**C-NOTES**

Table of Contents

[1 C 4](#_Toc511639947)

[1.1 General 4](#_Toc511639948)

[1.1.1 Sizeof() 5](#_Toc511639949)

[1.1.2 Comma operator 7](#_Toc511639950)

[1.1.3 Extern variable 7](#_Toc511639951)

[1.1.4 Function with no arguments 7](#_Toc511639952)

[1.1.5 Volatile 8](#_Toc511639953)

[1.2 Switch 8](#_Toc511639954)

[1.3 Structures 9](#_Toc511639955)

[1.4 Functions 10](#_Toc511639956)

[1.4.1 \_\_attribute\_\_ 12](#_Toc511639957)

[1.4.2 Others about functions 13](#_Toc511639958)

[1.4.3 Fucntion notes 15](#_Toc511639959)

[1.4.4 Function Declaration: 15](#_Toc511639960)

[1.4.5 Function that are executed before and after main() 17](#_Toc511639961)

[1.4.6 Implementing function overloading in C 19](#_Toc511639962)

[1.4.7 Function returning multiple values 20](#_Toc511639963)

[1.4.8 Static Functions in C: 20](#_Toc511639964)

[1.4.9 Exit(), abort(), assert() 21](#_Toc511639965)

[1.4.10 TAIL RECURSION vs HEAD RECURSION 21](#_Toc511639966)

[1.5 Variable no of arguments 21](#_Toc511639967)

[1.6 Pre and Post increment 22](#_Toc511639968)

[1.7 Others 23](#_Toc511639969)

[1.8 Dynamic memory management 23](#_Toc511639970)

[1.8.1 How to free memory allocated using “malloc” without using “free” 24](#_Toc511639971)

[1.9 Memory Layout of C programs 25](#_Toc511639972)

[1.10 Various stages of C code compilation 25](#_Toc511639973)

[1.11 Accessing command line arguments without using argv,argc 27](#_Toc511639974)

[2 ARRAYS 28](#_Toc511639975)

[3 POINTERS 29](#_Toc511639976)

[4 LINKED LIST 32](#_Toc511639977)

[4.1 Given only a pointer/reference to a node to be deleted in a singly linked list, how do you delete it? 32](#_Toc511639978)

[4.2 Delete a node 32](#_Toc511639979)

[4.3 Deleting a key at a given position 34](#_Toc511639980)

[4.4 Length of a linked list – RECURSIVE 35](#_Toc511639981)

[4.5 Search an element in a linked list – RECURSIVE 36](#_Toc511639982)

[4.6 Swap nodes without swapping data 36](#_Toc511639983)

[4.7 Print middle element of a list 39](#_Toc511639984)

[4.8 Nth node from end of Linked List 39](#_Toc511639985)

[4.9 Delete an entire linked list 40](#_Toc511639986)

[4.10 Detect loop in a linked list 41](#_Toc511639987)

[4.11 Reverse a linked list 42](#_Toc511639988)

[4.12 Generic linked list 42](#_Toc511639989)

[4.13 Function to check if a single linked list is palindrome 43](#_Toc511639990)

[4.14 Intersection point of two linked lists 46](#_Toc511639991)

[4.15 Recursive function to print reverse of a Linked list 47](#_Toc511639992)

[4.16 Remove duplicates from a sorted LL 48](#_Toc511639993)

[4.17 Remove duplicates from a unsorted LL 49](#_Toc511639994)

[4.18 Pairwise swap elements of a linked list 49](#_Toc511639995)

[4.19 Merge two sorted Linked list 50](#_Toc511639996)

[4.20 Insert a node in a sorted linked list 50](#_Toc511639997)

[5 C logics 51](#_Toc511639998)

[5.1 Macro to concatenate a string 51](#_Toc511639999)

[5.2 Set, clear and check a bit 52](#_Toc511640000)

[6 C programs 52](#_Toc511640001)

[6.1 Pre-processor’s 52](#_Toc511640002)

[Output: 57](#_Toc511640003)

[Programs 60](#_Toc511640004)

[6 .1 Pointers 63](#_Toc511640005)

[6 .2 Imp Links 64](#_Toc511640006)

[6 .3 Main() is not the actual start point in C programs 64](#_Toc511640007)

[Memory layout of a C program 67](#_Toc511640008)

[6.3.1 Shared libraries and malloc/calloc/realloc/free/alloca 69](#_Toc511640009)

[6.3.2 Setjmp and longjmp 70](#_Toc511640010)

[6 .4 Strlen vs sizeof 72](#_Toc511640011)

[7 DOUBTS 72](#_Toc511640012)

[8 Pragma 75](#_Toc511640013)

[9 FILE OPERATIONS 75](#_Toc511640014)

[10 OPERATING SYSTEMS 77](#_Toc511640015)

[10.1 Links 77](#_Toc511640016)

[10.2 Notes 77](#_Toc511640017)

**Index of Tables**

<http://www.geeksforgeeks.org/a-nested-loop-puzzle/>

<http://www.geeksforgeeks.org/pass-2d-array-parameter-c/>

<http://www.geeksforgeeks.org/dynamically-allocate-2d-array-c/>

In E drive==> E/C/C\_Pointers

# C

## General

In x86\_64, pointers are 64-bit wide and integers are 32-bits wide, <http://www.geeksforgeeks.org/importance-of-function-prototype-in-c/>

|  |
| --- |
| #include <errno.h>  #include <stdio.h>    int main(int argc, char \*argv[])  {  FILE \*fp;    fp = fopen(argv[1], "r");  if (fp == NULL) {  fprintf(stderr, "%s\n", strerror(errno));  return errno;  }    printf("file exist\n");    fclose(fp);    return 0;  }  **In x86\_64, pointers are 64-bit wide and integers are 32-bits wide, that’s why while returning from function, the returned address gets truncated (i.e. 32-bit wide address, which is size of integer on x86\_64) which is invalid and when we try to dereference this address, the result is segmentation fault.** |

Write a macro PRINTX which prints X:

#define PRINT(x) (#x)

int main()

{

printf("%s",PRINT(x));

return 0;

}

Print “%” using below.

printf("%c", '%');

printf("%s", "%");

printf("%%");

### Sizeof()

Implement Your Own sizeof

#define my\_sizeof(type) (char \*)(&type+1)-(char\*)(&type)

* Sizeof(int) is 2bytes on 32bit machine, 4bytes on 64bit machine
* On office laptop => unix 64bit => sizeof(void \*) = 8, sizeof(int)= 4
* Sizeof is an operator

Printing a variable byte by byte :

#include <stdio.h>

typedef unsigned char \*byte\_pointer;

/\*show bytes takes byte pointer as an argument

and prints memory contents from byte\_pointer

to byte\_pointer + len \*/

void show\_bytes(byte\_pointer start, int len)

{

int i;

for (i = 0; i < len; i++)

printf(" %.2x", start[i]);

printf("\n");

}

void show\_int(int x)

{

show\_bytes((byte\_pointer) &x, sizeof(int));

}

void show\_float(float x)

{

show\_bytes((byte\_pointer) &x, sizeof(float));

}

void show\_pointer(void \*x)

{

show\_bytes((byte\_pointer) &x, sizeof(void \*));

}

/\* Program to test above functions \*/

int main()

{

int i = 1;

float f = 1.0;

int \*p = &i;

show\_float(f);

show\_int(i);

show\_pointer(p);

show\_int(i);

getchar();

return 0;

}

|  |
| --- |
| Int a= 10;  Double b= 20.2;  Pf("%d", sizeof(a));  Pf("%d", sizeof a);//without parenthesis for sizeof a .  Pf("%d", sizeof(a+b));  Pf("%f", sizeof a+b);  **OUTPUT : 4 , 4, 8, 24.2000**  Addition between INT variable and double variable the result will be "double" => so size of a+b =8  **Explanation : sizeof w.o parenthesis works fine bcos sizeof is an operator not a function......sizeof thinks parenthesis as we r clubbing operands or expression....Expression : sizeof (a+b) ; is expression for sizeof....after adding int n double res is double...so sizeof expects o/p of expression....as it is double it prints 8.** |
|  |
|  |
|  |
|  |

### Comma operator

|  |
| --- |
| Associativity of comma operator is from left to right, but = operator has higher precedence than comma operator.  int i;  i = 1, 2, 3; printf("i = %d\n", i); OUTPUT = 1  i = (1, 2, 3); printf("i = %d\n", i); OUTPUT = 3 |

**Comma operator => right to left precedence**

**Comma separator => no specific order**

### Extern variable

### Function with no arguments

|  |
| --- |
| #include <stdio.h>  int func()  {    printf("\n inside func");  }  int main()  {  printf("\n in main function");  func(1,2,3);    return 0;  } |

### Volatile

Volatile is a qualifier in C and not a storage class

### Alloca

### Stack alignment

### Frame pointer, stack pointer

### Reading command line arguments without using argc,argv

## Switch

switch(value) ==> this value can be either int or character literal or enum or an arithmetic expression like 5\*2 but NOT Double, not float

Can’t use “goto” inside switch case.

