NYU
New York University

CSCI-GA.1144-001

PAC II

# Lecture 6: C programming: Advanced

Mohamed Zahran (aka Z)
mzahran@cs.nyu.edu
http://www.mzahran.com

# Pointers:

## A very strong (and dangerous!) concept!

# What is “Memory”?

Memory is like a big table of numbered slots where bytes can be stored.

The number of a slot is its Address.
One byte Value can be stored in each slot.

Some “logical” data values span more than one slot, like the character string “Hello\n”

A Type names a logical meaning to a span of memory. Some simple types are:

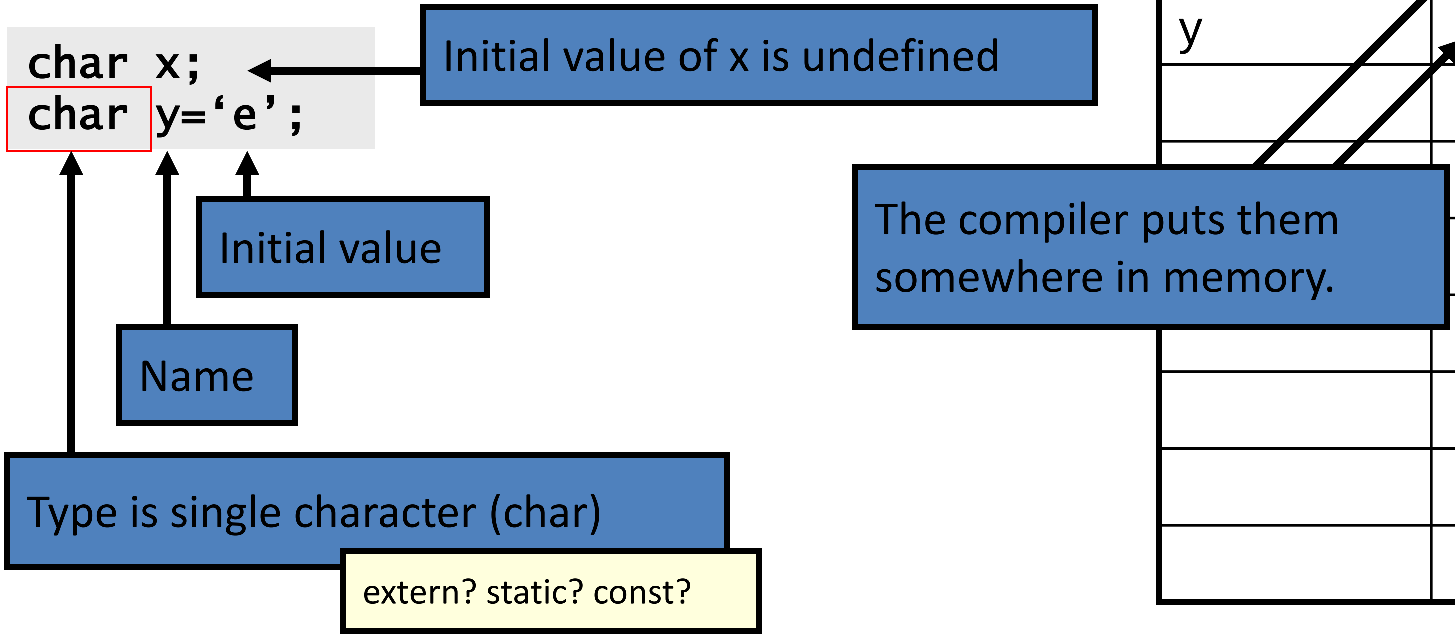
| | |
|---|---|
| `char` | a single character (1 slot) |
| `char [10]` | an array of 10 characters |
| `int` | signed 4 byte integer |
| `float` | 4 byte floating point |
| `int64_t` | signed 8 byte integer |

| Addr | Value |
|---|---|
| 0 | |
| 1 | |
| 2 | |
| 3 | |
| 4 | ‘H’ (72) |
| 5 | ‘e’ (101) |
| 6 | ‘l’ (108) |
| 7 | ‘l’ (108) |
| 8 | ‘o’ (111) |
| 9 | ‘\n’ (10) |
| 10 | ‘\0’ (0) |
| 11 | |
| 12 | |

# What is a Variable?

A Variable names a place in memory where you store a Value of a certain Type.

You first Define a variable by giving it a name and specifying the type, and optionally an initial value



```
char x;
char y='e';
```

Initial value of x is undefined

Initial value

Name

Type is single character (char)

extern? static? const?

The compiler puts them somewhere in memory.

| Symbol | Addr | Value |
|---|---|---|
| | 0 | |
| | 1 | |
| | 2 | |
| | 3 | |
| x | 4 | ? |
| y | 5 | 'e' (101) |
| | 6 | |
| | 7 | |
| | 8 | |
| | 9 | |
| | 10 | |
| | 11 | |
| | 12 | |

```
char x;
char y=‘e’;
```

What is y? → “101” pr ‘e’

What is &y? → 5

| Symbol | Addr | Value |
|---|---|---|
| | 0 | |
| | 1 | |
| | 2 | |
| | 3 | |
| x | 4 | ? |
| y | 5 | ‘e’ (101) |
| | 6 | |
| | 7 | |
| | 8 | |
| | 9 | |
| | 10 | |
| | 11 | |
| | 12 | |

# Pointers in C

- Addresses in memory
- Programs can manipulate addresses directly

**&***x* Evaluates to the address of the location of x

***x*** Evaluates to the value stored in x

# Multi-byte Variables

Different types consume different amounts of memory. Most architectures store data on “word boundaries”, or even multiples of the size of a primitive data type (int, char)

```
char x;
char y=‘e’;
int z = 0x01020304;
```

0x means the constant is written in hex

padding

An int consumes 4 bytes

| Symbol | Addr | Value |
|---|---|---|
| | 0 | |
| | 1 | |
| | 2 | |
| | 3 | |
| x | 4 | ? |
| y | 5 | ‘e’ (101) |
| | 6 | |
| | 7 | |
| z | 8 | 4 |
| | 9 | 3 |
| | 10 | 2 |
| | 11 | 1 |
| | 12 | |

# Pointer Validity

A Valid pointer is one that points to memory that your program controls. Using invalid pointers will cause non-deterministic behavior, and will often cause Linux to kill your process (SEGV or Segmentation Fault).

