
- Static allocation of a 2D array:

  char card[4][10]; //waste of space

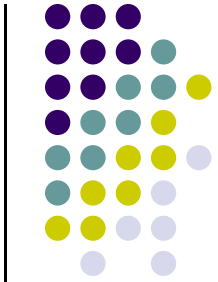
For the following code segment, complete the diagrams provided to graphically represent the pointer creations and assignments. Again, line numbers are provided. You may assume that the malloc on line 2 allocates space starting at address 5000 and the malloc on line 3 allocates space starting at address 6000.



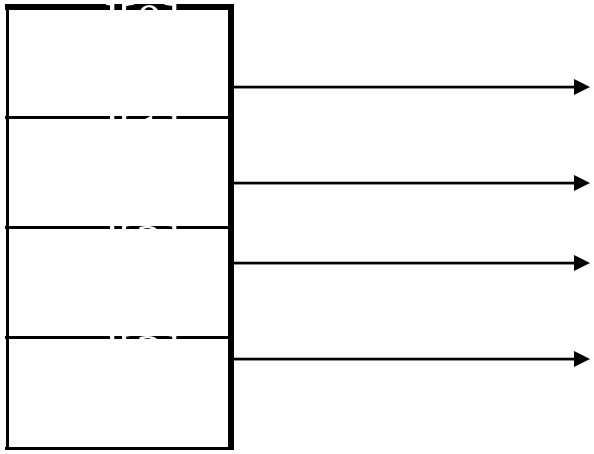
```
char *card[4];
card[0] = (char*)malloc(8*sizeof(char));        ----- line 2
card[3] = (char*)malloc(9*sizeof(char));        ------- line 3
strcpy(card[0],"hearts");                       ------- line 4
strcpy(card[3],"diamonds");                      ------ line 5
```



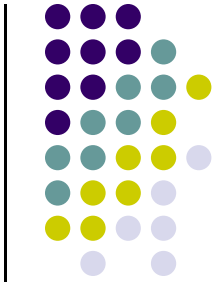
For the following code segment, complete the diagrams provided to graphically represent the pointer creations and assignments. Again, line numbers are provided. You may assume that the malloc on line 2 allocates space starting at address 5000 and the malloc on line 3 allocates space starting at address 6000.



```
char *card[4];
card[0] = (char*)malloc(8*sizeof(char));        ----- line 2
card[3] = (char*)malloc(9*sizeof(char));        ------- line 3
strcpy(card[0],"hearts");                       ------- line 4
strcpy(card[3],"diamonds");                      ------ line 5
```

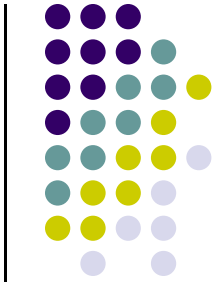


Common errors - Memory leak



```
int *ptr, x;
ptr = (int*)malloc(10*sizeof(int)); //ptr gets space
                                    //starting at address 3000
ptr = &x;
```

Common errors - Memory leak



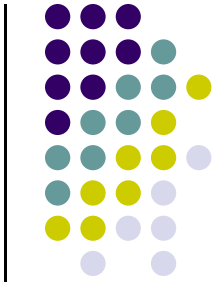
```
int *ptr, x;
ptr = (int*)malloc(10*sizeof(int)); //ptr gets space
                                    //starting at address 3000
ptr = &x;
```

The space allocated through malloc is no longer available for use by the program.

Released only when program quits.

Becomes a problem in large programs where a large number of variables are created and destroyed during the execution of the program.