Exmp: float x=10;

if(10==x) returns false because in C all constant integers by default are considered as **double**

case label 🡺 This label has to be only integer constant exmp: case 10.00 🡺ERROR

|  |  |
| --- | --- |
| **int i = 10;**  **int c = 10;**  **switch(c)**  **{**  **case i: // not a "const int" expression**  **printf("Value of c = %d", c);**  **break;**    **}**  **ERROR** | **Const int i = 10;**  **int c = 10;**  **switch(c)**  **{**  **case i: // a "const int" expression**  **printf("Value of c = %d", c);**  **break;**  **}**  **COMPILES** |
| case 1+1: printf(“\ninside 1+1”); break;  case 2: printf(“\ninside 2”); break;  OUTPUT : ERROR : duplicate case value | case (3/2): printf(“\ninside 1”); break;  OUTPUT : Works , this turns into case 1 |

## Structures

Struct student

{

char name[10];

int rollnum;

};  this is only declaration so no memory allocation happens

So now struct student abc;  Only now memory allocation happens

Typedef

Pointer to a struct

A struct with char\*, int\*

Example :

|  |
| --- |
| In order to pack a structure :  [#pragma](https://www.programmersheaven.com/search?Search=%23pragma&Mode=like) pack(push) [#pragma](https://www.programmersheaven.com/search?Search=%23pragma&Mode=like) pack(1) struct Packaging { char c; int i; } Packaging; [#pragma](https://www.programmersheaven.com/search?Search=%23pragma&Mode=like) pack(pop)    Size of above/below struct will be 5bytes  OR  struct Packaging { char c; int i; }\_\_attribute\_\_((packed))Packaging; |
| void ioread32\_rep(void \_\_iomem \*port, void \*dst, unsigned long count)  {  if (unlikely((unsigned long)dst & 0x3)) {  while (count--) {  ***struct S { int x \_\_attribute\_\_((packed)); };***  ***((struct S \*)dst)->x*** = ioread32(port);  dst += 4;  } |

## Functions

**Every function has a return type. If a function doesn’t return any value, then void is used as return type.**

**In C language, when we don’t provide prototype of function, the compiler assumes that function returns an integer.**

|  |
| --- |
| **#include <stdio.h>**  **#include <string.h>**  **int \* foo(){**  **int a = 5;**  **return &a;**  **}**  **void boo(){**  **int a = 7;**  **}**  **int main(){**  **int \* p = foo();**  **printf("%d\n",\*p);**  **boo();**  **printf("%d\n",\*p);**  **}** |
| **#include <stdio.h>**  **#include <string.h>**  **int\* func(int x)**  **{**  **x=2030;**  **printf("\n so in func %u \n",&x);**  **return &x;**  **}**    **main()**  **{**  **int zx=100;**  **int \*y;**  **printf("\n so %d \n",zx);**  **y = func(zx);**  **printf("\n so %d \n",\*y);**  **printf("\n so in main %u \n",y);**  **}** |

|  |
| --- |
| #include<stdio.h>float\* display(int,int);int max=5;int main(){  float \*(\*ptr)(int,int);ptr=display;(\*ptr)(2,2);printf("\n %u",ptr);ptr=ptr+1;printf("\n %u",ptr);return 0;}  float \*display(int x,int y)  {  float f;  f=x+y+max;  return &f;  }  OUTPUT : 4195666  4195667 BECAUSE INCREMENTING ADDRESS OF FUNCTION POINTER INCREASES ITS VALUE BY 1 ONLY |

Return type for sqrt or squareroot kind of functions is always double.

Overloading of functions is possible in C : using int foo(void \*arg1, int arg2);

### \_\_attribute\_\_

The keyword \_\_attribute\_\_ allows you to specify special attributes when making a declaration. This keyword is followed by an attribute specification inside double parentheses. The following attributes are currently defined for functions on all targets: noreturn, noinline, always\_inline, pure, const, format, format\_arg, no\_instrument\_function, section, constructor, destructor, used, unused, deprecated, weak, malloc, and alias.

To invoke a function before or after main :

|  |
| --- |
| /\* Apply the constructor attribute to myStartupFun() so that it      is executed before main() \*/  void myStartupFun (void) \_\_attribute\_\_ ((constructor));      /\* Apply the destructor attribute to myCleanupFun() so that it     is executed after main() \*/  void myCleanupFun (void) \_\_attribute\_\_ ((destructor)); |

\_\_attribute\_\_ works only in GNU C compilers., also for GCC compilers

**\_\_attribute\_\_** applies to function *declarations*, not *definitions*

Multiple **\_\_attributes\_\_ can be used:**

**extern void die(const char \*format, ...)**

**\_\_attribute\_\_((noreturn))**

**\_\_attribute\_\_((format(printf, 1, 2)))**;

/\*or\*/

extern void die(const char \*format, ...)

**\_\_attribute\_\_((noreturn, format(printf, 1, 2)))**;

Attributes are a compile-time annotation

\_\_attribute\_\_((always\_inline)) : Generally, functions are not inlined unless optimization is specified. For functions declared inline, this attribute inlines the function even if no optimization level was specified.

For variables : \_\_attribute\_\_((const))

Many functions do not examine any values except their arguments, and have no effects except the return value. Basically this is just slightly more strict class than the pure attribute above, since function is not allowed to read global memory. Note that a function that has pointer arguments and examines the data pointed to must not be declared const. Likewise, a function that calls a non-const function usually must not be const. It does not make sense for a const function to return void.

### Others about functions

Nested functions are allowed in C

A function can be declared multiple times in a file, but it has to defined ONLY ONCE.

Exmp :

#include <stdio.h>

int \*m();

void main()

{

int \*k = m();

printf("hello ");

printf("%d", k[0]); }

int \*m()

{

int a[2] = {5, 8};

LOCAL VARIBALE SO WILL BE STORED IN STACK

return a;

}

OUTPUT : hello followed by garbage value

Function pointer can be passed as argument to another function, but not like this exmp : int wow(void func());

When a function is invoked, function arguments are pushed from right to left followed by return variable followed by local variables onto STACk

Exmp : int func(char x, float f)

{

Double d;

}

Onto stack : First f , x, int(return variable), d etc….

### Fucntion notes

A function is a set of statements that take inputs, do some specific computation and produces output.

The idea is to put some commonly or repeatedly done task together and make a function, so that instead of writing the same code again and again for different inputs, we can call the function.

|  |
| --- |
| Below is a simple C program to demonstrate functions in C. |
| #include <stdio.h>  int max(int x, int y)  {     if (x > y)       return x;     else       return y;  }  int main(void)  {     int a = 10, b = 20;     // Calling above function to find max of 'a' and 'b'     int m = max(a, b);     printf("m is %d", m);     return 0;  } |
| OUTPUT : m is 20 |

### Function Declaration:

Function declaration tells compiler about number of parameters function takes, data-types of parameters and return type of function. Putting parameter names in function declaration is optional in function declaration, but it is necessary to put them in definition.

|  |
| --- |
| Example:  // A function that takes two integers as parameters  // and returns an integer  int max(int, int);  // A function that takes a char and an int as parameters  // and returns an integer  int fun(char, int); |

**Points to remember:**

It is always recommended to declare a function before it is used.

C allows to declare and define a function in same file or in separate file. This is especially needed in case of library functions.

The library functions are declared in header files and defined in library files.

**Parameter Passing to functions:**

The parameters passed to function are called ***actual parameters***.

The parameters received by function are called ***formal parameters***.

There are two most popular ways to pass parameters.

**Pass by Value:**  Values of actual parameters are copied to function’s formal parameters and the two types of parameters are stored in different memory locations. So any changes made inside functions are not reflected in actual parameters of caller.

**Pass by Reference:** Both actual and formal parameters refer to same locations, so any changes made inside the function are actually reflected in actual parameters of caller.

|  |  |
| --- | --- |
| Call by value | Call by Reference |
| #include <stdio.h>  void fun(int x)  {    x = 30;  }  int main(void)  {     int x = 20;     fun(x);     printf("x = %d", x);     return 0;  } | # include <stdio.h>  void fun(int \*ptr)  {     \*ptr = 30;  }  int main()  {   int x = 20;   fun(&x);   printf("x = %d", x);     return 0;  } |
| OUTPUT: X= 20 | OUTPUT : X=30 |

**Points to Remember:**

Every function has a return type. If a function doesn’t return any value, then void is used as return type.

In C, functions can return any type except arrays and functions. We can get around this limitation by returning pointer to array or pointer to function.

Empty parameter list in C mean that the parameter list is not specified and function can be called with any parameters. In C, it is not a good idea to declare a function like fun(). To declare a function that can only be called without any parameter, we should use “void fun(void)”.

In C language, when we don’t provide prototype of function, the compiler assumes that function returns an integer.

Implicit return type of a function is int: In C, if we do not specify a return type, compiler assumes an implicit return type as int. However, C99 standard doesn’t allow return type to be omitted even if return type is int. This was allowed in older C standard C89.

In C, if a function is called before its declaration, the **compiler assumes return type of the function as int**. **What about parameters?** compiler assumes nothing about parameters. Therefore, the compiler will not be able to perform compile-time checking of argument types and arity when the function is applied to some arguments.

**Prototype of a Function:**

Function prototype specifies the input/output interlace to the function i.e. what to give to the function and what to expect from the function. Prototype of a function is also called signature of the function.

The Function prototype serves the following purposes –

It tells the return type of the data that the function will return.

It tells the number of arguments passed to the function.

It tells the data types of the each of the passed arguments.

Also it tells the order in which the arguments are passed to the function.

**What if one doesn’t specify the function prototype?**

If one doesn’t specify the function prototype, the behavior is specific to C standard (either C90 or C99) that the compilers implement. Up to C90 standard, C compilers assumed the return type of the omitted function prototype as int. And this assumption at compiler side may lead to unspecified program behavior.

Later C99 standard specified that compilers can no longer assume return type as int. Therefore, C99 became more restrict in type checking of function prototype. But to make C99 standard backward compatible, in practice, compilers throw the warning saying that the return type is assumed as int. But they go ahead with compilation. Thus, it becomes the responsibility of programmers to make sure that the assumed function prototype and the actual function type matches.

### Function that are executed before and after main()

<http://www.geeksforgeeks.org/functions-that-are-executed-before-and-after-main-in-c/>

http://www.geeksforgeeks.org/executing-main-in-c-behind-the-scene/

With GCC family of C compilers, we can mark some functions to execute before and after main(). So some startup code can be executed before main() starts, and some cleanup code can be executed after main() ends. For example, in the following program, myStartupFun() is called before main() and myCleanupFun() is called after main().

|  |
| --- |
| #include<stdio.h>  /\* Apply the constructor attribute to myStartupFun() so that it is executed before main() \*/  void myStartupFun (void) \_\_attribute\_\_ ((constructor));  /\* Apply the destructor attribute to myCleanupFun() so that it is executed after main() \*/  void myCleanupFun (void) \_\_attribute\_\_ ((destructor));  /\* implementation of myStartupFun \*/  void myStartupFun (void)  {     printf ("startup code before main()\n");  }  /\* implementation of myCleanupFun \*/  void myCleanupFun (void)  {     printf ("cleanup code after main()\n");  }  int main (void)  {     printf ("hello\n");     return 0;  } |
| OUTPUT:  startup code before main()  hello  cleanup code after main() |

**Executing main() in C/C++ – behind the scene:**

From C/C++ programming perspective, the program entry point is main() function. From the perspective of program execution, however, it is not. Prior to the point when the execution flow reaches to the main(), calls to few other functions are made, which setup arguments, prepare environment variables for program execution etc.