There are two general causes for these errors:

- Program errors that set the pointer value to a strange number
- Use of a pointer that was at one time valid, but later became invalid

Will ptr be valid or invalid?

```
char * get_pointer()
{
  char x=0;
  return &x;
}

{
  char * ptr = get_pointer();
  *ptr = 12;  /* valid? */
}
```

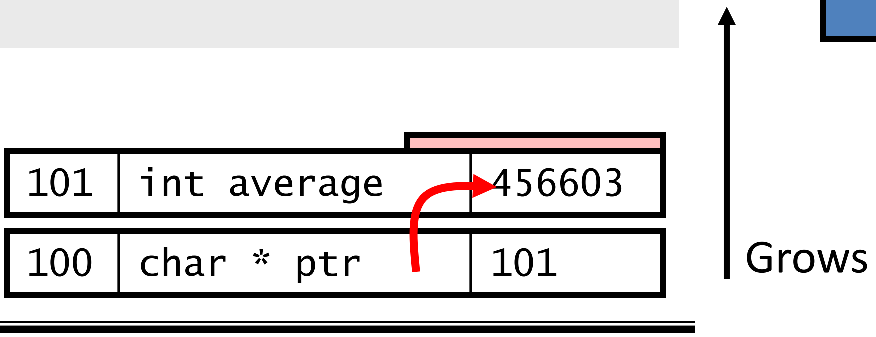
# Answer: Invalid!

A pointer to a variable allocated on the stack becomes invalid when that variable goes out of scope and the stack frame is “popped”. The pointer will point to an area of the memory that may later get reused and rewritten.

```
char * get_pointer()
{
    char x=0;
    return &x;
}

{
    char * ptr = get_pointer();
    *ptr = 12;  /* valid? */
    other_function();
}
```

But now, `ptr` points to a location that’s no longer in use, and will be reused the next time a function is called!



|     |              |        |
|-----|--------------|--------|
| 101 | int average  | 456603 |
| 100 | char * ptr   | 101    |

# Pointers and Functions

# First, Let's Discuss Function

- C is a procedural language
- A C program consists of functions calling each other

```
type_of_return_arg  function_name (comma_separated_arguments)
{
            Function_body
}
```

```
int add_numbers (int x, int y)
{
   return (x + y);
}

void print_numbers (int x)
{
    printf(“Result is: %d\n”, x);
}

int main()
{
   int m = 0, n = 0;

   printf(“Enter first number:\n”);
   scanf(“ %d “, &m);

   printf(“Enter second number:\n”);
   scanf(“ %d “, &n);

   print_numbers( add_numbers(m,n));
}
```

# Can a function modify its arguments?

What if we wanted to implement a function pow_assign() that *modified* its argument, so that these are equivalent:

```
float p = 2.0;
/* p is 2.0 here */
p = pow(p, 5);
/* p is 32.0 here */
```

⟺

```
float p = 2.0;
/* p is 2.0 here */
pow_assign(p, 5);
/* p is 32.0 here */
```

Would this work?

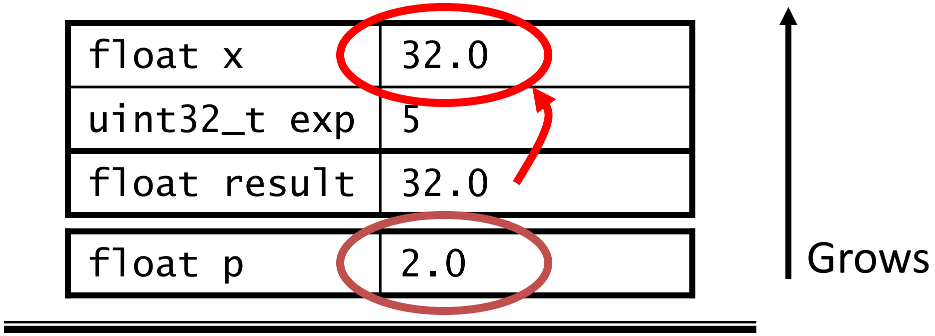
```
void pow_assign(float x, uint exp)
{
  float result=1.0;
  int i;
  for (i=0; (i < exp); i++) {
    result = result * x;
  }
  x = result;
}
```

# NO!

Remember the stack!

```
void pow_assign(float x, uint exp)
{
   float result=1.0;
   int i;
   for (i=0; (i < exp); i++) {
     result = result * x;
   }
   x = result;
}

{
   float p=2.0;
   pow_assign(p, 5);
}
```



| | |
|---|---|
| float x | 32.0 |
| uint32_t exp | 5 |
| float result | 32.0 |
| float p | 2.0 |

In C, all arguments are passed as values

But, what if the argument is the *address* of a variable?

# Passing Addresses

Recall our model for variables stored in memory

What if we had a way to find out the address of a symbol, and a way to reference that memory location by address?

```
address_of(y) == 5
memory_at[5] == 101
```

```
void f(address_of_char p)
{
  memory_at[p] = memory_at[p] - 32;
}
```

```
char y = 101;       /* y is 101 */
f(address_of(y));   /* i.e. f(5) */
/* y is now 101-32 = 69 */
```

| Symbol | Addr | Value |
|---|---|---|
| | 0 | |
| | 1 | |
| | 2 | |
| | 3 | |
| char x | 4 | ‘H’ (72) |
| char y | 5 | ‘e’ (101) |
| | 6 | |
| | 7 | |
| | 8 | |
| | 9 | |
| | 10 | |
| | 11 | |
| | 12 | |

# “Pointers”

This is exactly how “pointers” work.

```
“address of” or reference operator:  &
“memory_at” or dereference operator: *
```

```
void f(address_of_char p)
{
  memory_at[p] = memory_at[p] - 32;
}
```

```
char y = 101;       /* y is 101 */
f(address_of(y));   /* i.e. f(5) */
/* y is now 101-32 = 69 */
```

A “pointer type”: pointer to char

```
void f(char * p)
{
  *p = *p - 32;
}
```

```
char y = 101;       /* y is 101 */
f(&y);              /* i.e. f(5) */
/* y is now 101-32 = 69 */
```

Pointers are used in C for many other purposes:

- Passing large objects without copying them
- Accessing dynamically allocated memory
- Referring to functions

# Beware!

```
int *value (void)
{
  int i = 3;
  return &i;
}

void callme (void)
{
  int x = 35;
}

int main (void) {
  int *ip;
  ip = value ();
  printf (“*ip == %d\n", *ip);
  callme ();
  printf ("*ip == %d\n", *ip);
}
```

```
*ip == 3

*ip == 35
```

But it could really be anything!

