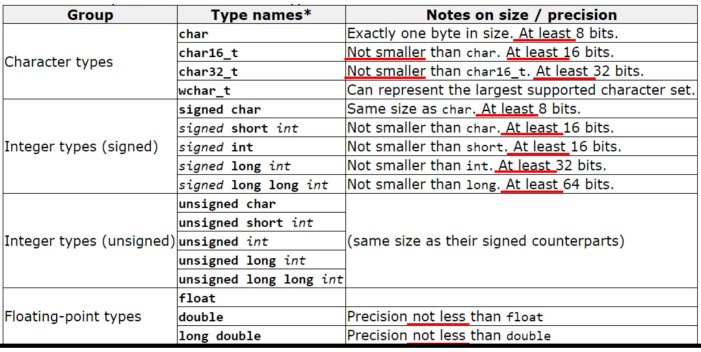
# CPE 2600 MIDTERM

Leigh Goetsch

10/29/2024

## C basic Types



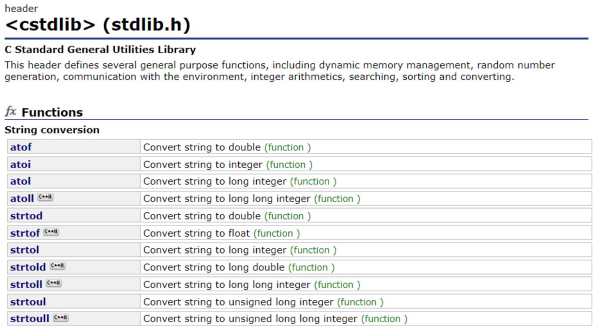
| Group | Type names* | Notes on size / precision |
|---|---|---|
| Character types | `char` | Exactly one byte in size. At least 8 bits. |
| | `char16_t` | Not smaller than `char`. At least 16 bits. |
| | `char32_t` | Not smaller than `char16_t`. At least 32 bits. |
| | `wchar_t` | Can represent the largest supported character set. |
| Integer types (signed) | `signed char` | Same size as `char`. At least 8 bits. |
| | *`signed`* `short` *`int`* | Not smaller than `char`. At least 16 bits. |
| | *`signed`* `int` | Not smaller than `short`. At least 16 bits. |
| | *`signed`* `long` *`int`* | Not smaller than `int`. At least 32 bits. |
| | *`signed`* `long long` *`int`* | Not smaller than `long`. At least 64 bits. |
| Integer types (unsigned) | `unsigned char` | (same size as their signed counterparts) |
| | `unsigned short` *`int`* | |
| | `unsigned` *`int`* | |
| | `unsigned long` *`int`* | |
| | `unsigned long long` *`int`* | |
| Floating-point types | `float` | |
| | `double` | Precision not less than `float` |
| | `long double` | Precision not less than `double` |

## String Conversions

ato functions
we have
discussed

Two conversion
methods:

1. stdlib
conversions
functions
2. stdio
formatted IO
functions
(sprint)



header
### <cstdlib> (stdlib.h)

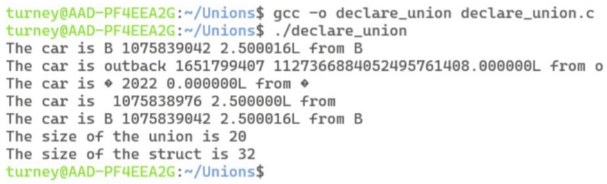
**C Standard General Utilities Library**

This header defines several general purpose functions, including dynamic memory management, random number generation, communication with the environment, integer arithmetics, searching, sorting and converting.