[**Executable and Linkable Format (**](https://en.wikipedia.org/wiki/Executable_and_Linkable_Format)**ELF**[**) file**](https://en.wikipedia.org/wiki/Executable_and_Linkable_Format)**:**

The executable file created after compiling a C source code is a [Executable and Linkable](https://en.wikipedia.org/wiki/Executable_and_Linkable_Format) Format [(ELF) file](https://en.wikipedia.org/wiki/Executable_and_Linkable_Format).

Every ELF file have a ELF header where there is a **e\_entry** field which contains the program memory address from which the execution of executable will start. This memory address point to the **\_start()** function.

After loading the program, loader looks for the e\_entry field from the ELF file header. [Executable and Linkable](https://en.wikipedia.org/wiki/Executable_and_Linkable_Format) Format [(ELF)](https://en.wikipedia.org/wiki/Executable_and_Linkable_Format) is a common standard file format used in UNIX system for executable files, object code, shared libraries, and core dumps.

|  |
| --- |
| example.c  int main()  {    return(0);  } |
| gcc eaxmple.c -o example |
| objdump -f example |
| startup\_func:     file format elf64-x86-64  architecture: i386:x86-64, flags 0x00000112:  EXEC\_P, HAS\_SYMS, D\_PAGED  start address 0x0000000000400440 |

**The role of start function():**

The \_start() function prepare the input arguments for another function **\_libc\_start\_main()** which will be called next. This is prototype of **\_libc\_start\_main()** function. Here we can see **the arguments which were prepared by \_start() function.**

|  |
| --- |
| int \_\_libc\_start\_main(int (\*main) (int, char \* \*, char \* \*), /\* address of main function\*/  int argc, /\* number of command line args\*/  char \*\* ubp\_av, /\* command line arg array\*/  void (\*init) (void), /\* address of init function\*/  void (\*fini) (void), /\* address of fini function\*/  void (\*rtld\_fini) (void), /\* address of dynamic linker fini function \*/  void (\* stack\_end) /\* end of the stack address\*/  ); |

**The role of \_libc\_start\_main() function:**

Preparing environment variables for program execution.

Calls **\_init()** function which performs initialization before the main() function start.

Register **\_fini()** and **\_rtld\_fini()** functions to perform cleanup after program terminates.

After all the prerequisite actions has been completed, \_libc\_start\_main() calls the main() function.

**Writing a main function without main:**

Situation where \_start() function is used : <https://stackoverflow.com/questions/3379190/avoiding-the-main-entry-point-in-a-c-program>

|  |
| --- |
| main() is nothing but a agreed term for startup code. We can have any name for startup code it doesn’t necessarily have to be “main”. As \_start() function by default calls main(), we have to change it if we want to execute our custom startup code. We can override the \_start() function to make it call our custom startup code not main(). |
| #include<stdio.h>  #include<stdlib.h>  void \_start()  {     int x = my\_fun(); //calling custom main function     exit(x);  }  int my\_fun() // our custom main function  {     printf("Hello world!\n");     return 0;  } |
| To force compiler to not use it’s own implementation of \_start()  gcc -nostartfiles -o nomain nomain.c |
| ./nomain  OUTPUT: Hello world! |

**return statement vs exit() in main():**

When exit(0) is used to exit from program, destructors for locally scoped non-static objects are not called. But destructors are called if return 0 is used.

|  |  |
| --- | --- |
| #include<iostream>  #include<stdio.h>  #include<stdlib.h>  using namespace std;  class Test {  public:   Test() {     printf("Inside Test's Constructor\n");   }   ~Test(){     printf("Inside Test's Destructor");     getchar();   }  };  int main() {   Test t1;   // using exit(0) to exit from main   exit(0);  } | #include<iostream>  #include<stdio.h>  #include<stdlib.h>  using namespace std;  class Test {  public:   Test() {     printf("Inside Test's Constructor\n");   }   ~Test(){     printf("Inside Test's Destructor");   }  };  int main() {   Test t1;    // using return 0 to exit from main   return 0;  } |
| OUTPUT:  Inside Test’s Constructor | OUTPUT:  Inside Test’s Constructor  Inside Test’s Destructor |

### Implementing function overloading in C

Function overloading is a feature of a programming language that allows one to have many functions with same name but with different signatures.

C (not Object Oriented Language) doesn’t support this feature. However, one can achieve the similar functionality in C indirectly. One of the approach is as follows.

|  |
| --- |
| #include<stdio.h>  int func\_over(void\* arg1, int arg2) ;  int main()  {  char c = 'a' ;  int num = 10 ;  float floating = 3.42124;  func\_over( &c, 0 ) ; // arg1 is pointer to char  func\_over( &num, 1 ) ; // arg1 is pointer to integer  func\_over( &floating, 2 ) ; // arg1 is pointer to float  }  int func\_over( void\* arg1, int arg2)  {  switch (arg2)  {  case 0 : printf ( "ARG1 is char = %c\n", \*((char\*)arg1) ) ; break ;  case 1 : printf ( "ARG1 is int = %d\n", \*((int \*)arg1) ) ; break ;  case 2 : printf ( "ARG1 is float = %f\n", \*((float \*)arg1) ) ; break ;  default : printf ("Error: Wrong type casting\n") ;  }    return 0 ;  } |

### Function returning multiple values

|  |
| --- |
| #include<stdio.h>  struct global\_return {  int b;  char ch ;  float fp ;  }return\_mul\_val ;  struct global\_return\* function( int i, char a, float f)  {  return\_mul\_val.b = i ;  return\_mul\_val.ch = a ;  return\_mul\_val.fp = f ;  return (&return\_mul\_val);  }  int main()  {  int i=12 ;  char a = 'a' ;  float f = 3.1234 ;  struct global\_return \*in\_main ;  in\_main = function( i, a,f) ;  printf ("%d, %c, %f\n", in\_main->b, in\_main->ch, in\_main->fp ) ;  } |

### Static Functions in C:

In C, functions are global by default. The “*static*” keyword before a function name makes it static. Access to static functions is restricted to the file where they are declared. Therefore, when we want to restrict access to functions, we make them static. Another reason for making functions static can be reuse of the same function name in other files.

|  |  |
| --- | --- |
| static void fun1(void)  {   puts("fun1 called");  } | int main(void)  {   fun1();   getchar();   return 0;  } |
| If we compile the above code with command “gcc file2.c file1.c”, we get the error “undefined reference to `fun1’” . This is because fun1() is declared static in file1.c and cannot be used in file2.c. | |

### Exit(), abort(), assert()

**exit():**

void exit(int status) ;

exit() terminates the process normally.

status: Status value returned to the parent process. Generally, a status value of 0 or EXIT\_SUCCESS indicates success, and any other value or the constant EXIT\_FAILURE is used to indicate an error. exit() performs following operations.

Flushes unwritten buffered data.

Closes all open files.

Removes temporary files.

Returns an integer exit status to the operating system.

The C standard atexit() function can be used to customize exit() to perform additional actions at program termination.

**abort():**

Void abort(void);

Unlike exit() function, abort() may not close files that are open. It may also not delete temporary files and may not flush stream buffer. Also, it does not call functions registered with atexit().

This function actually terminates the process by raising a SIGABRT signal, and your program can include a handler to intercept this signal.

**assert():**

void assert(int expression);

If expression evaluates to 0 (false), then the expression, sourcecode filename, and line number are sent to the standard error, and then abort() function is called. If the identifier NDEBUG (“no debug”) is defined with #define NDEBUG then the macro assert does nothing.

Common error outputting is in the form:

Assertion failed: expression, file filename, line line-number

|  |
| --- |
| #include<assert.h>  #include<stdio.h>  void open\_record(char \*record\_name)  {     assert(record\_name != NULL);  }  int main(void)  {    open\_record(NULL);  } |
| OUTPUT: a.out: assert.c:6: open\_record: Assertion `record\_name != ((void \*)0)' failed.  Aborted (core dumped). |

### TAIL RECURSION vs HEAD RECURSION

## Variable no of arguments

Stdarg.h

va\_list

va\_start

va\_arg(va\_list,int) OR va\_arg(va\_list,double) BUT va\_arg(va\_list,float) is ERROR

va\_end

<http://c.learncodethehardway.org/book/ex25.html>

C supports variable numbers of arguments. But C does not provide way for finding out total number of arguments passed. User has to handle this in one of the following ways:

By passing first argument as count of arguments.

By passing last argument as NULL (or 0).

Using some printf (or scanf) like mechanism where first argument has placeholders for rest of the arguments.

|  |
| --- |
| /\* Demonstrating vaiable arguement method in C which uses first argument *arg\_count* to hold count of other arguments\*/  #include <stdarg.h>  #include <stdio.h>  // this function returns minimum of integer numbers passed.  First  // argument is count of numbers.  int min(int arg\_count, ...)  {   int i;   int min, a;   // va\_list is a type to hold information about variable arguments   va\_list ap;   // va\_start must be called before accessing variable argument list   va\_start(ap, arg\_count);   // Now arguments can be accessed one by one using va\_arg macro   // Initialize min as first argument in list   min = va\_arg(ap, int);   // traverse rest of the arguments to find out minimum   for(i = 2; i <= arg\_count; i++) {     if((a = va\_arg(ap, int)) < min)       min = a;   }   //va\_end should be executed before the function returns whenever   // va\_start has been previously used in that function   va\_end(ap);   return min;  }  int main()  {    int count = 5;    // Find minimum of 5 numbers: (12, 67, 6, 7, 100)    printf("Minimum value is %d", min(count, 12, 67, 6, 7, 100));    getchar();    return 0;  } |

## Pre and Post increment

++\*p ==> first \*p then ++ of \*p

\*p++ ==> first p++ then \*

\*++p ==> first ++p then \*

postfix has higher priority than prefix

Prefix and \* have same precedence so from right to left , associativity.