# Pointers and Data Structure

- Strings
- Arrays
- Linked lists
- …

# Strings

- Series of characters treated as a single unit
- Can include letters, digits, and certain special characters (*, /, $)
- String literal (string constant) - written in double quotes
  - "Hello"
- Strings are arrays of characters
- Address of string is the address of first character

# Strings

- String declarations
  - Declare as a character array or a variable of type **`char *`**

    ```
    char color[] = "blue";
    char *colorPtr = "blue";
    ```

  - Remember that strings represented as character arrays end with `'\0'`
    - **`color`** has 5 elements
- Inputting strings
  - Use **`scanf`**

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

    - Copies input into **`word[]`**, which does not need **`&`** (because a string is a pointer)
  - Remember to leave space for `'\0'`

# Character Handling Library

- In `<ctype.h>`

| Prototype | Description |
|---|---|
| `int isdigit( int c )` | Returns **true** if **c** is a digit and **false** otherwise. |
| `int isalpha( int c )` | Returns **true** if **c** is a letter and **false** otherwise. |
| `int isalnum( int c )` | Returns **true** if **c** is a digit or a letter and **false** otherwise. |
| `int isxdigit( int c )` | Returns **true** if **c** is a hexadecimal digit character and **false** otherwise. |
| `int islower( int c )` | Returns **true** if **c** is a lowercase letter and **false** otherwise. |
| `int isupper( int c )` | Returns **true** if **c** is an uppercase letter; **false** otherwise. |
| `int tolower( int c )` | If **c** is an uppercase letter, **tolower** returns **c** as a lowercase letter. Otherwise, **tolower** returns the argument unchanged. |
| `int toupper( int c )` | If **c** is a lowercase letter, **toupper** returns **c** as an uppercase letter. Otherwise, **toupper** returns the argument unchanged. |
| `int isspace( int c )` | Returns **true** if **c** is a white-space character—newline (`'\n'`), space (`' '`), form feed (`'\f'`), carriage return (`'\r'`), horizontal tab (`'\t'`), or vertical tab (`'\v'`)—and **false** otherwise |
| `int iscntrl( int c )` | Returns **true** if **c** is a control character and **false** otherwise. |
| `int ispunct( int c )` | Returns **true** if **c** is a printing character other than a space, a digit, or a letter and **false** otherwise. |
| `int isprint( int c )` | Returns **true** value if **c** is a printing character including space (`' '`) and **false** otherwise. |
| `int isgraph( int c )` | Returns **true** if **c** is a printing character other than space (`' '`) and **false** otherwise. |

Each function receives a character (an **`int`**) or **`EOF`** as an argument

# String Conversion Functions

- Conversion functions
  - In `<stdlib.h>` (general utilities library)
  - Convert strings of digits to integer and floating-point values

| Prototype | Description |
|---|---|
| `double atof( const char *nPtr )` | Converts the string `nPtr` to `double`. |
| `int atoi( const char *nPtr )` | Converts the string `nPtr` to `int`. |
| `long atol( const char *nPtr )` | Converts the string `nPtr` to long `int`. |
| `double strtod( const char *nPtr, char **endPtr )` | Converts the string `nPtr` to `double.` |
| `long strtol( const char *nPtr, char **endPtr, int base )` | Converts the string `nPtr` to `long`. |
| `unsigned long strtoul( const char *nPtr, char **endPtr, int base)` | Converts the string `nPtr` to `unsigned long`. |

# Standard Input/Output Library Functions

- Functions in `<stdio.h>`
  - Used to manipulate character and string data

| Function prototype | Function description |
|---|---|
| `int getchar( void );` | Inputs the next character from the standard input and returns it as an integer. |
| `char *gets( char *s );` | Inputs characters from the standard input into the array **`s`** until a newline or end-of-file character is encountered. A terminating null character is appended to the array. |
| `int putchar( int c );` | Prints the character stored in **`c`**. |
| `int puts( const char *s );` | Prints the string **`s`** followed by a newline character. |
| `int sprintf( char *s, const char *format, ... );` | Equivalent to **`printf`**, except the output is stored in the array **`s`** instead of printing it on the screen. |
| `int sscanf( char *s, const char *format, ... );` | Equivalent to **`scanf`**, except the input is read from the array **`s`** instead of reading it from the keyboard. |

# String Manipulation Functions of the String Handling Library

- String handling library has functions to
  - Manipulate string data
  - Search strings
  - Determine string length

| Function prototype | Function description |
|---|---|
| `char *strcpy( char *s1, const char *s2 )` | Copies string **`s2`** into array **`s1`**. The value of **`s1`** is returned. |
| `char *strncpy( char *s1, const char *s2, size_t n )` | Copies at most **`n`** characters of string **`s2`** into array **`s1`**. The value of **`s1`** is returned. |
| `char *strcat( char *s1, const char *s2 )` | Appends string **`s2`** to array **`s1`**. The first character of **`s2`** overwrites the terminating null character of **`s1`**. The value of **`s1`** is returned. |
| `char *strncat( char *s1, const char *s2, size_t n )` | Appends at most **`n`** characters of string **`s2`** to array **`s1`**. The first character of **`s2`** overwrites the terminating null character of **`s1`**. The value of **`s1`** is returned. |

# Pointers and Arrays in C

- An array name by itself is an address, or pointer in C.
- When an array is declared, the compiler allocates sufficient space beginning with some base address to accommodate every element in the array.
- The base address of the array is the address of the first element in the array (index position 0).
  - &num[0] is the same as num

# Pointers and Arrays in C

- Suppose we define the following array and pointer:

  ```
  int   a[100],  *ptr;
  ```

  Assume that the system allocates memory bytes 400, 404, 408, ..., 796 to the array. Recall that integers are allocated 32 bits = 4 bytes.

  - The two statements: ptr = a; and ptr = &a[0]; are equivalent and would assign the value of 400 to ptr.

- Pointer arithmetic provides an alternative to array indexing in C.

  - The two statements: ptr = a + 1; and ptr = &a[1]; are equivalent and would assign the value of 404 to ptr.

# Pointers and Arrays in C

- Assuming the elements of the array have been assigned values, the following code would sum the elements of the array:

```
sum = 0;
for (ptr = a; ptr < &a[100]; ++ptr)
    sum += *ptr;
```

- Here is a way to sum the array:

```
sum = 0;
for (i = 0; i < 100; ++i)
    sum += *(a + i);
```

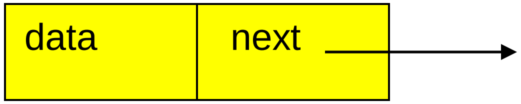
a[b] in C is just syntactic sugar for

*(a + b)

# Linked List Implementation/Coding Issues in C

- We can define structures with pointer fields that refer to the structure type containing them

```
struct list {
        int data;
        struct list *next;
}
```

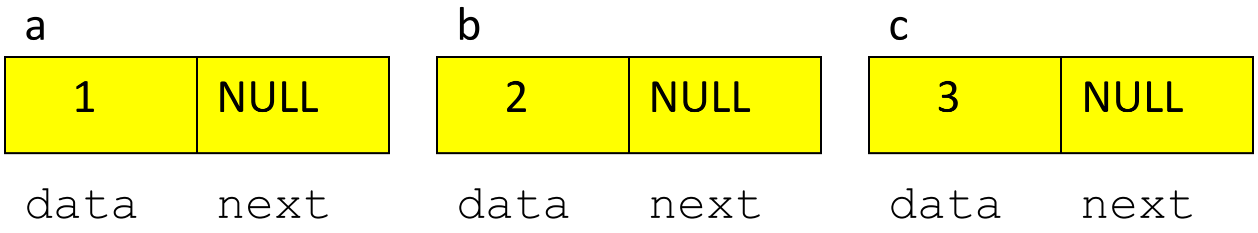


- The pointer variable next is called a *link.*
- Each structure is linked to a succeeding structure by way of the field next.
- The pointer variable next contains an address of either the location in memory of the successor struct list element or the special value NULL.

