Cheatography

# C Exam Cheat Sheet

by prewd6 via cheatography.com/51151/cs/14045/

## recursive functions, parameter passing

```
recursive functions,
parameter passing
/*
* Recursive descent parser
for simple C expressions.
* Very little error
checking.
*/
#include <stdio.h>
#include <stdlib.h>
int expr(void);
int mul_exp(void);
int unary_exp(void);
int primary(void);
main(){
      int val;
      for(;;){
              printf("expr
ession: ");
              val =
expr();
              if(getchar()
!= '\n'){
                      prin
tf("error\n");
                      whil
e(getchar() != '\n')
                          -
     ; / NULL /
              } else{
                      prin
tf("result is %d\n", val);
              }
      }
      exit(EXIT_SUCCESS);
}
int
```

## recursive functions, parameter passing (cont)

```
expr(void){
      int val, ch_in;
      val = mul_exp();
      for(;;){
              switch(ch_in
= getchar()){
              default:
                      unge
tc(ch_in,stdin);
                      retu
rn(val);
              case '+':
                      val
= val + mul_exp();
                      brea
k;
              case '-':
                      val
= val - mul_exp();
                      brea
k;
              }
      }
}
int
mul_exp(void){
      int val, ch_in;
      val = unary_exp();
      for(;;){
              switch(ch_in
= getchar()){
              default:
                      unge
tc(ch_in, stdin);
                      retu
rn(val);
              case '*':
                      val
= val * unary_exp();
                      brea
k;
              case '/':
```

## recursive functions, parameter passing (cont)

```
                      val
= val / unary_exp();
                      brea
k;
              case '%':
                      val
= val % unary_exp();
                      brea
k;
              }
      }
}
int
unary_exp(void){
      int val, ch_in;
      switch(ch_in =
getchar()){
      default:
              ungetc(ch_in
, stdin);
              val =
primary();
              break;
      case '+':
              val =
unary_exp();
              break;
      case '-':
              val = -
unary_exp();
              break;
      }
      return(val);
}
int
primary(void){
      int val, ch_in;
      ch_in = getchar();
      if(ch_in >= '0' &&
ch_in <= '9'){
```

## recursive functions, parameter passing (cont)

```
              val = ch_in
- '0';
              goto out;
      }
      if(ch_in == '('){
              val =
expr();
              getchar(); /
skip closing ')' /
              goto out;
      }
      printf("error:
primary read %d\n",
ch_in);
      exit(EXIT_FAILURE);
out:
      return(val);
}
```

## malloc() and free()

Malloc() is used to allocate a certain amount of memory during the execution of a program. It requests a block of memory from the heap, if the request is granted the operating system will reserve the amount of memory. When the amount of memory is not needed anymore you must return it to the operating system by calling the free() function.

```
#include<stdio.h>
int main()
```

## malloc() and free() (cont)

```
{
int *ptr_one;
ptr_one = (int
*)malloc(sizeof(int));
if (ptr_one == 0)
{
printf("ERROR: Out of
memory\n");
return 1;
}
*ptr_one = 25;
printf("%d\n", *ptr_one);
free(ptr_one);
return 0;
}
```

The malloc statement will ask of an amount of memory with size of an integer (32 bits or 4 bytes) If there is not enough memory available the malloc function will return a NULL If the request is granted the address of the reserved block will be placed into the pointer variable.

```
#include<stdio.h>
typedef struct rec
{
     int i;
     float PI;
     char A;
}RECORD;
int main()
{
     RECORD *ptr_one;
     ptr_one = (RECORD *)
malloc (sizeof(RECORD));
     (*ptr_one).i = 10;
     (*ptr_one).PI = 3.14;
     (*ptr_one).A = 'a';
     printf("First value:
%d\n",(*ptr_one).i);
     printf("Second value:
%f\n", (*ptr_one).PI);
     printf("Third value:
%c\n", (*ptr_one).A);
     free(ptr_one);
     return 0;
}
```

## multiple inclusion protection

The basic use of header files is to provide symbol declarations for functions and globals. Because multiple declarations of a given symbol in a single translation unit are a syntax error, you have to defensively structure your header files to not redefine anything in case they are included multiple times.

## multiple inclusion protection (cont)

Keep in mind that you just cannot prevent header files from being included more than once unless you were to forbid header files themselves from including other header files... and doing that would be suboptimal at best as we shall see in a future post on self-containment.