## Others

* printf(" %d ",printf("geeks” );

OUTPUT : 6 geeks PRINTF==> right to left

* int i;

i=1,2,3;

printf("%d",i);

* Implement sizeof :

#define my\_sizeof(type) (char \*)(&type+1)-(char\*)(&type)

* The maximum number of arguments that can be passed in a single function are **253**
* Value of a cons tint can be changed by using a pointer

Example:

Const int 1=10;

Int \*ptr=&I;

\*ptr=55;

Printf(“%d”,i);

i value is changed to 55 now.

## Dynamic memory management

Free(ptr) another way 🡺 realloc(ptr,0);

Ptr = (int\*)Malloc(10); another way 🡺 ptr=realloc(NULL,10);

Always better to do this way 🡺 free(ptr); ptr=NULL;

TO achieve calloc functionality using malloc :

Int \*ptr=(int\*)Malloc(10); memset(ptr,10,0);

|  |
| --- |
| The allocation routines are usually implemented with the sbrk(2) system call. This system call  expands (or contracts) the heap of the process. (Refer to Figure 7.6.) A sample implementation of  malloc and free is given in Section 8.7 of Kernighan and Ritchie [1988].  Although sbrk can expand or contract the memory of a process, most versions of malloc and free  never decrease their memory size. The space that we free is available for a later allocation, but the  freed space is not usually returned to the kernel; that space is kept in the malloc pool.  It is important to realize that most implementations allocate a little more space than is requested and  use the additional space for record keepingthe size of the allocated block, a pointer to the next  allocated block, and the like. This means that writing past the end of an allocated area could  overwrite this record-keeping information in a later block. These types of errors are often  catastrophic, but difficult to find, because the error may not show up until much later. Also, it is  possible to overwrite this record keeping by writing before the start of the allocated area.  Writing past the end or before the beginning of a dynamically-allocated buffer can corrupt more than  internal record-keeping information. The memory before and after a dynamically-allocated buffer can  potentially be used for other dynamically-allocated objects. These objects can be unrelated to the  code corrupting them, making it even more difficult to find the source of the corruption. |

### How to free memory allocated using “malloc” without using “free”

|  |
| --- |
| int \*ptr;  ptr = (int \*) malloc( 10\*sizeof(int) );  TO FREE THE ABOVE ALLOCATED MEMORY :  ptr = realloc( ptr, 0 );  ptr = NULL; |

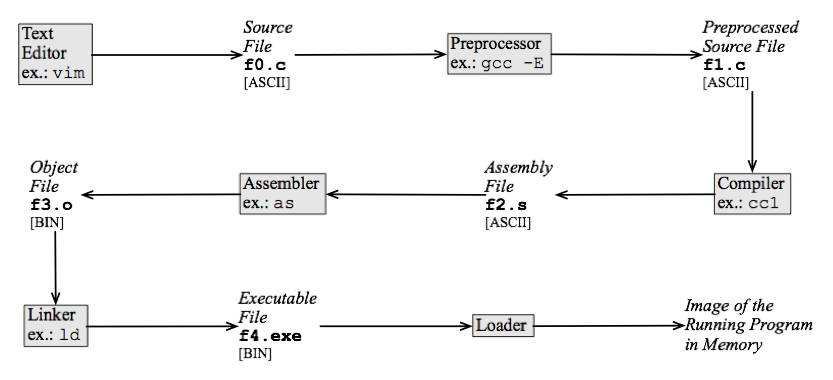
## Memory Layout of C programs

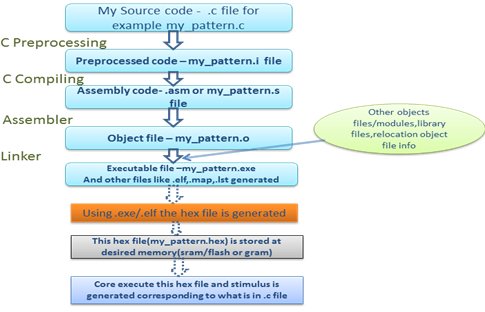
Mainly contains :

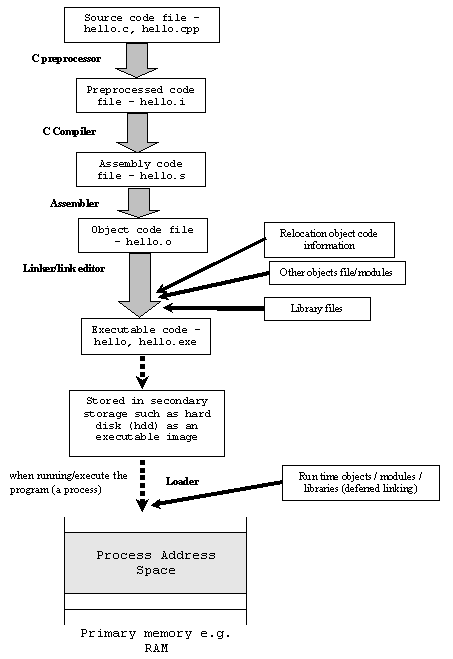
* Text segment 🡪 executable code i.e. instructions + RO data (exmp: char \* str=”HI” then the string HI is stored here).
* Data segment 🡪 initialized segment and uninitialized segment(BSS). All global and static variables are stored here.
* Stack
* Heap

## Various stages of C code compilation

[](https://www.google.ch/url?sa=i&rct=j&q=&esrc=s&source=images&cd=&cad=rja&uact=8&ved=0ahUKEwjQ_JqDiuTNAhWIORoKHQAEAVMQjRwIBw&url=https://www3.ntu.edu.sg/home/ehchua/programming/cpp/gcc_make.html&psig=AFQjCNEVq4jiXhwDQACQzNmIZDxkp7xxhg&ust=1468074493242003)

[](https://www.google.ch/url?sa=i&rct=j&q=&esrc=s&source=images&cd=&cad=rja&uact=8&ved=0ahUKEwjhxPaTiuTNAhWCvRoKHcwYBS0QjRwIBw&url=https://en.wikibooks.org/wiki/Introduction_to_Programming_Languages/Compiled_Programs&psig=AFQjCNEVq4jiXhwDQACQzNmIZDxkp7xxhg&ust=1468074493242003)

[](http://www.google.ch/url?sa=i&rct=j&q=&esrc=s&source=images&cd=&cad=rja&uact=8&ved=0ahUKEwjkj_jTjOTNAhWNhRoKHUkrD-sQjRwIBw&url=http://www.design-reuse.com/articles/37754/building-process-for-the-c-c-program-on-complex-socs.html&psig=AFQjCNEVq4jiXhwDQACQzNmIZDxkp7xxhg&ust=1468074493242003)



## Accessing command line arguments without using argv,argc

# ARRAYS

Int arr[];

Arr = p;=🡺INVALID

|  |
| --- |
| **Arr[i]= \*(arr+i) = \*(i+arr) = i[arr]**  **Arr[i][j] = \*(\*(arr+i)+j) = j[i[arr]]**  **Arr[j][i] = \*(\*(arr+j)+i) = i[j[arr]]** |
|  |
| **For a 1D array, int arr1D[]={1,2,3}**  **arr1D , &arr1D, &arr1D[0] all have the same value, BUT**  **arr1D , &arr1D are of type pointer to an array i.e.int (\*)[3] where as &arr1D[0] is of type int \*** |
|  |

1. Variable sized array

#define M 2

Int main()

{

Int arr[M][M]={1,2,3,4};

}

**WORKS**

|  |  |
| --- | --- |
| int M = 2;    int arr[M][M];    int i, j;    for (i = 0; i < M; i++)    {      for (j = 0; j < M; j++)      {         arr[i][j] = 0;         printf ("%d ", arr[i][j]);      }  **WORKS** | int M = 2;    int arr[M][M] = {0}; // Trying to initialize all values as 0    int i, j;    for (i = 0; i < M; i++)    {      for (j = 0; j < M; j++)         printf ("%d ", arr[i][j]);      printf("\n");    }  **Compiler Error: variable-sized object may not be initialized** |
| #define M 2    int arr[M][M];    int i, j;    for (i = 0; i < M; i++)    {      for (j = 0; j < M; j++)      {         arr[i][j] = 0;         printf ("%d ", arr[i][j]);      }  **WORKS** | Int a=1,b=2,c=3;  Int arr[3]={a,b,c};  **WORKS** |

1. Array is a pointer constant.

Int arr[4]={1,2,3,4};

Int p[4];

P = arr; 🡺 Compile time ERROR

# POINTERS

1. Void pointer can’t be dereferenced directly :

Exmp : #include<stdio.h>

int main()

{

    int a = 10;

    void \*ptr = &a;

    printf("%d", \*ptr); ERROR

    printf("%d", \*(int\*)ptr); CORRECT

  printf("%d", (int)\*ptr); ERROR 🡺 void pointer has to be type-casted before dereferencing it

int \*p2=&5;

printf("\n So \*p2 is %d",\*p2); COMPILATION ERROR 🡪 lvalue required as unary & operator

    return 0;

}

1. NULL pointer is nothing but (void\*)0
2. Size of (void\*) 🡺8, size of (int\*) 🡺8bytes
3. Example :

Void foo(float\*);

Int main90

{

Int i=10, \*p=&I;

Foo(&i);

}

Void foo(float \*p)

{

Printf(“\n %f”,\*p);

}

OUTPUT : 0.000000

Function returning pointer to a function

void (\*signal (int signo, void(\*sighandler)(int)))(int)

|  |
| --- |
| **If the function return type is void\* then**  **Void \* dummyfunc(void)**  **{**  **Return (void \*)0;**  **}** |

# LINKED LIST

## Given only a pointer/reference to a node to be deleted in a singly linked list, how do you delete it?

## Delete a node

/\* Given a reference (pointer to pointer) to the head of a list

and a key, deletes the first occurrence of key in linked list \*/

void deleteNode(struct node \*\*head\_ref, int key)

{

// Store head node

struct node\* temp = \*head\_ref, \*prev;

// If head node itself holds the key to be deleted

if (temp != NULL && temp->data == key)

{

\*head\_ref = temp->next; // Changed head

free(temp); // free old head

return;

}

// Search for the key to be deleted, keep track of the

// previous node as we need to change 'prev->next'

while (temp != NULL && temp->data != key)

{

prev = temp;

temp = temp->next;

}

// If key was not present in linked list

if (temp == NULL) return;