# Example

```
struct list a, b, c;

a.data = 1;
b.data = 2;
c.data = 3;
a.next = b.next = c.next = NULL;
```



a

1 NULL

data next

b

2 NULL

data next

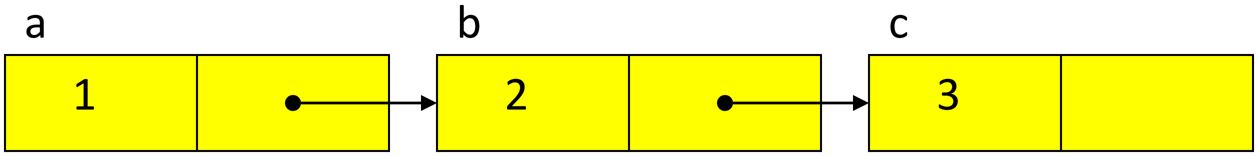
c

3 NULL

data next

# Example

- `a.next = &b;`
- `b.next = &c;`
- `a.next -> data` has value 2
- `a.next -> next -> data` has value 3
- `b.next -> next -> data` error !!



# Dynamic Memory Allocation

# Dynamic Memory Allocation

- So far all of our examples have allocated variables statically by defining them in our program. This allocates them in the stack.
- The ability for a program to obtain more memory space at execution time to hold new values, and to release space no longer needed.
- In C, functions *malloc* and *free*, and operator *sizeof* are essential to dynamic memory allocation.

# Dynamic Memory Operators *sizeof* and *malloc*

- Unary operator *sizeof* is used to determine the size in bytes of any data type.

  sizeof(double)     sizeof(int)

- Function *malloc* takes as an argument the number of bytes to be allocated and return a pointer of type void * to the allocated memory. (A void * pointer may be assigned to a variable of any pointer type. ) It is normally used with the *sizeof* operator.
- calloc is very similar to malloc but initializes the bytes allocated to zero.

# Dynamic Memory Allocation

sizeof() reports the size of a type in bytes

For details:
$ man calloc

```
int * alloc_ints(size_t requested_count)
{
  int * big_array;
  big_array = (int *)calloc(requested_count, sizeof(int));
  if (big_array == NULL) {
    printf(“can’t allocate %d ints: %m\n”, requested_count);
    return NULL;
  }

  /* now big_array[0] .. big_array[requested_count-1] are
   * valid and zeroed. */
  return big_array;
}
```

calloc() allocates memory for N elements of size k

Returns NULL if can’t alloc

%m ?

It’s OK to return this pointer. It will remain valid until it is freed with free()

# Caveats with Dynamic Memory

Dynamic memory is useful. But it has several caveats:

Whereas the stack is automatically reclaimed, dynamic allocations must be tracked and free()'d when they are no longer needed. With every allocation, be sure to plan how that memory will get freed. Losing track of memory is called a "memory leak".

Whereas the compiler enforces that reclaimed stack space can no longer be reached, it is easy to accidentally keep a pointer to dynamic memory that has been freed. Whenever you free memory you must be certain that you will not try to use it again. It is safest to erase any pointers to it.

Because dynamic memory always uses pointers, there is generally no way for the compiler to statically verify usage of dynamic memory. This means that errors that are detectable with static allocation are not with dynamic

# Some Common Errors and Hints

sizeof() can take a variable reference in place of a type name.  This gurantees the right allocation, but don't accidentally allocate the sizeof() the *pointer* instead of the *object*!

malloc() allocates n bytes

```
/* allocating a struct with malloc() */
struct my_struct *s = NULL;
s = (struct my_struct *)malloc(sizeof(*s));  /* NOT sizeof(s)!! */
if (s == NULL) {
  printf(stderr, “no memory!”);
  exit(1);
}
```

Always check for NULL.. Even if you just exit(1).

```
memset(s, 0, sizeof(*s));
```

malloc() does not zero the memory, so you should memset() it to 0.

```
/* another way to initialize an alloc’d structure: */
struct my_struct init = {
  counter: 1,
  average: 2.5,
  in_use: 1
};

/* memmove(dst, src, size) (note, arg order like assignment) */
memmove(s, &init, sizeof(init));
```

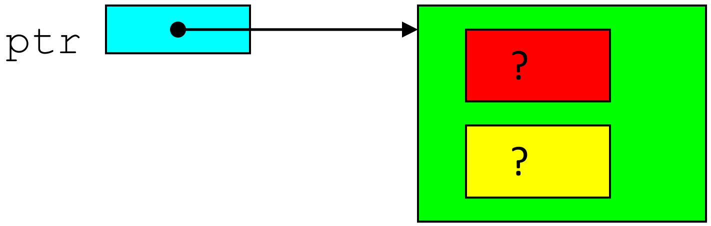
memmove is preferred because it is safe for shifting buffers

```
/* when you are done with it, free it! */
free(s);
s = NULL;
```

# Dynamic Memory Operators in C Example

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

ptr = (struct node *)    /*type casting */
         malloc(sizeof(struct node));
```



# The *Free* Operator in C

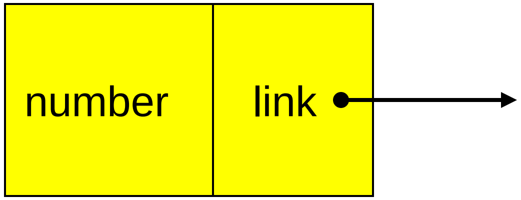
- Function *free* deallocates memory- i.e. the memory is returned to the system so that the memory can be reallocated in the future.

# Examples of the Nodes of a Linked List

- A node in a linked list is a structure that has at least two fields. One of the fields is a data field; the other is a pointer that contains the address of the next node in the sequence.

- A node with one data field:

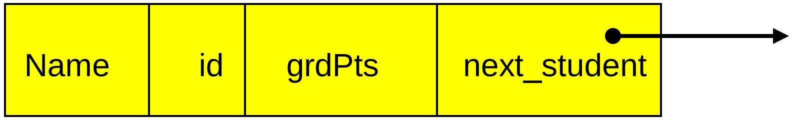
```
struct node{
    int number;
    struct node * link;
};
```



# The Nodes of a Linked List - Examples

- A node with three data fields:

