Pointers and Compound Data Types in C Language CIS*2520, Lab 02

Mohammad Naeem
mnaeem@uoguelph.ca
Shool of Computer Science, University of Guelph

Topics

Pointers and Data Structures in C

- sizeof()
- structures
- unions
- typedef
- malloc(), calloc(), realloc(), free
- pointers
- pointer arithmetic
- pointers to pointers
- pointers to functions

Struct

- describes a group of related data items treated as a single unit
- member data items have different data types
- Examples: Name (First Name, Middle Name, Last Name)

```
struct personal_data          ⟶  Name of the new data type
{

  char  name[100];
  char  address[200];
  int   year_of_birth;
  int   month_of_birth;       ⟶  One member of the new data type
  int   day_of_birth;

};
```


declaring a structure data type

- declare structure a structure data type

```
struct personal_data
{

  char  name[100];
  char  address[200];
  int   year_of_birth;
  int   month_of_birth;
  int   day_of_birth;

};
```

- declare a variable of the structure data type

```
struct personal_data person0001;
```

- alternatively

```
struct personal_data
{

  char  name[100];
  char  address[200];
  int   year_of_birth;
  int   month_of_birth;
  int   day_of_birth;

} person0001;
```

using structures

```
struct personal_data
{

  char  name[100];
  char  address[200];
  int   year_of_birth;
  int   month_of_birth;
  int   day_of_birth;

};

struct personal_data person0001;

person0001.year_of_birth = 1993;
person0001.month_of_birth = 09;
person0001.day_of_birth = 25;
…
…
```

structure_var_name.member_name

-> operator

```
struct personal_data
{

  char  name[100];
  char  address[200];
  int   year_of_birth;
  int   month_of_birth;
  int   day_of_birth;

};


struct personal_data person0001, *pdPtr;


pdPtr = &person0001
```

```
strcpy (person_ptr1->name, "Alice, Jordon");
strcpy (person_ptr1->address, “Aberdeen");

pdPtr->year_of_birth = 1990;
pdPtr->month_of_birth = 07;
pdPtr->day_of_birth = 13;
```

using struct

```
#include <stdio.h>
int main()
{
 struct personal_data
 {
        char   name[100];
        char   address[200];
        int    year_of_birth;
        int    month_of_birth;
        int    day_of_birth;
 };
```

```
struct   personal_data   person, * person_ptr1;

person_ptr1 =   &person;

strcpy (person_ptr1->name, "Adams, Douglas");
strcpy (person_ptr1->address, "The Galaxy");

 person_ptr1->year_of_birth  = 1990;
 person_ptr1->month_of_birth  = 3;
 person_ptr1->day_of_birth  = 25;


  return 0;
}
```


union

- declared the same way as structure data type
- unlike structure data type, may assume one of the type of the members

-decalre union

```
union int_or_float
{
  int     int_member;
  float   float_member;
};
```

-declare a variable of union

```
union   int_or_float  my_union;
```

```
my_union.int_member = 7;
```

(my_union has integer type at this point)

```
my_union.float_member = 7.5;
```

(now, my_union has float type)

using -> with unions

```
union int_or_float
{
  int    int_member;
  float  float_member;
};

union   int_or_float    my_union,   *my_union_ptr;

my_union_ptr = &my_union;


my_union_ptr->int_member = 7;        /* my_union has integer type at this point */

my_union_ptr->float_member = 7.5;/* now, my_union has  float type */
```


typedef

- assigns alternative names to existing types
- example

```
typedef   int   km_per_hour ;      /* assigns km_per_hour as name to the int type */
typedef   int   points ;           /* assigns points as name to the int type */

km_per_hour   current_speed ;      /* current_speed is of integer type* /
points    high_score ;             /* high_score is of integer type */
...
```

typedef

- Can be used to simplify naming for complex dara structure

```
struct var {
    int data1;
    char data2;
};

struct var a;
```