Just follow this pattern and encapsulate the whole contents of the whole header file within a guard:

```
#if
!defined(PROJECT_MODULE_H)
#define PROJECT_MODULE_H
... all header file
contents go here ...
#endif /
```
*!defined(PROJECT_MODULE_H)*
/

## multiple inclusion protection (cont)

properly scope the guard names. These names must be unique within your project and within any project that may ever include them. Therefore, it is good practice to always prefix your guard names with the name of your project and follow them by the name of the module.

Compilers expect the structure above in order to apply optimizations against multiple inclusions of a single file. If you break the pattern, you can unknowingly incur higher build times.

The exception

As with any rule there is an exception: not all header files can safely be included more than once. If a header file defines a static symbol or helper function, you have to ensure that it is not pulled in from more than one place.

### multiple inclusion protection (cont)

Yes, the compiler would detect this on its own but, for readability purposes, your header file should explicitly state this fact. Use this other pattern instead:

```
#if
defined(PROJECT_MODULE_H)
#error "Must only be
included once and only
from .c files"
#endif
#define PROJECT_MODULE_H
... all header file
contents go here ...
```

### multiple inclusion protection (cont)

But when can this happen? Very rarely, really. A specific case of the above would be a header file providing helper functions for testing, both their definitions and their implementation. Theoretically, you could split the two into a traditional header file and a source file, compile them separately and link them together with each test program you write. However, doing so may complicate your build unnecessarily.

### comparator functions

Standard C library provides qsort() that can be used for sorting an array. As the name suggests, the function uses QuickSort algorithm to sort the given array. Following is prototype of qsort()

```
void qsort (void* base,
size_t num, size_t size,
            int
(comparator)(const
void,const void*));
```

### comparator functions (cont)

The key point about qsort() is comparator function comparator. The comparator function takes two arguments and contains logic to decide their relative order in sorted output. The idea is to provide flexibility so that qsort() can be used for any type (including user defined types) and can be used to obtain any desired order (increasing, decreasing or any other).

The comparator function takes two pointers as arguments (both type-casted to const void*) and defines the order of the elements by returning (in a stable and transitive manner

```
int comparator(const void
p, const void q)
{
    int l = ((struct
Student *)p)->marks;
    int r = ((struct
Student *)q)->marks;
    return (l - r);
}
```

### comparator functions (cont)

```
// This function is used
in qsort to decide the
relative order
// of elements at
addresses p and q.
int comparator(const void
p, const void q)
{
    // Get the values at
given addresses
    int l = (const int )p;
    int r = (const int )q;

    // both odd, put the
greater of two first.
    if ((l&1) && (r&1))
        return (r-l);

    // both even, put the
smaller of two first
    if ( !(l&1) && !(r&1)
)
        return (l-r);

    // l is even, put r
first
```

## comparator functions (cont)

```
    if (!(1&1))
        return 1;

    // 1 is odd, put 1
first
    return -1;
}

// A utility function to
print an array
void printArr(int arr[],
int n)
{
    int i;
    for (i = 0; i < n;
++i)
        printf("%d ",
arr[i]);
}

// Driver program to test
above function
int main()
{
    int arr[] = {1, 6, 5,
2, 3, 9, 4, 7, 8};

    int size =
sizeof(arr) /
sizeof(arr[0]);
    qsort((void*)arr,
size, sizeof(arr[0]),
comparator);

    printf("Output array
is\n");
    printArr(arr, size);

    return 0;
}
Output:
Output array is
9 7 5 3 1 2 4 6 8
```

## unions

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

union Data {
   int i;
   float f;
   char str[20];
};
int main( ) {
   union Data data;
   printf( "Memory size
occupied by data : %d\n",
sizeof(data));
   return 0;
} // returns: Memory size
occupied by data : 20
```

## unions (cont)

To access any member of a union, we use the member access operator (.). The member access operator is coded as a period between the union variable name and the union member that we wish to access. You would use the keyword union to define variables of union type. The following example shows how to use unions in a program -

```
#include <stdio.h>
#include <string.h>
union Data {
   int i;
   float f;
   char str[20];
};
int main( ) {
   union Data data;
   data.i = 10;
   data.f = 220.5;
   strcpy( data.str, "C
Programming");
   printf( "data.i :
%d\n", data.i);
   printf( "data.f :
%f\n", data.f);
   printf( "data.str :
%s\n", data.str);
   return 0;
}//returns the following:
data.i : 1917853763
//data.f :
41223605803277948604527599
94368.000000
//data.str : C
Programming
```