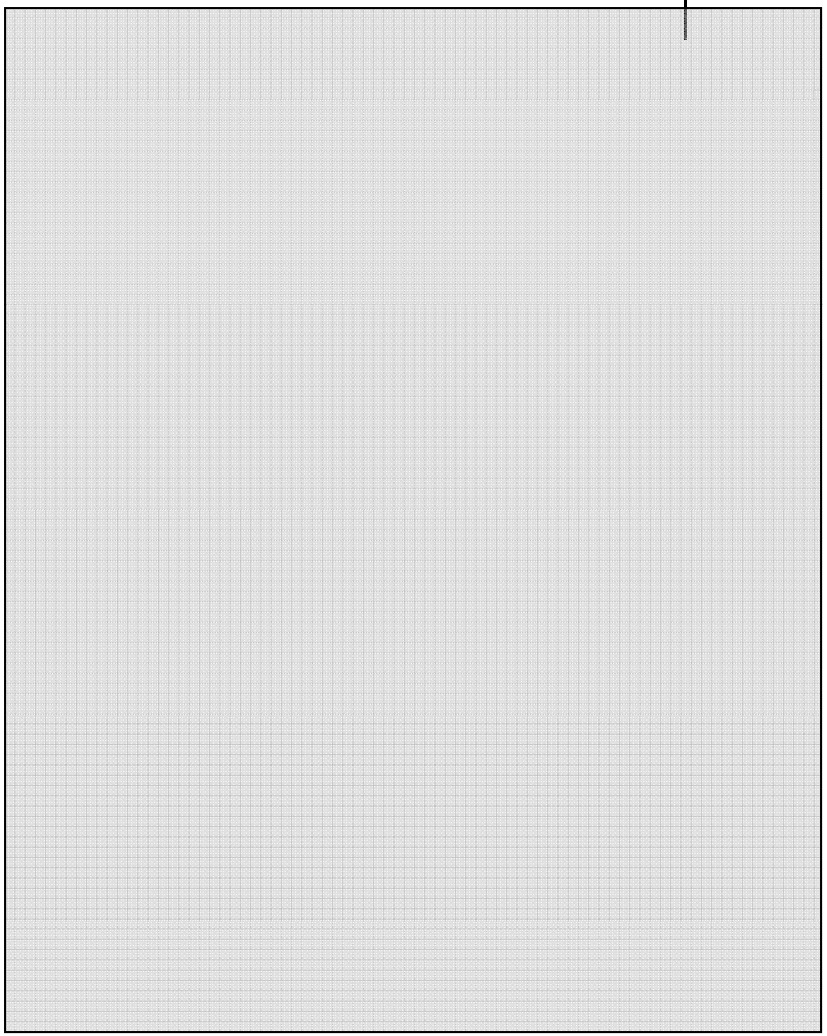
Common erros - Dangling pointers



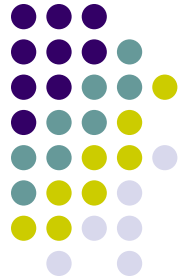
```
int *i, *x;
i = (int*)malloc( 5 x sizeof(int));
x = i;       / * both point to the same address. */
free(x);    /* both i and x are dangling pointers and  trying to
                    access either of them can cause logical
   errors
                    */
x = NULL;    /* One way to prevent incorrect access */
i = NULL;

void free_ptr(void *ptr){
      free(*ptr);
      ptr = NULL;
}
```

Identify the error in the following code segments and correct them?