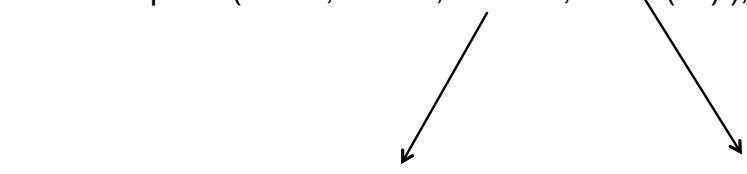
```
typedef struct var {
    int data1;
    char data2;
} newtype;

newtype a;
```

```
typedef struct {
    int data1;
    char data2;
} newtype;

newtype a;
```


sizeof() operator

- used to find the size of a data type or item, primitive or compound
- example

```
char c;

printf ( "%zu,%zu\n", sizeof c, sizeof(int) );
```



no parenthesis needed with variable

parenthesis needed with data type

sizeof() operator

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

int main(int argc, char **argv)
{
  char buffer[10];  /* Array of 10 chars */

  /* Only copy 9 characters from argv[1] into buffer.
   *  sizeof(char) is defined to be 1, so the number of
   *  elements in buffer is equal to its size in bytes.
   */

  strncpy(buffer, argv[1], sizeof(buffer) - sizeof(char));

  /* Set the last element of the buffer equal to null */
  buffer[sizeof(buffer) - 1] = '\0';

  return 0;
}
```


sizeof() operator

```
#include <stdio.h>

struct flexarray
{
    char val;
    char array[];  /* Flexible array member; must be last element of struct */
};

int main(int argc, char **argv)
{
    printf("sizeof(struct flexarray) = %zu\n", sizeof(struct flexarray) );

    return 0;
}
```

why sizeof() operator

- Useful to allocate memory of appropriate size to storing data items when dynamically created

```
int * pointer = malloc(sizeof(int) * 10 );
```


malloc()

- a library function
- allocates a block of memory on the heap
- returns a pointer if successful, NULL pointer otherwise

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

- size is the size of the element to which to allocate memory
- pointer returned is void, so should be casted to appropriate type

using malloc ()

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

int main ()
{

      int i,n;
      char * buffer;

        printf ("How long do you want the string? ");
        scanf ("%d", &i);

        buffer = (char*) malloc (i+1);
        if (buffer==NULL) exit (1);
```


using malloc ()

```
        for (n=0; n<i; n++)
          buffer[n]=rand()%26+'a';
        buffer[i]='\0';

        printf  ("Random string: %s\n",buffer);
        free (buffer);

        return 0;

 }
```

using malloc ()

```
typedef struct {
  int age;
  char name[20];

} data;
data *bob;
bob = (data*) malloc( sizeof(data) );
if( bob != NULL ) {
        bob->age = 22;
        strcpy( bob->name, "Robert" );
        printf ( "%s is %d years old\n", bob->name, bob->age );

}
free( bob );
```


calloc()

- allocates a block of memory for an array of *specified* elements
- each of them *size* bytes long
- then initializes each element to 0

```
void * calloc   ( size_t num,   size_t size );
```

num: number of elements to be allocated.
size : size of elements.

- returns a pointer to the memory block allocated
- pointer is always void*, which can be casted
- In case of failure, a NULL pointer is returned

using calloc()

```
/* calloc example */

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

int main ()
{
        int i,n;
        int * pData;

        printf ("Amount of numbers to be entered: ");
        scanf ("%d",&i);

        pData = (int*) calloc ( i, sizeof(int));
        if (pData==NULL) exit (1);
```


using calloc()

```
    for ( n = 0; n < i; n++ )  {

            printf ("Enter number #%d: ",n);
            scanf ("%d", &pData[n] );
        }

        printf ("You have entered: ");

        for ( n=0;n < i; n++ )    printf ("%d ", pData[n]);

        free (pData);

        return 0;
}
```

free()

- de-allocates space allocated through malloc(), calloc(), or realloc() and makes it available for use…

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

- **ptr** is a pointer
- no action happens if **ptr** is null

using free()

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

int main ()
{
        int * buffer1, * buffer2, * buffer3;

        buffer1 = (int*) malloc (100*sizeof(int));
        buffer2 = (int*) calloc (100,sizeof(int));
        buffer3 = (int*) realloc (buffer2,500*sizeof(int));

        free (buffer1);
        free (buffer3);

return 0;
}
```

using free()

```
#include <stdlib.h>
….
typedef struct data_type {

  int age;
  char name[20];