```
struct student{
   char name[20];
   int id;
   double grdPts;
   struct student
        *next_student;}
```



Name

id

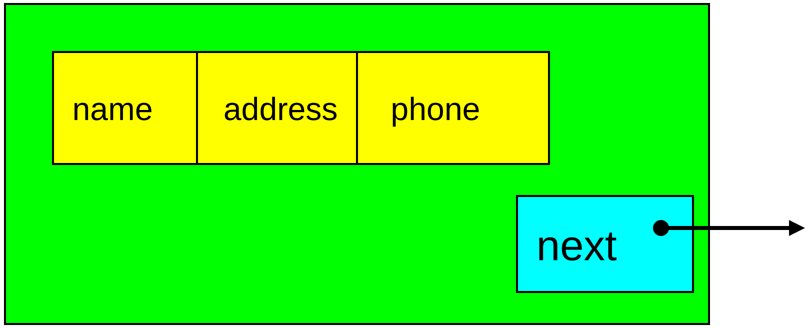
grdPts

next_student

# The Nodes of a Linked List - Examples

- A structure in a node:

```
struct person{
    char name[20];
    char address[30];
    char phone[10];
};
struct person_node{
    struct person data;
    struct person_node   *next;
};
```



# Basic Operations on a Linked List

1. Add a node.
2. Delete a node.
3. Search for a node.
4. Traverse (walk) the list. Useful for counting operations or aggregate operations.

# Adding Nodes to a Linked List

**Adding a Node**

There are four steps to add a node to a linked list:

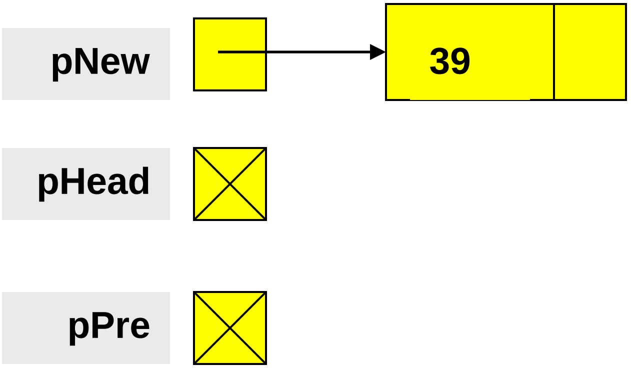
1. Allocate memory for the new node.
2. Determine the insertion point (you need to know only the new node's predecessor (`pPre`)
3. Point the new node to its successor.
4. Point the predecessor to the new node.

Pointer to the predecessor (`pPre`) can be in one of two states:

- it can contain the address of a node (i.e. you are adding somewhere after the first node – in the middle or at the end)
- it can be NULL (i.e. you are adding either to an empty list or at the beginning of the list)

# Adding Nodes to an Empty Linked List

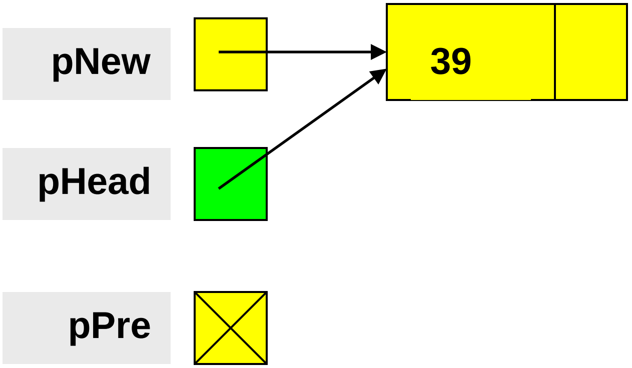
## Before:



## Code:

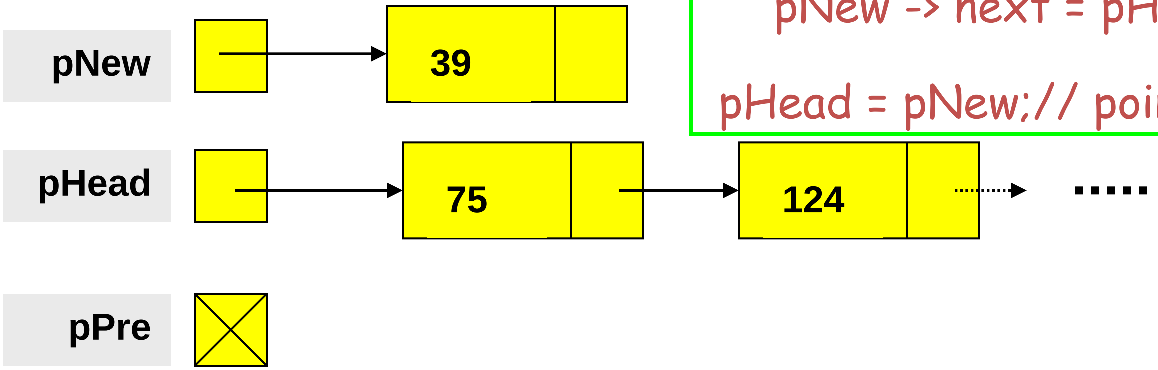
```
pNew -> next = pHead;  // set link to NULL
pHead  =  pNew;//  point  list  to  first  node
```

## After:



# Adding a Node to the Beginning of a Linked List

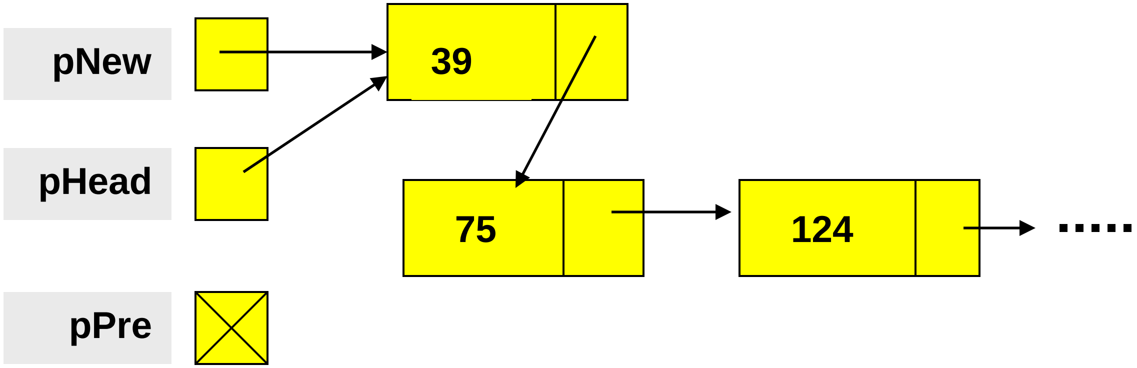
Before:



Code (same):

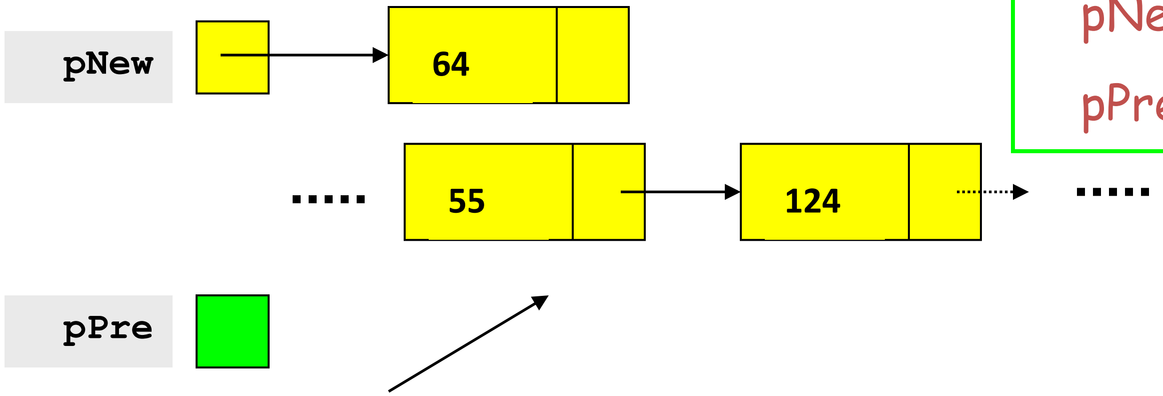
```
pNew -> next = pHead;  // set link to NULL
pHead = pNew;// point list to first node
```

After:



# Adding a Node to the Middle of a Linked List

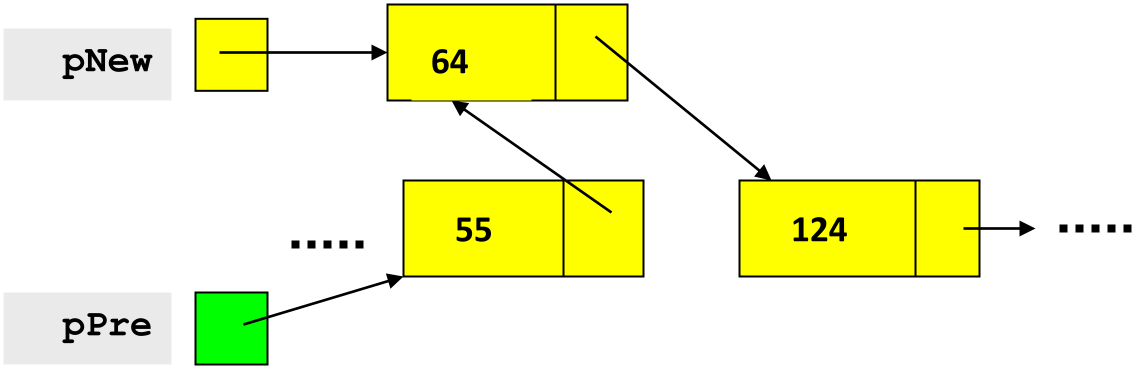
Before:



## Code