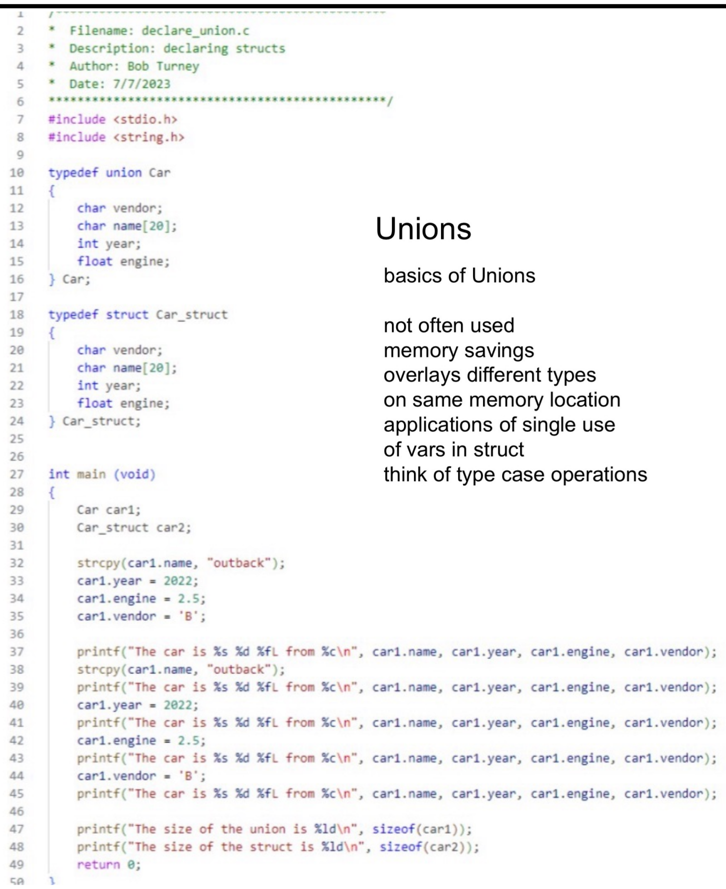
**Functions**

**String conversion**

| | |
|---|---|
| **atof** | Convert string to double (function ) |
| **atoi** | Convert string to integer (function ) |
| **atol** | Convert string to long integer (function ) |
| **atoll** C++11 | Convert string to long long integer (function ) |
| **strtod** | Convert string to double (function ) |
| **strtof** C++11 | Convert string to float (function ) |
| **strtol** | Convert string to long integer (function ) |
| **strtold** C++11 | Convert string to long double (function ) |
| **strtoll** C++11 | Convert string to long long integer (function ) |
| **strtoul** | Convert string to unsigned long integer (function ) |
| **strtoull** C++11 | Convert string to unsigned long long integer (function ) |



```
turney@AAD-PF4EEA2G:~/Unions$ gcc -o declare_union declare_union.c
turney@AAD-PF4EEA2G:~/Unions$ ./declare_union
The car is B 1075839042 2.500016L from B
The car is outback 1651799407 1127366884052495761408.000000L from o
The car is ◆ 2022 0.000000L from ◆
The car is  1075838976 2.500000L from
The car is B 1075839042 2.500016L from B
The size of the union is 20
The size of the struct is 32
turney@AAD-PF4EEA2G:~/Unions$
```



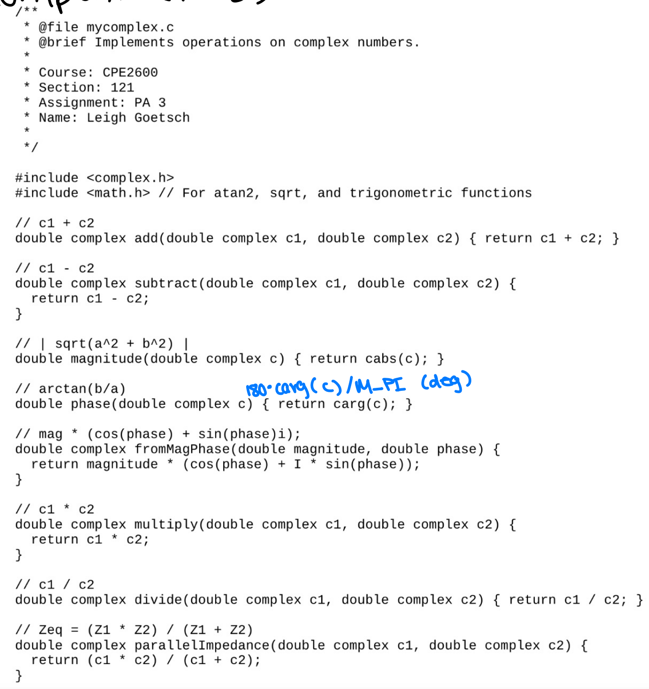
```c
/*----------------------------------------------
 *  Filename: declare_union.c
 *  Description: declaring structs
 *  Author: Bob Turney
 *  Date: 7/7/2023
 **********************************************/
#include <stdio.h>
#include <string.h>

typedef union Car
{
    char vendor;
    char name[20];
    int year;
    float engine;
} Car;

typedef struct Car_struct
{
    char vendor;
    char name[20];
    int year;
    float engine;
} Car_struct;


int main (void)
{
    Car car1;
    Car_struct car2;

    strcpy(car1.name, "outback");
    car1.year = 2022;
    car1.engine = 2.5;
    car1.vendor = 'B';

    printf("The car is %s %d %fL from %c\n", car1.name, car1.year, car1.engine, car1.vendor);
    strcpy(car1.name, "outback");
    printf("The car is %s %d %fL from %c\n", car1.name, car1.year, car1.engine, car1.vendor);
    car1.year = 2022;
    printf("The car is %s %d %fL from %c\n", car1.name, car1.year, car1.engine, car1.vendor);
    car1.engine = 2.5;
    printf("The car is %s %d %fL from %c\n", car1.name, car1.year, car1.engine, car1.vendor);
    car1.vendor = 'B';
    printf("The car is %s %d %fL from %c\n", car1.name, car1.year, car1.engine, car1.vendor);

    printf("The size of the union is %ld\n", sizeof(car1));
    printf("The size of the struct is %ld\n", sizeof(car2));
    return 0;
}
```