// Unlink the node from linked list

prev->next = temp->next;

free(temp); // Free memory

}

## Deleting a key at a given position

/\* Given a reference (pointer to pointer) to the head of a list

   and a position, deletes the node at the given position \*/

void deleteNode(struct node \*\*head\_ref, int position)

{

   // If linked list is empty

   if (\*head\_ref == NULL)

      return;

   // Store head node

   struct node\* temp = \*head\_ref;

    // If head needs to be removed

    if (position == 0)

    {

        \*head\_ref = temp->next;   // Change head

        free(temp);               // free old head

        return;

    }

    // Find previous node of the node to be deleted

    for (int i=0; temp!=NULL && i<position-1; i++)

         temp = temp->next;

    // If position is more than number of ndoes

    if (temp == NULL || temp->next == NULL)

         return;

    // Node temp->next is the node to be deleted

    // Store pointer to the next of node to be deleted

    struct node \*next = temp->next->next;

    // Unlink the node from linked list

    free(temp->next);  // Free memory

    temp->next = next;  // Unlink the deleted node from list

}

## Length of a linked list – RECURSIVE

/\* Counts the no. of occurences of a node

(search\_for) in a linked list (head)\*/

int getCount(struct node\* head)

{

// Base case

if (head == NULL)

return 0;

// count is 1 + count of remaining list

return 1 + getCount(head->next);

}

## Search an element in a linked list – RECURSIVE

/\* Counts no. of nodes in linked list \*/

bool search(struct node\* head, int x)

{

    // Base case

    if (head == NULL)

        return false;

    // If key is present in current node, return true

    if (head->key == x)

        return true;

    // Recur for remaining list

    return search(head->next, x);

}

## Swap nodes without swapping data

/\* Function to swap nodes x and y in linked list by

changing links \*/

void swapNodes(struct node \*\*head\_ref, int x, int y)

{

// Nothing to do if x and y are same

if (x == y) return;

// Search for x (keep track of prevX and CurrX

struct node \*prevX = NULL, \*currX = \*head\_ref;

while ((currX && currX->data != x)||(currY && currY->data != y))

{

if(currX && currX->data != x)

{

prevX = currX;

currX = currX->next;

}

if(currY && currY->data != y)

{

prevY = currY;

currY = currY->next;

}

}

// If either x or y is not present, nothing to do

if (currX == NULL || currY == NULL)

return;

// If x is not head of linked list

if (prevX != NULL)

prevX->next = currY;

else // Else make y as new head

\*head\_ref = currY;

// If y is not head of linked list

if (prevY != NULL)

prevY->next = currX;

else // Else make x as new head

\*head\_ref = currX;

// Swap next pointers

struct node \*temp = currY->next;

currY->next = currX->next;

currX->next = temp;

}

## Print middle element of a list

/\* Function to get the middle of the linked list\*/

void printMiddle(struct node \*head)

{

    struct node \*slow\_ptr = head;

    struct node \*fast\_ptr = head;

    if (head!=NULL)

    {

        while (fast\_ptr != NULL && fast\_ptr->next != NULL)

        {

            fast\_ptr = fast\_ptr->next->next;

            slow\_ptr = slow\_ptr->next;

        }

        printf("The middle element is [%d]\n\n", slow\_ptr->data);

    }

}

## Nth node from end of Linked List

/\* Function to get the nth node from the last of a linked list\*/

void printNthFromLast(struct node \*head, int n)

{

  struct node \*main\_ptr = head;

  struct node \*ref\_ptr = head;

  int count = 0;

  if(head != NULL)

  {

     while( count < n )

     {

        if(ref\_ptr == NULL)

        {

           printf("%d is greater than the no. of "

                    "nodes in list", n);

           return;

        }

        ref\_ptr = ref\_ptr->next;

        count++;

     } /\* End of while\*/

     while(ref\_ptr != NULL)

     {

        main\_ptr = main\_ptr->next;

        ref\_ptr  = ref\_ptr->next;

     }

     printf("Node no. %d from last is %d ",

              n, main\_ptr->data);

  }

}

## Delete an entire linked list

/\* Function to delete the entire linked list \*/

void deleteList(struct node\*\* head\_ref)

{

   /\* deref head\_ref to get the real head \*/

   struct node\* current = \*head\_ref;

   struct node\* next;

   while (current != NULL)

   {

       next = current->next;

       free(current);

       current = next;

   }

   /\* deref head\_ref to affect the real head back

      in the caller. \*/

   \*head\_ref = NULL;

}

## Detect loop in a linked list

int detectloop(struct node \*list)

{

struct node \*slow\_p = list, \*fast\_p = list;

while (slow\_p && fast\_p && fast\_p->next )

{

slow\_p = slow\_p->next;

fast\_p = fast\_p->next->next;

if (slow\_p == fast\_p)

{

printf("Found Loop");

return 1;

}

}

return 0;

}

## Reverse a linked list

## Generic linked list

/\* Function to add a node at the beginning of Linked List.

   This function expects a pointer to the data to be added

   and size of the data type \*/

void push(struct node\*\* head\_ref, void \*new\_data, size\_t data\_size)

{

    // Allocate memory for node

    struct node\* new\_node = (struct node\*)malloc(sizeof(struct node));

    new\_node->data  = malloc(data\_size);

    new\_node->next = (\*head\_ref);

    // Copy contents of new\_data to newly allocated memory.

    // Assumption: char takes 1 byte.

    int i;

    for (i=0; i<data\_size; i++)

        \*(char \*)(new\_node->data + i) = \*(char \*)(new\_data + i);

    // Change head pointer as new node is added at the beginning

    (\*head\_ref)    = new\_node;

}

/\* Function to print nodes in a given linked list. fpitr is used

   to access the function to be used for printing current node data.

   Note that different data types need different specifier in printf() \*/

void printList(struct node \*node, void (\*fptr)(void \*))

{

    while (node != NULL)

    {

        (\*fptr)(node->data);

        node = node->next;

    }

}

// Function to print an integer

void printInt(void \*n)

{

   printf(" %d", \*(int \*)n);

}

// Function to print a float

void printFloat(void \*f)

{

   printf(" %f", \*(float \*)f);

}

/\* Driver program to test above function \*/

int main()

{

    struct node \*start = NULL;

    // Create and print an int linked list

    unsigned int\_size = sizeof(int);

    int arr[] = {10, 20, 30, 40, 50}, i;

    for (i=4; i>=0; i--)

       push(&start, &arr[i], int\_size);

    printf("Created integer linked list is \n");

    printList(start, printInt);

    // Create and print a float linked list

    unsigned float\_size = sizeof(float);

    start = NULL;

    float arr2[] = {10.1, 20.2, 30.3, 40.4, 50.5};

    for (i=4; i>=0; i--)

       push(&start, &arr2[i], float\_size);

    printf("\n\nCreated float linked list is \n");

    printList(start, printFloat);

    return 0;

}

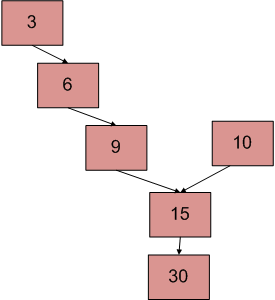
## Function to check if a single linked list is palindrome

This method takes O(n) time and O(1) extra space.  
**1)** Get the middle of the linked list.  
**2)** Reverse the second half of the linked list.  
**3)** Check if the first half and second half are identical.  
**4)** Construct the original linked list by reversing the second half again and attaching it back to the first half

When number of nodes are even, the first and second half contain exactly half nodes. The challenging thing in this method is to handle the case when number of nodes are odd. We don’t want the middle node as part of any of the lists as we are going to compare them for equality. For odd case, we use a separate variable ‘midnode’.

|  |
| --- |
| /\* Program to check if a linked list is palindrome \*/  #include<stdio.h>  #include<stdlib.h>  #include<stdbool.h>    /\* Link list node \*/  struct node  {      char data;      struct node\* next;  };    void reverse(struct node\*\*);  bool compareLists(struct node\*, struct node \*);    /\* Function to check if given linked list is    palindrome or not \*/  bool isPalindrome(struct node \*head)  {      struct node \*slow\_ptr = head, \*fast\_ptr = head;      struct node \*second\_half, \*prev\_of\_slow\_ptr = head;      struct node \*midnode = NULL;  // To handle odd size list      bool res = true; // initialize result        if (head!=NULL && head->next!=NULL)      {          /\* Get the middle of the list. Move slow\_ptr by 1            and fast\_ptrr by 2, slow\_ptr will have the middle            node \*/          while (fast\_ptr != NULL && fast\_ptr->next != NULL)          {              fast\_ptr = fast\_ptr->next->next;                /\*We need previous of the slow\_ptr for               linked lists  with odd elements \*/              prev\_of\_slow\_ptr = slow\_ptr;              slow\_ptr = slow\_ptr->next;          }              /\* fast\_ptr would become NULL when there are even elements in list.             And not NULL for odd elements. We need to skip the middle node             for odd case and store it somewhere so that we can restore the             original list\*/          if (fast\_ptr != NULL)          {              midnode = slow\_ptr;              slow\_ptr = slow\_ptr->next;          }            // Now reverse the second half and compare it with first half          second\_half = slow\_ptr;          prev\_of\_slow\_ptr->next = NULL; // NULL terminate first half          reverse(&second\_half);  // Reverse the second half          res = compareLists(head, second\_half); // compare            /\* Construct the original list back \*/           reverse(&second\_half); // Reverse the second half again           if (midnode != NULL)  // If there was a mid node (odd size case) which                                 // was not part of either first half or second half.           {              prev\_of\_slow\_ptr->next = midnode;              midnode->next = second\_half;           }           else  prev\_of\_slow\_ptr->next = second\_half;      }      return res;  }    /\* Function to reverse the linked list  Note that this      function may change the head \*/  void reverse(struct node\*\* head\_ref)  {      struct node\* prev   = NULL;      struct node\* current = \*head\_ref;      struct node\* next;      while (current != NULL)      {          next  = current->next;          current->next = prev;          prev = current;          current = next;      }      \*head\_ref = prev;  }    /\* Function to check if two input lists have same data\*/  bool compareLists(struct node\* head1, struct node \*head2)  {      struct node\* temp1 = head1;      struct node\* temp2 = head2;        while (temp1 && temp2)      {          if (temp1->data == temp2->data)          {              temp1 = temp1->next;              temp2 = temp2->next;          }          else return 0;      }        /\* Both are empty reurn 1\*/      if (temp1 == NULL && temp2 == NULL)          return 1;        /\* Will reach here when one is NULL        and other is not \*/      return 0;  } |