} data;

data *willy;

willy = (data*) malloc( sizeof(data) );
...
free( willy );
```


pointer arithmetic

- memory locations accessible via symbolic names
- in C, they are also accessible via other locations containing their addresses (called pointers)

- Pointer arithmetic
  - Using operators like -, +, ++, etc. to access data elements of a compound data type e.g. array, string, etc.
  - provides faster access to data

pointer arithmetic

❖ How do + and – work with pointers

int x = 50;

int *ptr = &x;

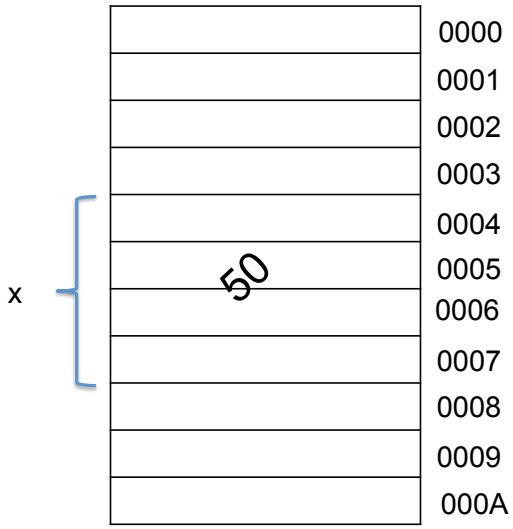
What is the value of ptr at this point?

ptr + 1;

What is the value of ptr at this point?

ptr - 1;

What is the value of ptr at this point?



pointer arithmetic

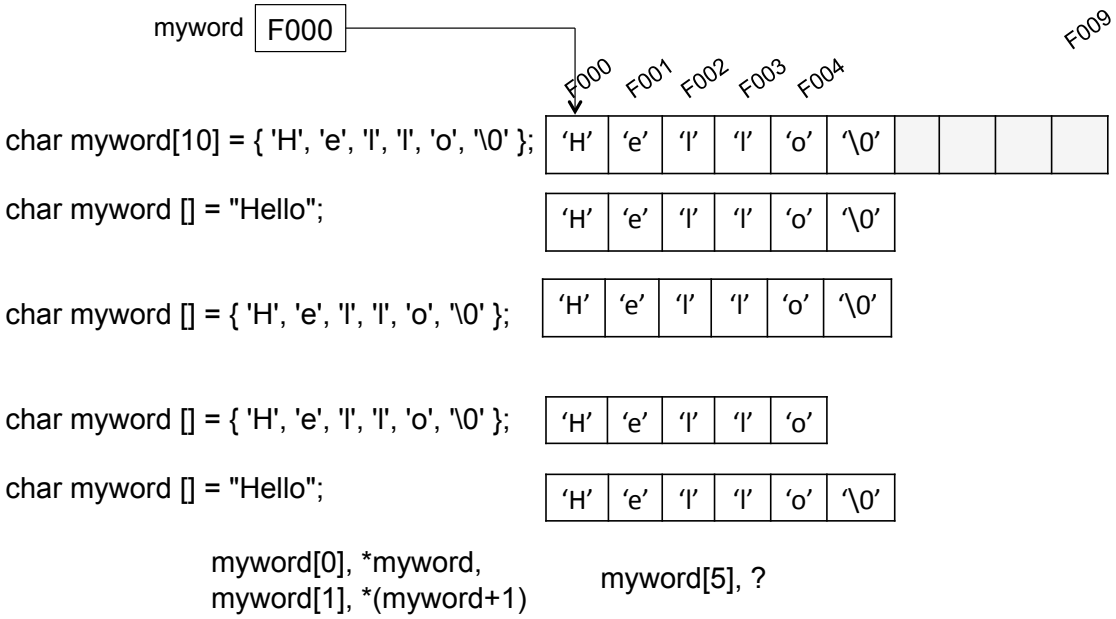