```
pNew -> next = pPre -> next;
pPre -> next = pNew;
```

After:

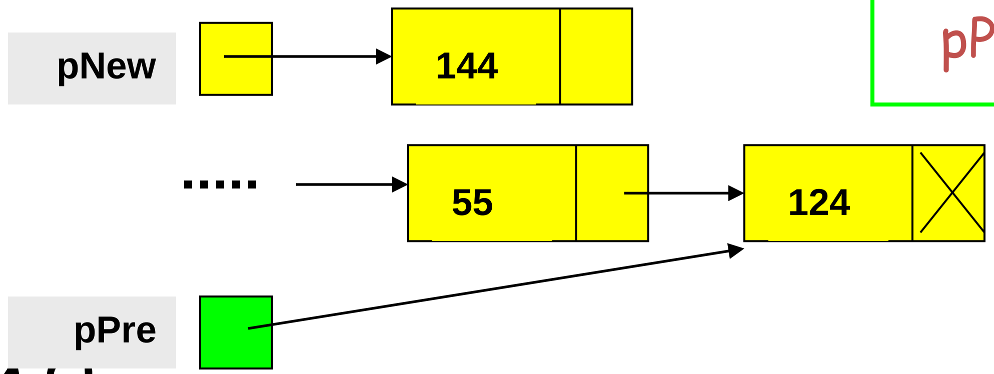


# Adding a Node to the End of a Linked List

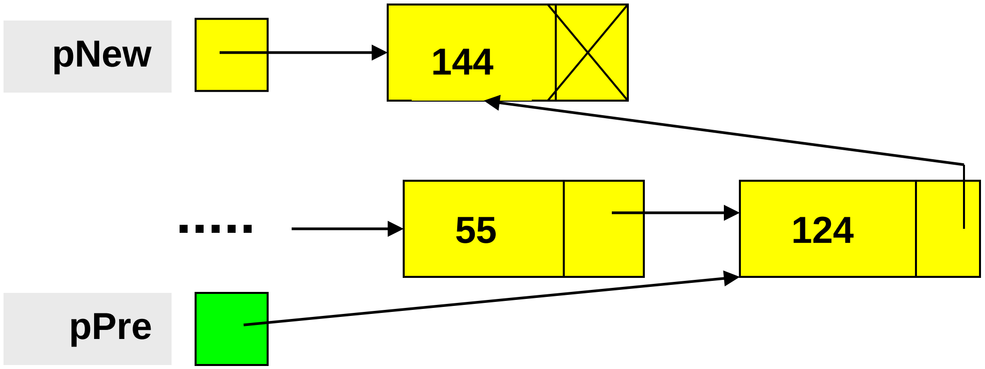
Before:

## Code