## Intersection point of two linked lists



* Get count of the nodes in first list, let count be c1.  
  2) Get count of the nodes in second list, let count be c2.  
  3) Get the difference of counts d = abs(c1 – c2)  
  4) Now traverse the bigger list from the first node till d nodes so that from here onwards both the lists have equal no of nodes.  
  5) Then we can traverse both the lists in parallel till we come across a common node. (Note that getting a common node is done by comparing the address of the nodes)

/\* Function to get the counts of node in a linked list \*/

int getCount(struct node\* head);

/\* function to get the intersection point of two linked

   lists head1 and head2 where head1 has d more nodes than

   head2 \*/

int \_getIntesectionNode(int d, struct node\* head1, struct node\* head2);

/\* function to get the intersection point of two linked

   lists head1 and head2 \*/

int getIntesectionNode(struct node\* head1, struct node\* head2)

{

  int c1 = getCount(head1);

  int c2 = getCount(head2);

  int d;

  if(c1 > c2)

  {

    d = c1 - c2;

    return \_getIntesectionNode(d, head1, head2);

  }

  else

  {

    d = c2 - c1;

    return \_getIntesectionNode(d, head2, head1);

  }

}

/\* function to get the intersection point of two linked

   lists head1 and head2 where head1 has d more nodes than

   head2 \*/

int \_getIntesectionNode(int d, struct node\* head1, struct node\* head2)

{

  int i;

  struct node\* current1 = head1;

  struct node\* current2 = head2;

  for(i = 0; i < d; i++)

  {

    if(current1 == NULL)

    {  return -1; }

    current1 = current1->next;

  }

  while(current1 !=  NULL && current2 != NULL)

  {

    if(current1 == current2)

      return current1->data;

    current1= current1->next;

    current2= current2->next;

  }

  return -1;

}

/\* Takes head pointer of the linked list and

   returns the count of nodes in the list \*/

int getCount(struct node\* head)

{

  struct node\* current = head;

  int count = 0;

  while (current != NULL)

  {

    count++;

    current = current->next;

  }

  return count;

}

## Recursive function to print reverse of a Linked list

/\* Function to reverse the linked list \*/

void printReverse(struct node\* head)

{

    // Base case

    if (head == NULL)

       return;

    // print the list after head node

    printReverse(head->next);

    // After everything else is printed, print head

    printf("%d  ", head->data);

}

## Remove duplicates from a sorted LL

Traverse the list from the head (or start) node. While traversing, compare each node with its next node. If data of next node is same as current node then delete the next node. Before we delete a node, we need to store next pointer of the node

/\* The function removes duplicates from a sorted list \*/

void removeDuplicates(struct node\* head)

{

    /\* Pointer to traverse the linked list \*/

    struct node\* current = head;

    /\* Pointer to store the next pointer of a node to be deleted\*/

    struct node\* next\_next;

    /\* do nothing if the list is empty \*/

    if (current == NULL)

       return;

    /\* Traverse the list till last node \*/

    while (current->next != NULL)

    {

       /\* Compare current node with next node \*/

       if (current->data == current->next->data)

       {

           /\* The sequence of steps is important\*/

           next\_next = current->next->next;

           free(current->next);

           current->next = next\_next;

       }

       else /\* This is tricky: only advance if no deletion \*/

       {

          current = current->next;

       }

    }

}

## Remove duplicates from a unsorted LL

TWO loops are used. Outer loop is used to pick the elements one by one and inner loop compares the picked element with rest of the elements.

/\* Function to remove duplicates from a unsorted linked list \*/

void removeDuplicates(struct node \*start)

{

  struct node \*ptr1, \*ptr2, \*dup;

  ptr1 = start;

  /\* Pick elements one by one \*/

  while(ptr1 != NULL && ptr1->next != NULL)

  {

     ptr2 = ptr1;

     /\* Compare the picked element with rest of the elements \*/

     while(ptr2->next != NULL)

     {

       /\* If duplicate then delete it \*/

       if(ptr1->data == ptr2->next->data)

       {

          /\* sequence of steps is important here \*/

          dup = ptr2->next;

          ptr2->next = ptr2->next->next;

          free(dup);

       }

       else /\* This is tricky \*/

       {

          ptr2 = ptr2->next;

       }

     }

     ptr1 = ptr1->next;

  }

}

## Pairwise swap elements of a linked list

## Merge two sorted Linked list

## Insert a node in a sorted linked list

1) If Linked list is empty then make the node as head and return it.

2) If value of the node to be inserted is smaller than value of head node

then insert the node at start and make it head.

3) In a loop, find the appropriate node after which the input node (let 9) is

to be inserted. To find the appropriate node start from head, keep moving

until you reach a node GN (10 in the below diagram) who's value is

greater than the input node. The node just before GN is the appropriate

node (7).

4) Insert the node (9) after the appropriate node (7) found in step 3.

/\* function to insert a new\_node in a list. Note that this

  function expects a pointer to head\_ref as this can modify the

  head of the input linked list (similar to push())\*/

void sortedInsert(struct node\*\* head\_ref, struct node\* new\_node)

{

    struct node\* current;

    /\* Special case for the head end \*/

    if (\*head\_ref == NULL || (\*head\_ref)->data >= new\_node->data)

    {

        new\_node->next = \*head\_ref;

        \*head\_ref = new\_node;

    }

    else

    {

        /\* Locate the node before the point of insertion \*/

        current = \*head\_ref;

        while (current->next!=NULL &&

               current->next->data < new\_node->data)

        {

            current = current->next;

        }

        new\_node->next = current->next;

        current->next = new\_node;

    }

}

# C logics

## Macro to concatenate a string

#define str\_cat(x,y) name\_##x\_##y

## Set, clear and check a bit

|  |
| --- |
| int set\_bit(int val, int bit)  {  return (val |= (1<<bit));  }  int clr\_bit(int val, int bit)  {  return(val &= ~(1<<bit));  }  int isbitset (int val, int bit)  {  return( (val &= (1<<bit)) !=0) ;  }  ALSO  #define set\_bit(val,bit) (val | (1<<bit))  #define clr\_bit(val,bit) (val & (~(1<<bit)) )  #define isbitset(val,bit) ((val & (1<<bit))!=0) RETRUNS 0 IF BIT IS NOT SET AND 1 IF BIT IS SET |

# C programs

## 6.1 Pre-processor’s

|  |
| --- |
| **#include <stdio.h>**  **#**  **#**  **int main()**  **{**  **printf("\n hello \n");**  **return 0;**  **}**  **OUTPUT** : hello |

|  |
| --- |
| **#include <stdio.h>**  **#define A 10**  **int f1()**  **{**  **printf("\n in f1 %d %d %d",A,B,C);**  **}**  **#define B 20**  **int f2()**  **{**  **printf("\n in f2 %d %d %d",A,B,C);**  **}**  **#define C 30**  **int f3()**  **{**  **printf("\n in f3 %d %d %d",A,B,C);**  **}**  **int main()**  **{**  **printf("\n in main %d %d %d",A,B,C);**  **f1();**  **f2();**  **f3();**  **return 0;**  **}**  OUTPUT :  prog.c: In function 'f1':  prog.c:6:34: error: 'B' undeclared (first use in this function)  printf("\n in f1 %d %d %d",A,B,C);  ^  prog.c:6:34: note: each undeclared identifier is reported only once for each function it appears in  prog.c:6:36: error: 'C' undeclared (first use in this function)  printf("\n in f1 %d %d %d",A,B,C);  ^  prog.c: In function 'f2':  prog.c:12:36: error: 'C' undeclared (first use in this function)  printf("\n in f2 %d %d %d",A,B,C);  ^ |

|  |
| --- |
| **#include <stdio.h>**  **#define C 20**  **#define B A**  **int f1()**  **{**  **printf("\n in f1 C %d \n",C);**    **}**  **#define A 200**  **int f2()**  **{**  **printf("\n in f2 %d %d %d \n",A,B,C);**    **}**  **int main()**  **{**  **f1();**  **f2();**  **printf("\n in f2 %d %d %d \n",A,B,C);**    **}**  **OUTPUT :**  in f1 C 20  in f2 200 200 20  in f2 200 200 20  **#define’s => preprocessors, so they are pre-processed line by line , so just before any #define is being used it has to be available thts it…** |

|  |
| --- |
| **#include <stdio.h>**  **#define A B**  **#define B 100**  **#define C 200**  **#define D 300**  **int main()**  **{**  **printf("\n so %d \n",A);**    **#define A C**  **printf("\n so %d \n",A);**    **#define A D**  **printf("\n so %d \n",A);**  **}**  prog.c: In function 'main':  prog.c:13:0: warning: "A" redefined  #define A C  ^  prog.c:3:0: note: this is the location of the previous definition  #define A B  ^  prog.c:16:0: warning: "A" redefined  #define A D  ^  prog.c:13:0: note: this is the location of the previous definition  #define A C  **OUTPUT :**  so 100  so 200  so 300 |

|  |
| --- |
| **#include<stdio.h>**  **typedef char\* PCHAR;**  **#define pchar char\***  **int main()**  **{**  **PCHAR a,b,c;**  **pchar x,y,z;**  **printf("\n size of a %lu \n", sizeof(a));**  **printf("\n size of b %lu\n", sizeof(b));**  **printf("\n size of c %lu\n", sizeof(c));**  **printf("\n size of x %lu\n", sizeof(x));**  **printf("\n size of y %lu\n", sizeof(y));**  **printf("\n size of z %lu\n", sizeof(z));**  **return 0;**  **}** |