declaration	Accessing using array indexes	Accessing via pointer arithmetic	Internal mapping
char ch[10]; char *pch = &ch[0];	ch[3]	*(pch+3)	pch + 3*size of char
int x[10]; Int *px = &x[0];	x[5]	*(px+5)	px+5*size of int
int x[5][10]; int *px = &x[0][0];	x[3][7]	*(*(px+3)+7)	?

❖ what *pch++ means?

❖ accessing contents of pch then incrementing pch

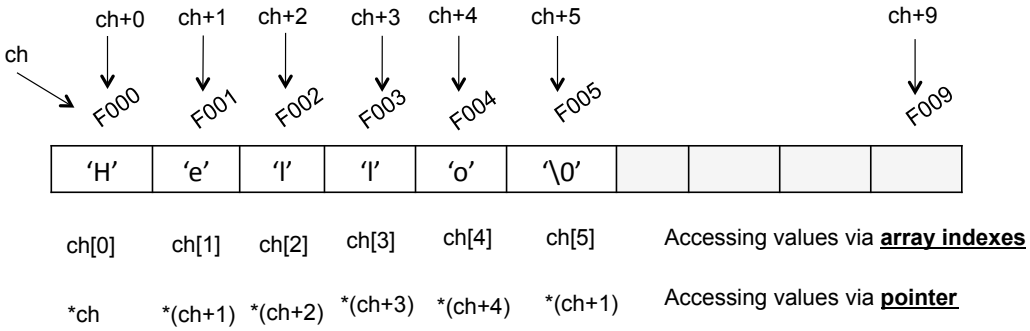
or

❖ incrementing pch then accessing contents of pch



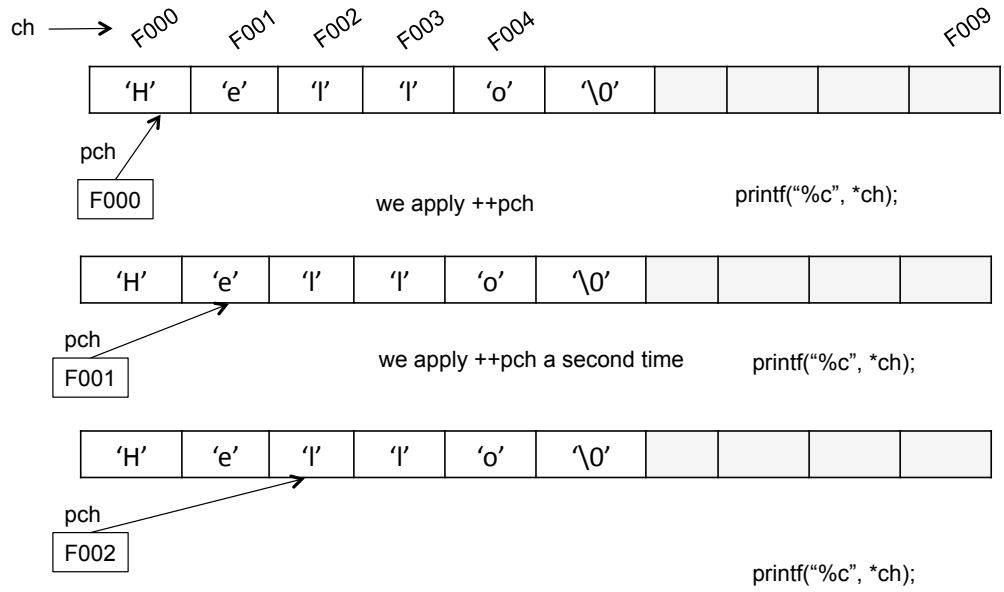
char ch[10] = { 'H', 'e', 'l', 'l', 'o', '\0' };

- the array name ch is a pointer
- always represents the address of the zeroth element of the array
- can’t be changed (i.e., ch++, ch+=1, etc. invalid)

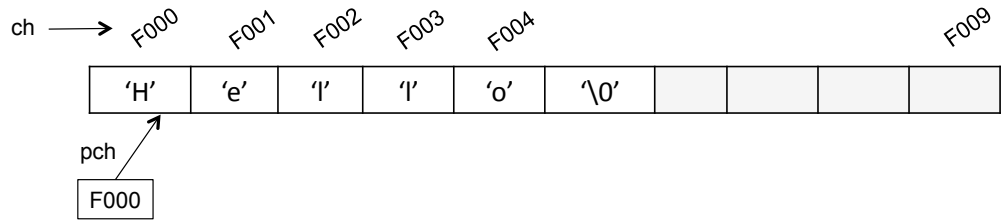


printf(“%c %c”, ch[3], *(ch+3));