```
pNew -> next = NULL;
pPre -> next = pNew;
```



After:



# Inserting a Node Into a Linked List

- Given the head pointer (pHead), the predecessor (pPre) and the data to be inserted (item).
- Memory must be allocated for the new node (pNew) and the links properly set.

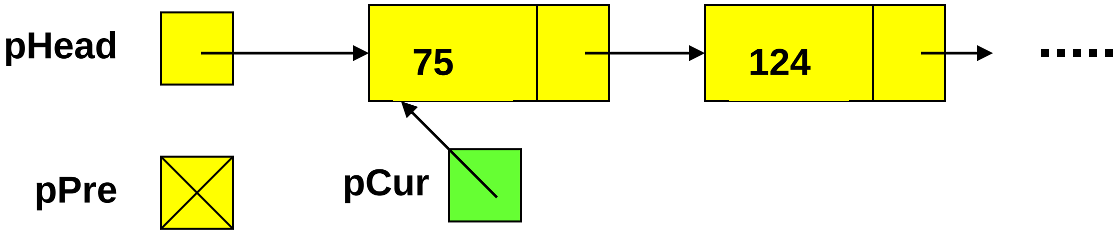
```
//insert a node into a linked list
   struct node *pNew;
   pNew = (struct node *)  malloc(sizeof(struct node));
   pNew -> data = item;
   if (pPre == NULL){
       //add before first logical node or to an empty list
       pNew -> next = pHead;
       pHead = pNew;
   }
   else {
        //add in the middle or at the end
       pNew -> next = pPre -> next;
       pPre -> next = pNew;
  }
```

# Deleting a Node from a Linked List

- Deleting a node requires that we logically remove the node from the list by changing various links and then physically deleting the node from the list (i.e., return it to the heap).
- Any node in the list can be deleted. Note that if the only node in the list is to be deleted, an empty list will result. In this case the head pointer will be set to NULL.
- To logically delete a node:
  - First locate the node itself (pCur) and its logical predecessor (pPre).
  - Change the predecessor's link field to point to the deleted node's successor (located at pCur -> next).
  - Recycle the node using the free() function.

# Deleting the First Node from a Linked List

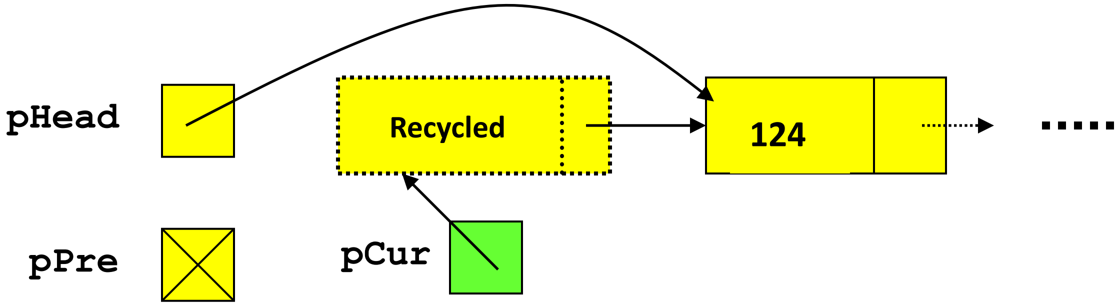
Before:



Code:

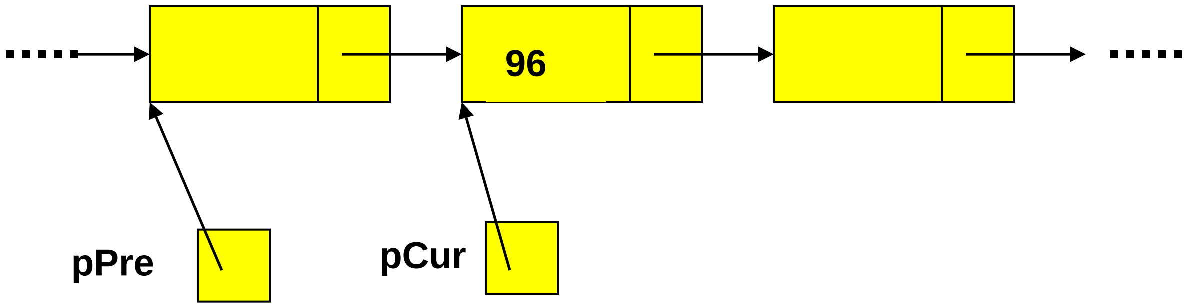
```
pHead = pCur -> next;
free(pCur);
```

After:



# Deleting a Node from a Linked List – General Case

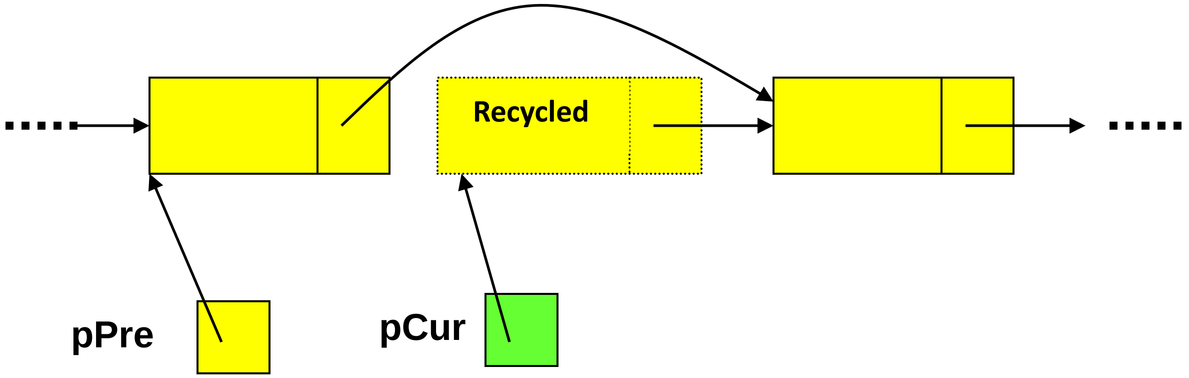
**Before:**



Code:

```
pPre -> next= pCur -> next;
  free(pCur);
```

**After:**



# Deleting a Node From a Linked List

- Given the head pointer (pHead), the node to be deleted (pCur), and its predecessor (pPre), delete pCur and free the memory allocated to it.

```
//delete a node from a linked list
if (pPre == NULL)
         //deletion is on the first node of the list
        pHead = pCur -> next;
else
         //deleting a node other than the first node of the list
        pPre -> next = pCur -> next;
free(pCur).
```

# Searching a Linked List

- Notice that both the insert and delete operations on a linked list must search the list for either the proper insertion point or to locate the node corresponding to the logical data value that is to be deleted.

```
//search the nodes in a linked list
pPre = NULL;
pCur = pHead;
//search until the target value is found or the end of the list is reached
while (pCur != NULL && pCur -> data != target) {
    pPre = pCur;
    pCur = pCur -> next;
}
//determine if the target is found or ran off the end of the list
if (pCur != NULL)
   found = 1;
else
   found = 0;
```

# Traversing a Linked List

- List traversal requires that all of the data in the list be processed. Thus each node must be visited and the data value examined.