|  |
| --- |
| **typedef char\* PCHAR;**  **#define PCHAR char\***  **int main()**  **{**  **PCHAR a,b,c;**    **printf("sizeof of a %ul\n", sizeof(a));**  **printf("sizeof of b %ul\n", sizeof(b));**  **printf("sizeof of c %ul\n", sizeof(c));**  **return 0;**  **}** Output: sizeof of a 8l  sizeof of b 1l  sizeof of c 1l |
| **#define PCHAR char\***  **typedef char\* PCHAR;**  **int main()**  **{**  **PCHAR a,b,c;**    **printf("sizeof of a %ul\n", sizeof(a));**  **printf("sizeof of b %ul\n", sizeof(b));**  **printf("sizeof of c %ul\n", sizeof(c));**  **return 0;**  **}**  prog.c:1:15: error: expected identifier or '(' before 'char'  #define PCHAR char\*  ^  prog.c:2:15: note: in expansion of macro 'PCHAR'  typedef char\* PCHAR; |

|  |
| --- |
| **typedef PCHAR1 PCHAR;**  **#define PCHAR1 char\***  **int main()**  **{**  **PCHAR a,b,c;**    **printf("sizeof of a %ul\n", sizeof(a));**  **printf("sizeof of b %ul\n", sizeof(b));**  **printf("sizeof of c %ul\n", sizeof(c));**  **return 0;**  **}** |

|  |
| --- |
| **/\*\*/ # /\*\*/ defi\**  **ne FO\**  **O 10\**  **20**  **int main()**  **{**  **printf("\n %d",FOO);**  **return 0;**  **}**  Output:  1020 |

|  |
| --- |
| **#include <stdio.h>**  **int main()**  **{**  **int \*ptr,\*ptr1=0;**  **\*ptr = 50;**  **\*ptr1 = 55;**  **printf("\n %d %d",ptr,ptr1);**    **return 0;**  **}**  **OUTPUT:**  **SIGSEGV , endhukante ikkada address sored in ptr, ptr1 is outside the current C program memory stack, andhuke we get it as SEGMENTATION FAULT.** |

|  |
| --- |
| **#include <stdio.h>**  **#include<string.h>**  **int main() {**  **long double ar[10]={1,2};**  **long double arr[3];**  **memcpy(arr,ar,2);**  **printf("%lg %lg\n",arr[0],arr[1]);**  **return 0;**    **}** |

|  |  |
| --- | --- |
| **#include<stdio.h>**  **#include<string.h>**  **int main(void)**  **{**  **char \*src = NULL;**  **char dst[12] = {0};**  **memcpy(dst,src,12);**  **}** | **#include<stdio.h>**  **#include<string.h>**  **int main(void)**  **{**  **char src[5];**  **char dst[12] = {0};**  **memcpy(dst,src,12);**  **}** |
| **Segmentation fault (core dumped)** | **Worked,but memcpy dint check for the size of “src”** |

|  |
| --- |
| #include <stdio.h>  #include<string.h>  int main() {  char a[]="Hello World";  char b[]="Helo Word";  memmove(&a[2],"hi",2);  printf("%s\n",a);  memcpy(&a[2],"hey",3);  printf("a-> %s\n",a);  printf("%s\n",b);  return 0;  }  **OUTPUT:**  Hehio World  a-&gt; Hehey World  Helo Word |

## Programs

int i = 10;

void \*p = &i;

printf("%d\n", \*(int\*)p);

-----------------------------------------------------------

#include <stdio.h>

void main()

{

int x = 0;

int \*ptr = &5;

printf("%p\n", ptr);

}

Ans : Compile time error

-----------------------------------------------------------

#include <stdio.h>

void foo(int\*);

int main()

{

int i = 10, \*p = &i;

foo(p++);

}

void foo(int \*p){

printf("%d\n", \*p);

}

-----------------------------------------------------------

#include <stdio.h>

int main()

{

int i = 97, \*p = &i;

foo(&p);

printf("%d ", \*p);

return 0;

}

void foo(int \*\*p)

{

int j = 2;

\*p = &j;

printf("%d ", \*\*p);

}

-----------------------------------------------------------

#include <stdio.h>

int main()

{

int i = 97, \*p = &i;

foo(&p);

printf("%d ", \*p);

return 0;

}

void foo(int \*p)

{

int j = 2;

p = &j;

printf("%d ", \*p);

}

-----------------------------------------------------------

#include <stdio.h>

int main()

{

int i = 11;

int \*p = &i;

foo(&p);

printf("%d ", \*p);

}

void foo(int \*const \*p)

{

int j = 10;

\*p = &j;

printf("%d ", \*\*p);

}

## Pointers

String : this is something like a **const** character array

char str1[]=”hello”;

char str2[10];

char \*ptr1=”Hi”;

char \*ptr2;

str2 = str1; //ERROR

ptr2 = ptr1; //VALID

ptr1=”Bye”; //VALID

str1 = “welcome” //ERROR

[http://stackoverflow.com/questions/692564/concept-of-void-pointer-in-c-programming](http://stackoverflow.com/questions/692564/concept-of-void-pointer-in-c-programming#_blank)

[http://cdn.oreillystatic.com/oreilly/booksamplers/9781449344184\_sampler.pdf](http://cdn.oreillystatic.com/oreilly/booksamplers/9781449344184_sampler.pdf#_blank)

**Pointer arithmetic is not possible on pointers of void due to lack of concrete value underneath the pointer and hence the size.**

**void\* p = ...**

**void \*p2 = p + 1; /\* what exactly is the size of void?? \*/**

Pointer arithmetic is about changing pointer values by multiples of the sizeof the pointed-to objects. Again, because void is not a true type, sizeof(void) has no meaning so pointer arithmetic is not valid on void \*.

***A void pointer is known as generic pointer, which can refer to variables of any data type.***

void \*ptr;

int a;

ptr=&a;

ptr++; // This statement is invalid and will result in an error because 'ptr' is a void pointer variable.

## Imp Links

[http://www.geeksforgeeks.org/how-linkers-resolve-multiply-defined-global-symbols/](http://www.geeksforgeeks.org/how-linkers-resolve-multiply-defined-global-symbols/#_blank)

[http://www.geeksforgeeks.org/complicated-declarations-in-c/](http://www.geeksforgeeks.org/complicated-declarations-in-c/#_blank)

http://www.geeksforgeeks.org/evaluation-order-of-operands/

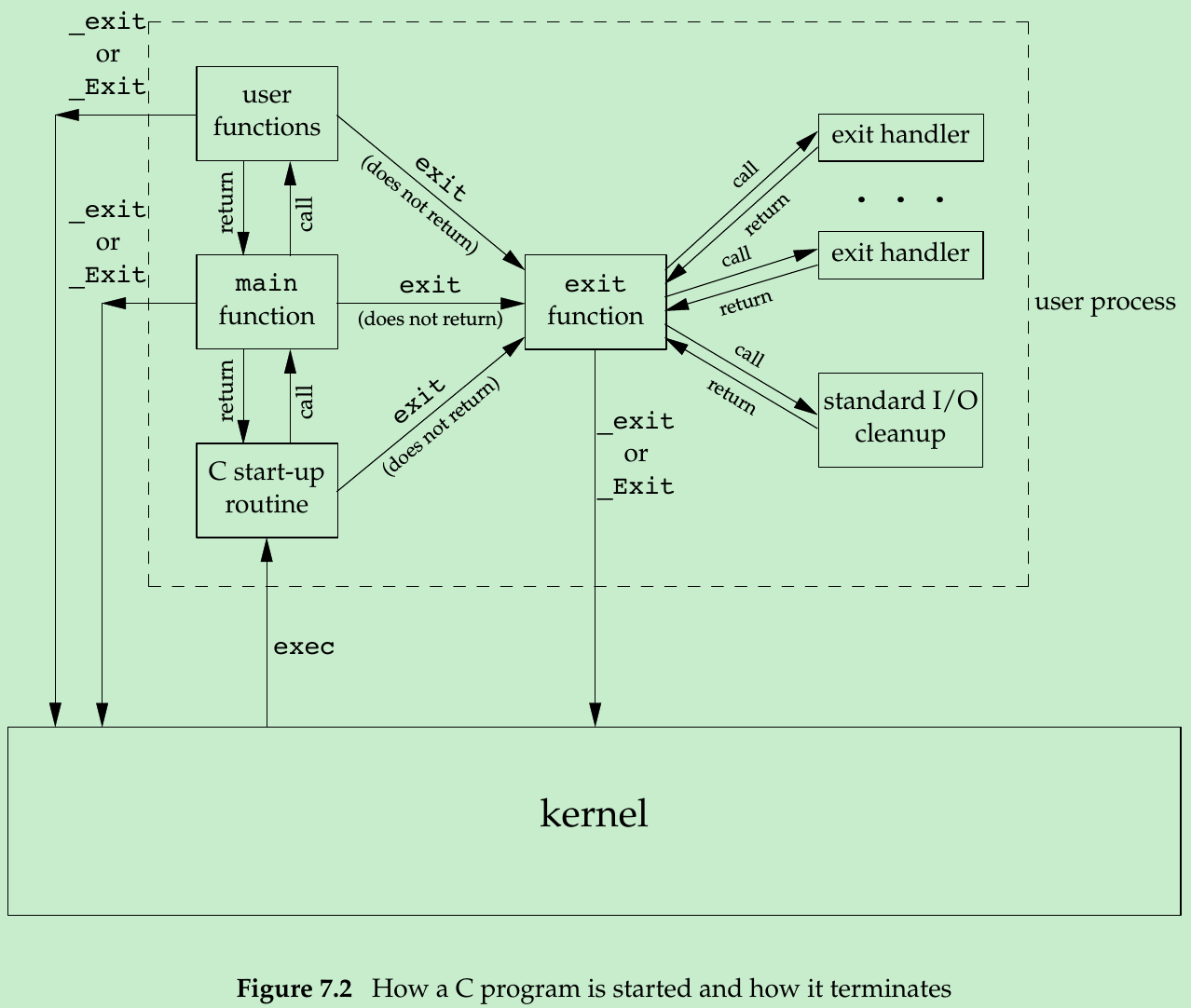
http://www.geeksforgeeks.org/puts-vs-printf-for-printing-a-string/

http://www.codingunit.com/printf-format-specifiers-format-conversions-and-formatted-output

[http://www.codingunit.com/c-tutorial-the-functions-malloc-and-free](http://www.codingunit.com/c-tutorial-the-functions-malloc-and-free#_blank)

## Main() is not the actual start point in C programs

* Return type of main() is always **integer**

****

* **In the above figure : C start up function collects command line arguments and environmental variables and passed them as arguments to main() function**

****

* \_exit and \_Exit() directly return to the kernel , but exit() goes and performs the cleanup routine
* To register an exit function => int atexit(void(\*func)(void));
* Also the above registered exit handler gets invoked only when **exit() OR return 0** is used to return from main function
* If in the main() function \_exit/\_Exit ic alled then the control goes to kernel cleanup function.
* **Historically, most UNIX systems have provided a third argument to the main function that is the address of the environment list => extern char \*\*environ**
* **int main(int argc, char \*argv[], char \*envp[]);**
* 
* But getenv() and putenv(), setenv(), unsetenv(), clearenv() can be used to get/out environmental variables.