```
char ch[10] = { 'H', 'e', 'l', 'l', 'o', '\0' };
char *pch = &ch;
```




```
char ch[10] = { 'H', 'e', 'l', 'l', 'o', '\0' };
char *pch = &ch;
```

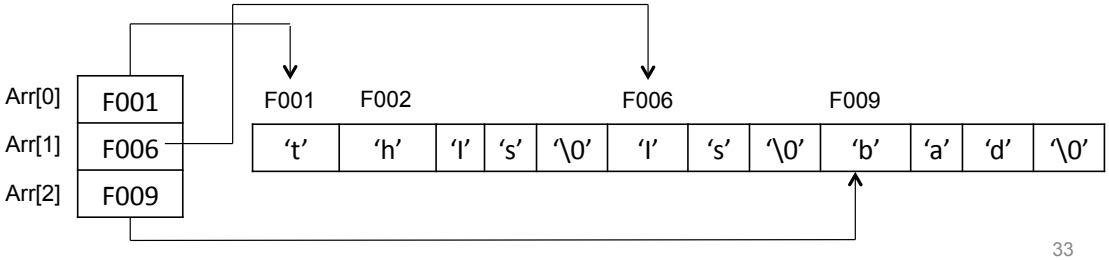




```
while(*pch != '\0')
  printf ("%c", *pch++);
```

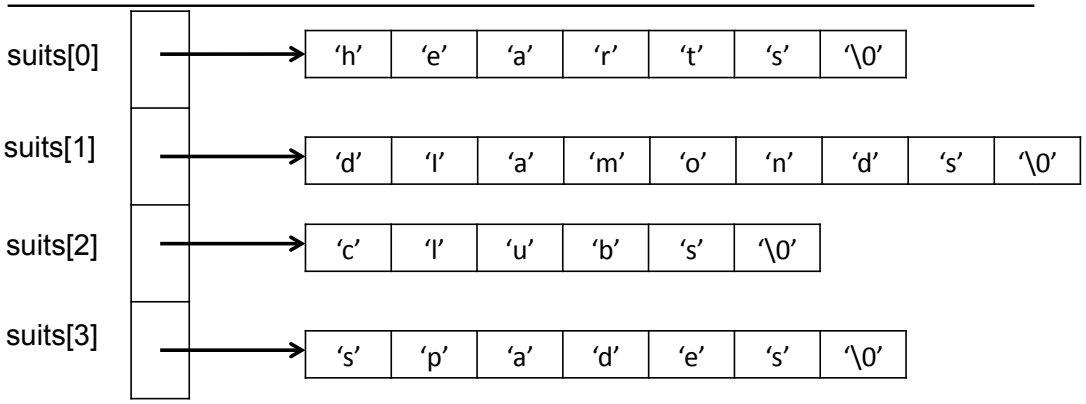
- a pointer to an array can be used like an array name

```
int i=0;
while (pch[i] != '\0')
  printf ("%c" , pch[i++]);
```

```
#include <stdio.h>
#include <conio.h>
void main() {
   char *arr[3] = { “this", “is", “bad" };

  printf("Array of String is = %s,%s,%s\n", arr[0], arr[1], arr[2]);