## Unions

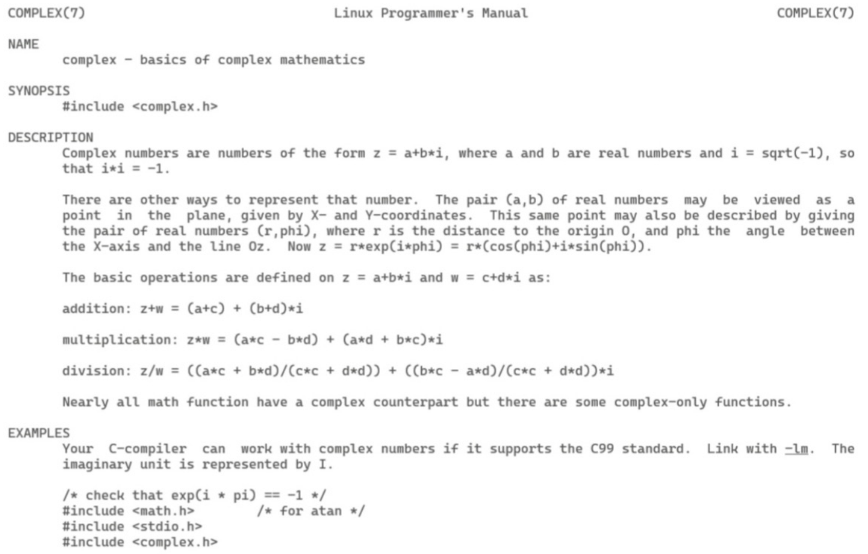
basics of Unions

not often used
memory savings
overlays different types
on same memory location
applications of single use
of vars in struct
think of type case operations

## Complex (PA3)



```c
/**
 * @file mycomplex.c
 * @brief Implements operations on complex numbers.
 *
 * Course: CPE2600
 * Section: 121
 * Assignment: PA 3
 * Name: Leigh Goetsch
 *
 */

#include <complex.h>
#include <math.h> // For atan2, sqrt, and trigonometric functions

// c1 + c2
double complex add(double complex c1, double complex c2) { return c1 + c2; }

// c1 - c2
double complex subtract(double complex c1, double complex c2) {
  return c1 - c2;
}

// | sqrt(a^2 + b^2) |
double magnitude(double complex c) { return cabs(c); }

// arctan(b/a)                          180·carg(c)/M_PI (deg)
double phase(double complex c) { return carg(c); }

// mag * (cos(phase) + sin(phase)i);
double complex fromMagPhase(double magnitude, double phase) {
  return magnitude * (cos(phase) + I * sin(phase));
}

// c1 * c2
double complex multiply(double complex c1, double complex c2) {
  return c1 * c2;
}

// c1 / c2
double complex divide(double complex c1, double complex c2) { return c1 / c2; }

// Zeq = (Z1 * Z2) / (Z1 + Z2)
double complex parallelImpedance(double complex c1, double complex c2) {
  return (c1 * c2) / (c1 + c2);
}
```



```
COMPLEX(7)                    Linux Programmer's Manual                    COMPLEX(7)

NAME
       complex - basics of complex mathematics

SYNOPSIS
       #include <complex.h>

DESCRIPTION
       Complex numbers are numbers of the form z = a+b*i, where a and b are real numbers and i = sqrt(-1), so
       that i*i = -1.

       There are other ways to represent that number.  The pair (a,b) of real numbers  may  be  viewed  as  a
       point  in  the  plane, given by X- and Y-coordinates.  This same point may also be described by giving
       the pair of real numbers (r,phi), where r is the distance to the origin O, and phi the  angle  between
       the X-axis and the line Oz.  Now z = r*exp(i*phi) = r*(cos(phi)+i*sin(phi)).

       The basic operations are defined on z = a+b*i and w = c+d*i as:

       addition: z+w = (a+c) + (b+d)*i

       multiplication: z*w = (a*c - b*d) + (a*d + b*c)*i

       division: z/w = ((a*c + b*d)/(c*c + d*d)) + ((b*c - a*d)/(c*c + d*d))*i

       Nearly all math function have a complex counterpart but there are some complex-only functions.

EXAMPLES
       Your  C-compiler  can  work with complex numbers if it supports the C99 standard.  Link with -lm.  The
       imaginary unit is represented by I.

       /* check that exp(i * pi) == -1 */
       #include <math.h>        /* for atan */
       #include <stdio.h>
       #include <complex.h>
```