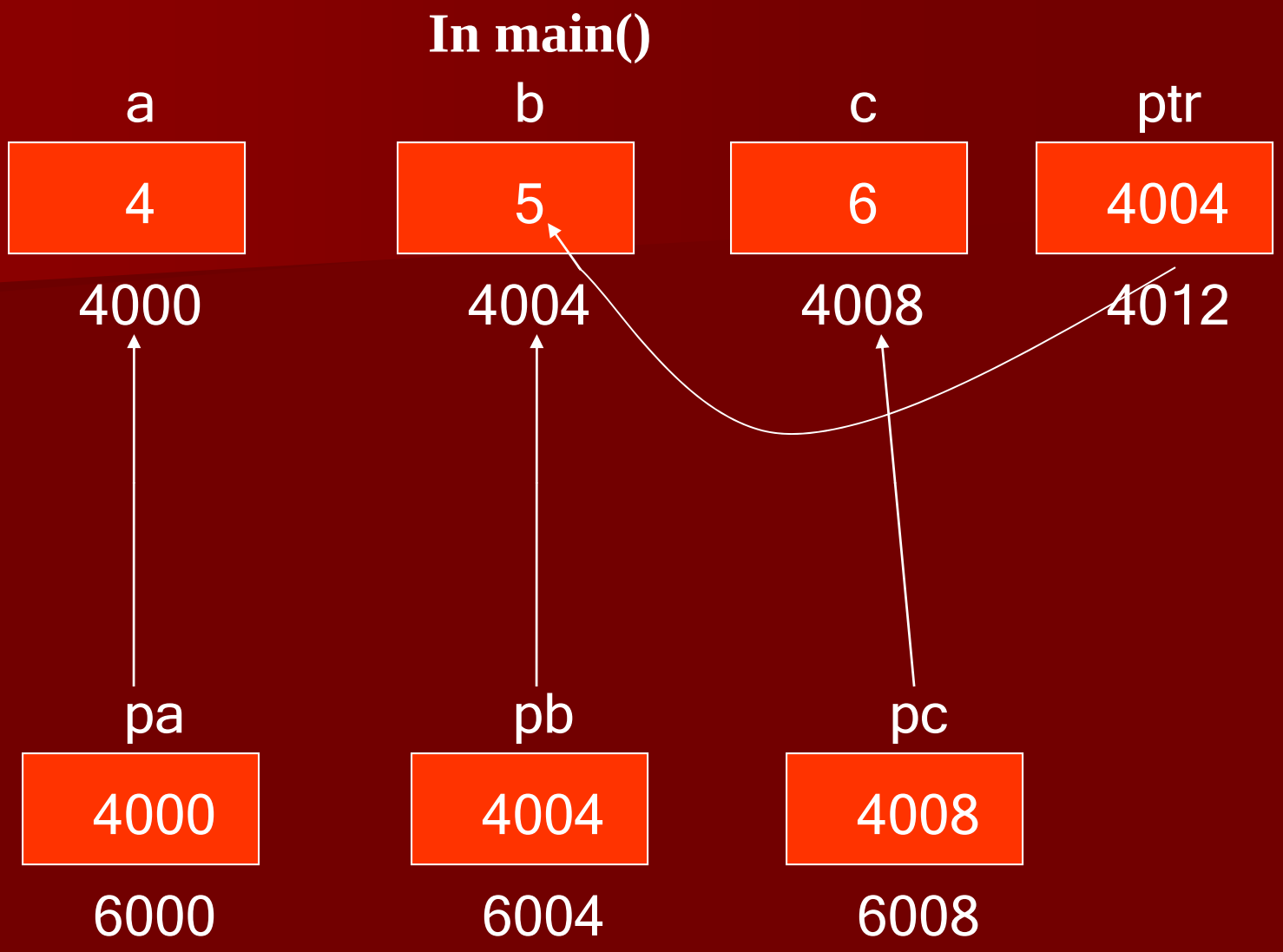
```
char *name;
scanf("%s",name);

char *city;
strncpy(city,"Bombay",3);

int *a;
a[0] = 5;
```