  getch();
}
```

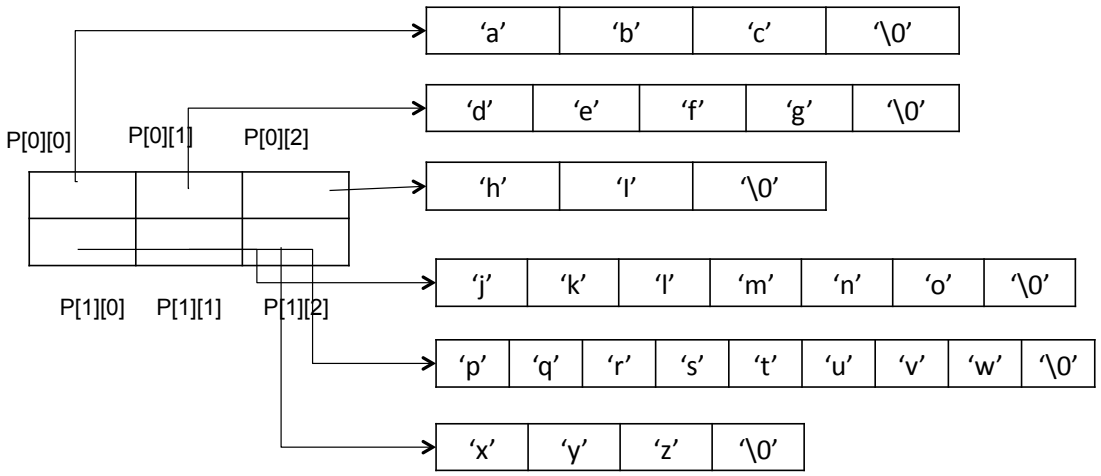


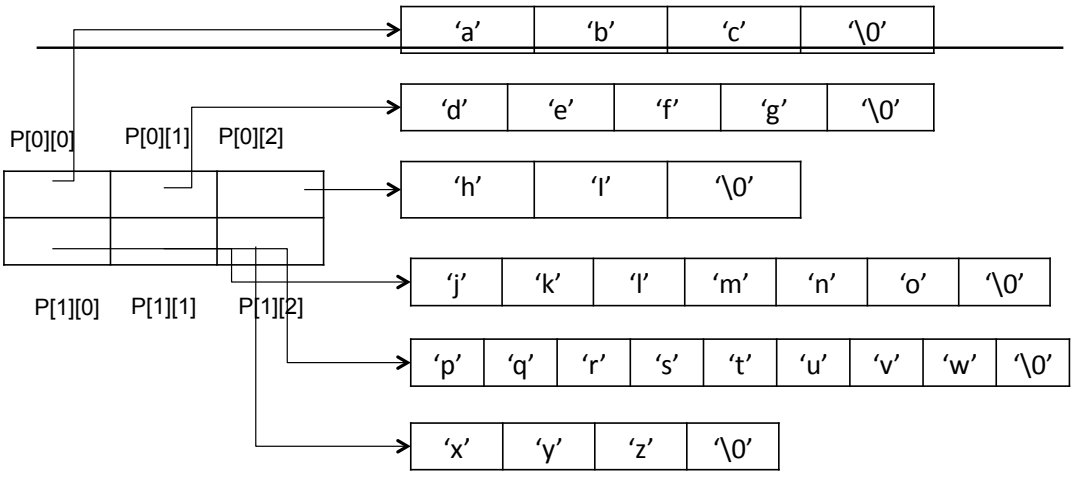
char *suits[4] = { “hearts”, “diamonds”, “clubs”, “spades” };



Expression	Equivalent Expression	Value
**suit	suit[0][0]	‘h’
*suit[2]	suit[2][0]	‘c’
*(suit[2]+3)	suit[2][3]	‘b’

```
char *p[2][3] = {{“abc”, “defg”, “hi”},
                 {“jklmno”, “pqrstuvw”, “xyz”}};
```

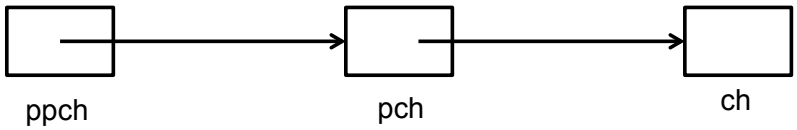




Expression	Equivalent Expression	Value
**(p[1]+2)	p[1][2][0]	‘x’
(*(*(p+1)+1))[7]	p[1][1][7]	‘w’

pointer to pointer

```
char    ch = 'h';       /* a character */
char    *pch;           /* a pointer to a character */
char    **ppch;         /* a pointer to a pointer to a character */
pch = &ch;
ppch = &pch;
```



If ch begins at A001 and pch at 00F1 in memory, what are the contents of ppch?

pointer to pointer

```
#include<stdio.h>

int main(){

    int x = 25;
    int *ptr = &x;          /*ptr is pointer */
    int **temp = &ptr;      /* temp is pointer to ptr */

    printf("%d %d %d", x, *ptr,**temp);

   return 0;
}
```

pointers to functions

- every function has an address in memory i.e., where its first instruction starts
- a pointer which keeps address of a function is known as function pointer
- a function can be called through its pointer as well


```
#include <stdio.h>
void foo(int arg);
typedef void FuncType(int);  /* define a new type */
int main(void)
{

  FuncType *func_ptr;    /*declare function pointer (fp)*/
  func_ptr = &foo;       /* get the address of the funstion*/
  (*func_ptr)(17);       /* call the function using fp*/

}
void foo(arg)
    { printf("foo got an arg of %d\n", arg); }
```

END Lab 02