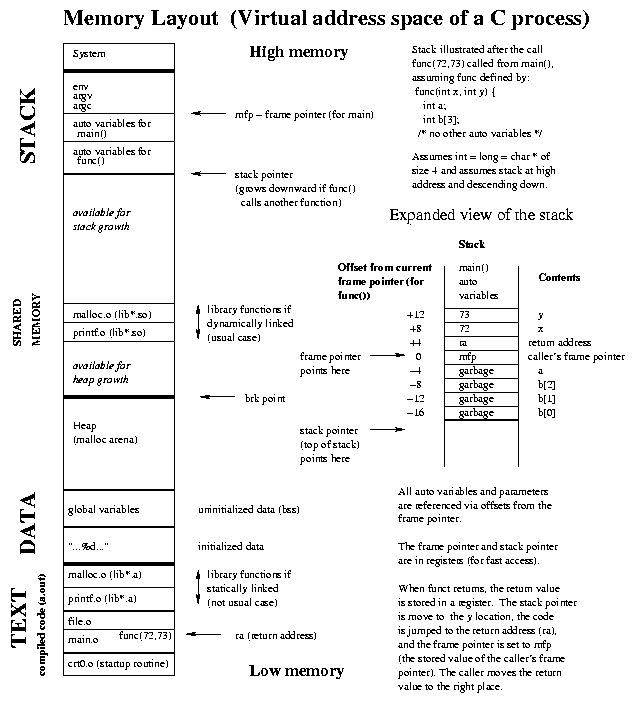




## Memory layout of a C program

The dynamically allocated data (ie. the data created using malloc in C) as well as the static data without a fixed size (such as arrays of variable size) are created and kept in the heap.





* In addition to this Several more segment types exist in an a.out, containing the symbol table, debugging information, linkage tables for dynamic shared libraries, and the like.
* the contents of the uninitialized data segment are not stored in the program file on disk. This is because the kernel sets it to 0 before the program starts running. The only portions of the program that need to be saved in the program file are the text segment and the initialized data..
* size(1) command reports the sizes (in bytes) of the text, data, and bss segments. For example:

$ size /usr/bin/cc /bin/sh

text data bss dec hex filename

79606 1536 916 82058 1408a /usr/bin/cc

619234 21120 18260 658614 a0cb6 /bin/sh

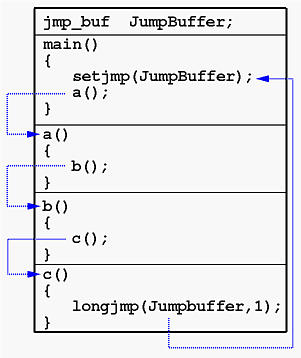
### Shared libraries and malloc/calloc/realloc/free/alloca

* Shared libraries remove the common library routines from the executable file, instead maintaining a single copy of the library routine somewhere in memory that all processes reference. This reduces the size of each executable file but may add some runtime overhead, either when the program is first executed or the first time each shared library function is called. Another advantage of shared libraries is that library functions can be replaced with new versions without having to relink edit very program that uses the library. (This assumes that the number and type of arguments haven't changed.)

|  |
| --- |
| $ cc -static hello1.c prevent gcc from using shared libraries |
| $ ls -l a.out |
| -rwxrwxr-x 1 sar 475570 Feb 18 23:17 a.out |
| $ size a.out |
| text data bss dec hex filename |
| 375657 3780 3220 382657 5d6c1 a.out |
| If we compile this program to use shared libraries, the text and data sizes of the executable file are |
| greatly decreased: |
| $ cc hello1.c gcc defaults to use shared libraries |
| $ ls -l a.out |
| -rwxrwxr-x 1 sar 11410 Feb 18 23:19 a.out |
| $ size a.out |
| text data bss dec hex filename |
| 872 256 4 1132 46c a.out |

* Malloc/calloc/realloc => It is important to realize that most implementations allocate a little more space than is requested and use the additional space for record keepingthe size of the allocated block, a pointer to the next allocated block, and the like. This means that writing past the end of an allocated area could overwrite this record-keeping information in a later block.
* Malloc/calloc/realloc => Writing past the end or before the beginning of a dynamically-allocated buffer can corrupt more than internal record-keeping information. The memory before and after a dynamically-allocated buffer can potentially be used for other dynamically-allocated objects.
* **alloca Function :**One additional function is also worth mentioning. The function alloca has the same calling sequence as malloc; however, instead of allocating memory from the heap, the memory is allocated from the stack frame of the current function. The advantage is that we don't have to free the space; it goes away automatically when the function returns. The alloca function increases the size of the stack frame. The disadvantage is that some systems can't support alloca, if it's impossible to increase the size of the stack frame after the function has been called. Nevertheless, many software packages use it, and implementations exist for a wide variety of systems.

### Setjmp and longjmp



Stack frames without setjmp and longjmp



Stack frames with setjmp and longjmp

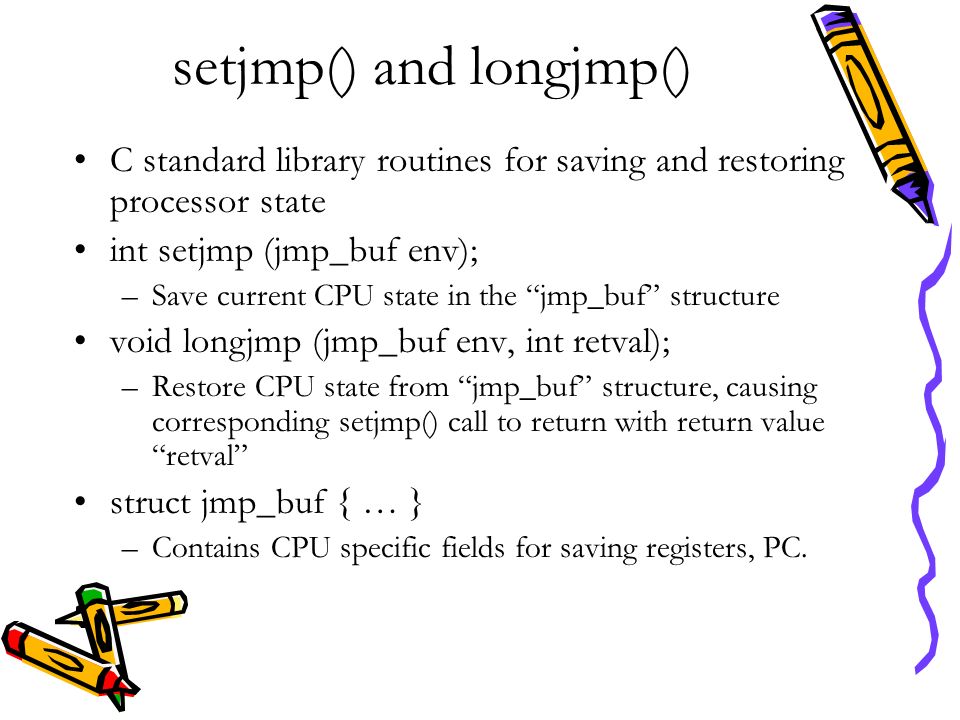


Effect of setjmp/longjmp on variables





**Without optimization all variables will be stored in memory including auto,register variables ,But in case of optimization auto and register are stored in registers and volatile in memory itself.**

****

## Strlen vs sizeof

strlen will calculate the length of a string not including the terminating null byte,

sizeof calculates the size of the buffer, which does include the terminating null byte. Another difference is that using strlen requires a function call, whereas sizeof calculates the buffer length at compile time, as the buffer is initialized with a known string, and its size is fixed.

## 6.5 Malloc

[**http://web.eecs.utk.edu/~huangj/cs360/360/notes/Malloc1/lecture.html**](http://web.eecs.utk.edu/~huangj/cs360/360/notes/Malloc1/lecture.html) **==> VIMP**

**http://web.eecs.utk.edu/~huangj/cs360/360/notes/Malloc2/lecture.html.utk.edu/~huangj/cs360/360/notes/Malloc2/lecture.html ==> VIMP http://web.eecs.utk.edu/~huangj/cs360/360/notes/Malloc2/lecture.html ==> VIMP**

say for example in main(), malloc(20) ==> here 20bytes are allocated and rounded to nearby 8bytes boundary + another 8bytes for malloc-HOUSEKEEPING so intotal around 32bytes.

SO now assume above malloc(20) has been free’ed then again if we do malloc(20) then there are chances the above memory will be allocated.

# DOUBTS

|  |
| --- |
| #include <stdio.h>  void main()  {  int k = m();  printf("%d", k);  }  void m()  {  printf("hello");  } |

|  |
| --- |
| 1. #include <stdio.h> 2. void m(); 3. void n() 4. { 5. m(); 6. } 7. void main() 8. { 9. void m() 10. { 11. printf("hi"); 12. } 13. } |

|  |
| --- |
| int f1(int val)  {  int \*ptr;  if (val == 0) {  int val;  val = 5;  ptr = &val;  printf("\n inside if %d \n",val);  }  printf("\n after if %d \n",val);  return(\*ptr+1); => SEGEMNTATION FAULT since \*ptr (or) \*ptr+1 refers outside the current process stack  } |

# Pragma

#include and #define are pre-processors,

Generally, the #pragma directives are intended for implementing compiler-specific preprocessor instructions. They are not standardized, so you shouldn't rely on them too heavily.

# FILE OPERATIONS

|  |
| --- |
|  |

# OPERATING SYSTEMS

## Links

## Notes

Process switch vs context switch

Thread switching

Does threads also have something like PCB

When a child process is created what does it share from its parent process