```
//traverse a linked list
Struct node *pWalker;
pWalker = pHead;
printf(“List contains:\n”);
while (pWalker != NULL){
   printf(“%d ”, pWalker -> data);
   pWalker = pWalker -> next;
}
```

# The main() Function

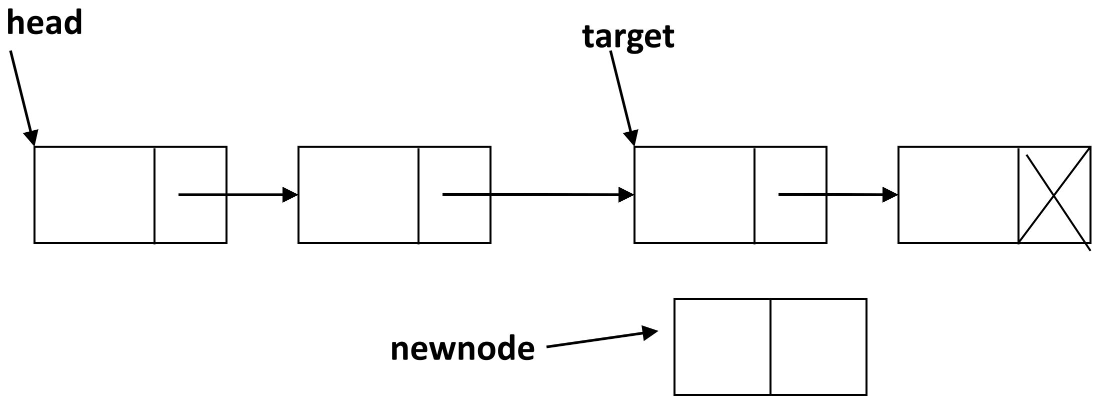
```
int main ( int argc, char *argv[] )
```

is the **argument count**.
It is the number of arguments passed into the program from the command line, including the name of the program.

- array of character pointers
- listing of all the arguments
- argv[0] is the name of the program

# Practice Problems Using a Linked List

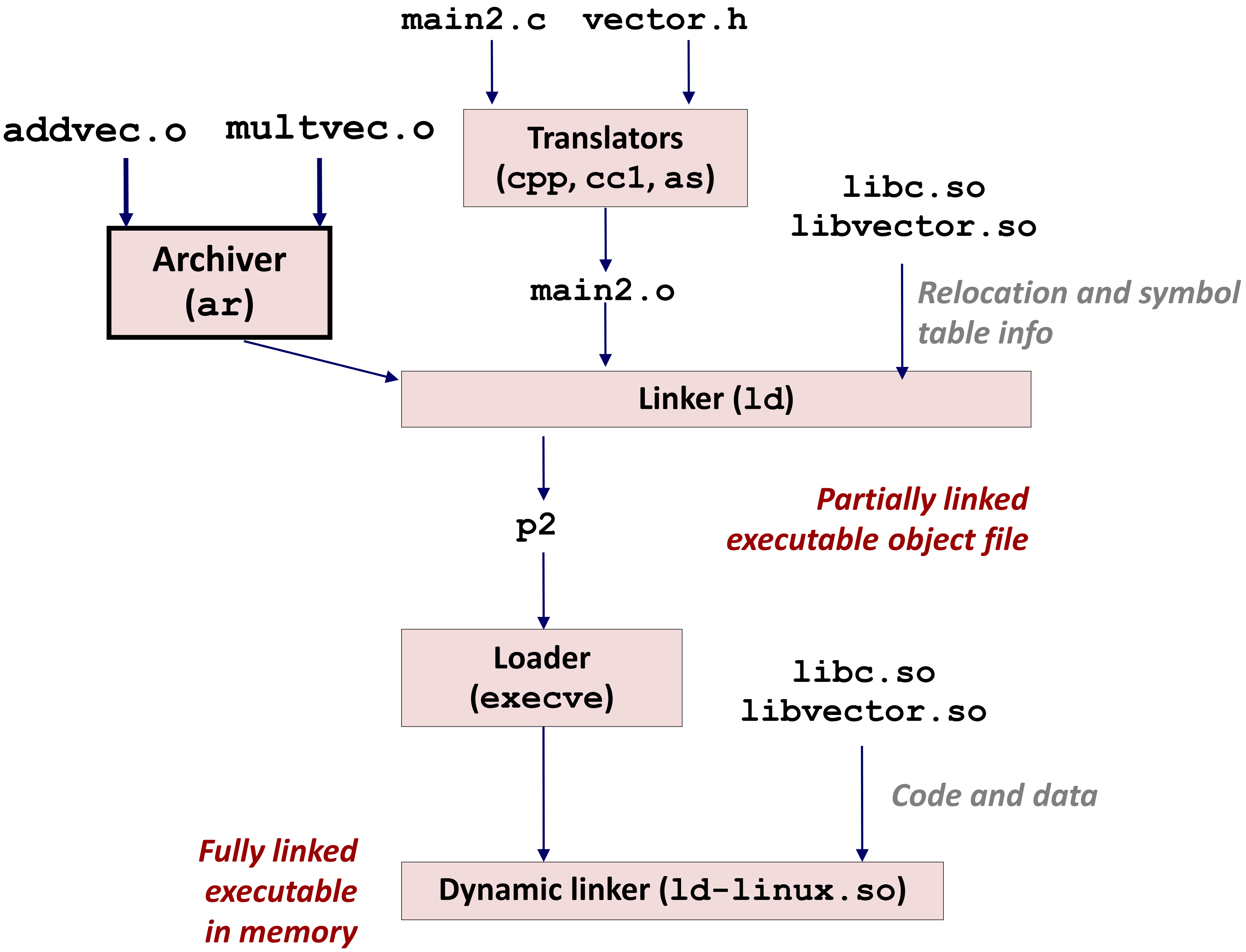
1. Write a recursive function that will search a list for a specific item, returning NULL if it is not found and a pointer to the node containing the value if it is found.
2. Write a function that will insert a node pointed to by *newnode* before the node pointed to by *target*.



# More C Practice Problems

- Write a function that computes the integer log, base 2, of a number using only shifts.

# Compilation - Linking - Loading



main2.c vector.h

addvec.o multvec.o

Translators (cpp, cc1, as)

Archiver (ar)

main2.o

libc.so libvector.so

*Relocation and symbol table info*

Linker (ld)

p2

*Partially linked executable object file*

Loader (execve)

libc.so libvector.so

*Code and data*

*Fully linked executable in memory*

Dynamic linker (ld-linux.so)

# Conclusions

- By now you must have good knowledge of the C programming language.
  - We have not covered all the functions available in all libraries in C, but for sure you can very easily complete the pieces of the puzzle whenever you need to.
- It is a great exercise to see how the C code is translated to assembly (gcc –S is your friend).
- Always keep the big picture in  your mind.