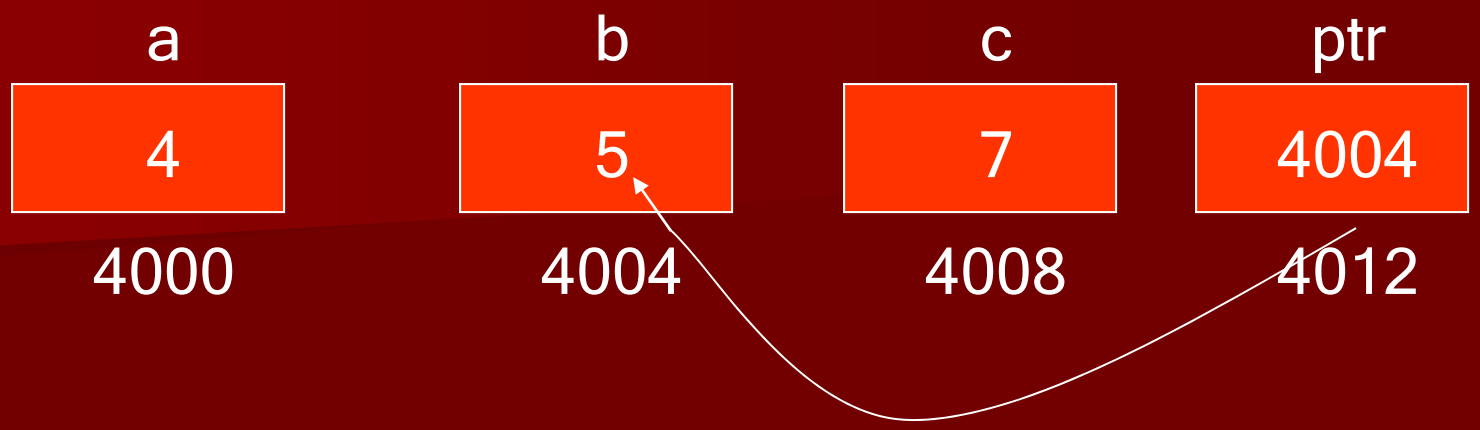


Functions – pointers as arguments



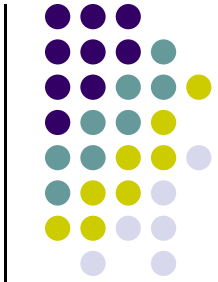
```
#include <stdio.h
int SumAndInc(int *pa, int *pb,int* pc);

int
main(int argc, char *argv[])
{
   int a=4, b=5, c=6;
   int *ptr = &b;
   int total = SumAndInc(&a,ptr,&c);
       /* call to the function */
   printf("The sum of 4 and 5 is %d and c is %p\n",total,c);
}
int
SumAndInc(int *pa, int *pb,int *pc ){/* pointers as arguments */
      *pc = *pc+1;      /* return a pointee value */
       /*NOT *(pc+1)*/
return (*pa+*pb);       /* return by value */
}
```


In main()
a
b
c
ptr
4
5
6
4004
4000
4004
4008
4012
pa
pb
pc
4000
4004
4008
6000
6004
6008
In function



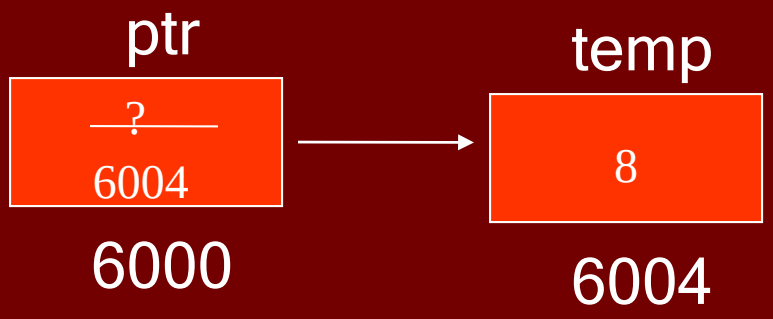
In main() after the function call

What's wrong with this ?

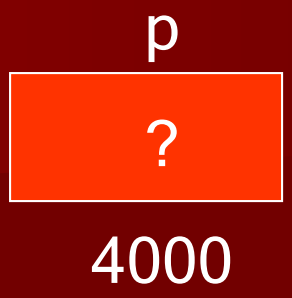


```
#include <stdio.h>

void DoSomething(int *ptr);

int
main(int argc, char *argv[]) {
int *p;
DoSomething(p);
printf("%d", *p);     /* will this work ? */
return 0;
}
void
DoSomething(int *ptr){ /* passed and returned by
                  reference */
  int temp= 5+3;
  ptr = &(temp);
}

/* compiles correctly, but gives incorrect output */
```

In main()

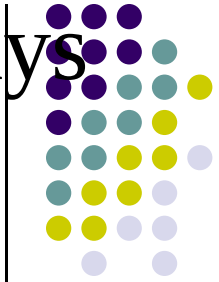


In the function



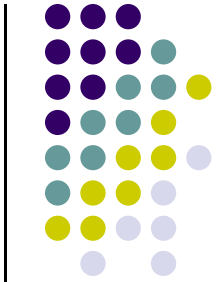
In main() after the function call

Functions - Passing and returning arrays



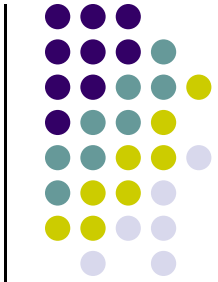
```
#include <stdio.h>

void init_array( int array[], int size ) ;

int
main(int argc, char *argv[] )
{
      int list[5];

      init_array( list, 5);
      for (i = 0; i < 5; i++)
            printf("next:%d", list[i]);
}

void init_array(int array[], int size) { /* why size ? */
       /* arrays ALWAYS passed by reference */
      int i;
      for (i = 0; i < size; i++)
            array[i] = 0;
}
```

Passing/Returning a structure



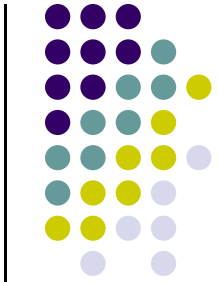
```
void DisplayYear_1( struct birthday mybday ) {
  printf("I was born in %d\n", mybday.year);
}                  /* - inefficient: why ? */

      /* pass pointer to struct */
void DisplayYear_2( struct birthday *pmybday ) {
  printf("I was born in %d\n", pmybday->year);
/* Note: '->', not '.', after a struct pointer*/
}

      /* return struct by value */
struct birthday GetBday(void){
  struct birthday newbday;
  newbday.year=1971;  /* '.' after a struct */
  return newbday;
}                  /* - also inefficient: why ? */
```

Input/Output statements



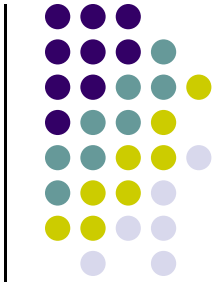
- fprintf(stdout,"….",…); - buffered output
  - Equivalent to printf("….",…)
- fscanf(stdin,…);
  - Equivalent to scanf(…)
- fprintf(stderr,"…",…); - un-buffered output
  - Use for error messages.
- perror(…);
  - Use to print messages when system calls fail.

gdb - debugger



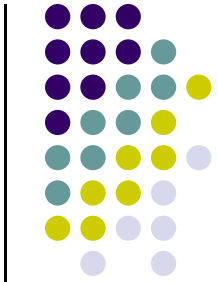
- Tutorial on class website.
- Reference sheet on class website.

Storage classes



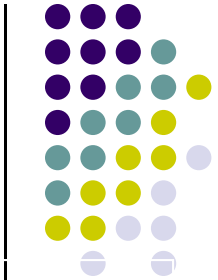
- Automatic (default for local variables)
  - Allocate memory only when function is executed
  - e.g. auto int i;
- Register
  - Direct compiler to place variable in a register
  - e.g. register counter = 1;
- Static
  - Allocate memory as soon as program execution begins
  - Scope is local to the function that declares the variable.
  - Value is retained and space is de-allocated only when program (not function) quits.
  - e.g. static int i;

Storage classes - contd



- Extern
  - Default for function names.
  - For a variable shared by two or more files:
    - int i; //global variable in file 1
    - extern int i; //global in files 2,3 … x
  - For a function shared by 2 or more files, place a function prototype at the beginning of the files.

Program with multiple files



```
#include <stdio.h>
#include “mypgm.h”

int g_data=5;
int
main()
{
  Myproc();
  return 0;
}
```

- Library headers
  - Standard
  - User-defined

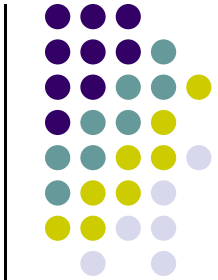
```
#include <stdio.h>
#include “mypgm.h”

void
Myproc()
{
int mydata=g_data * 2;
 . . . /* some code */
}
```

mypgm.c

```
void Myproc();
extern int g_data;
```

mypgm.h

To compile



- gcc main.c mypgm.c –o run

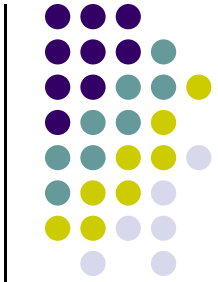
  or

  gcc main.c –c -o main.o

  gcc mypgm.c –c -o mypgm.o

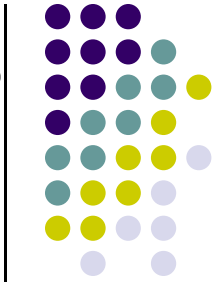
  gcc mypgm.o main.o –o run
- Can also use a makefile.

Externs



```
#include <stdio.h>

extern char user2line [20];   /* global variable defined
            in another file */
char user1line[30];           /* global variable defined
             in this file */


int main() {
 char user1line[20];       /* local scope - different from
            earlier user1line[30] */
 . . .              /* restricted to this func */
}
```

enum - enumerated data types



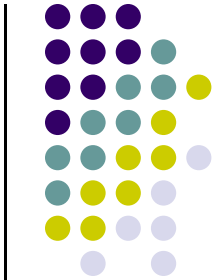
```
#include <stdio.h>
enum month{
JANUARY,        /* like #define JANUARY 0 */
FEBRUARY,       /* like #define FEBRUARY 1 */
MARCH           /* … */
};
```

In main:
enum month birthMonth;
If(birthMonth = = JANUARY){…}

```
/* alternatively, …. */
enum month{
JANUARY=1,     /* like #define JANUARY 1 */
MARCH=3,      /* like #define MARCH 3 */
FEBRUARY=2,           /* … */
};
```

Note: if you use the #define, the value of JANUARY will not be visible in the debugger. An enumerated data type's value will be.

Before you go….



- Always initialize anything before using it (especially pointers)
- Don’t use pointers after freeing them
- Don’t return a function’s local variables by reference
- No exceptions – so check for errors everywhere
- An array is also a pointer, but its value is immutable.