## dynamic memory:

Type *ptr = malloc(size·sizeof(Type))
ptr = realloc(ptr2, size·sizeof(Type))
free(ptr)

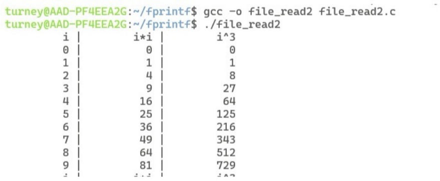
double x;
scanf("%lf", &x);

# CPE 2600
# MIDTERM
Leigh Goetsch

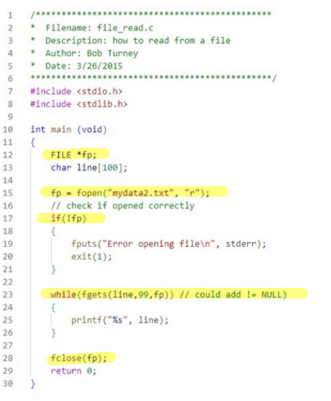
10/29/2024

# File IO

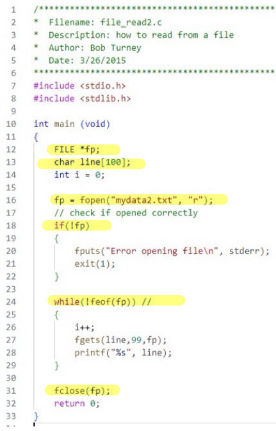
example
reading an entire file
feof()
another way to read until end



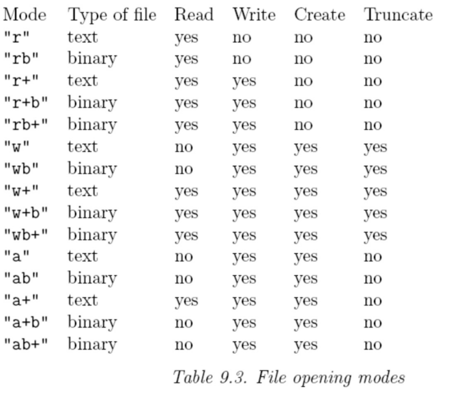
```
turney@AAD-PF4EEA2G:~/fprintf$ gcc -o file_read2 file_read2.c
turney@AAD-PF4EEA2G:~/fprintf$ ./file_read2
     i |        i*i |        i^3
     0 |          0 |          0
     1 |          1 |          1
     2 |          4 |          8
     3 |          9 |         27
     4 |         16 |         64
     5 |         25 |        125
     6 |         36 |        216
     7 |         49 |        343
     8 |         64 |        512
     9 |         81 |        729
```



```c
/***********************************************
*  Filename: file_read.c
*  Description: how to read from a file
*  Author: Bob Turney
*  Date: 3/26/2015
************************************************/
#include <stdio.h>
#include <stdlib.h>

int main (void)
{
    FILE *fp;
    char line[100];

    fp = fopen("mydata2.txt", "r");
    // check if opened correctly
    if(!fp)
    {
        fputs("Error opening file\n", stderr);
        exit(1);
    }

    while(fgets(line,99,fp)) // could add != NULL)
    {
        printf("%s", line);
    }

    fclose(fp);
    return 0;
}
```



```c
/***********************************************
*  Filename: file_read2.c
*  Description: how to read from a file
*  Author: Bob Turney
*  Date: 3/26/2015
************************************************
#include <stdio.h>
#include <stdlib.h>

int main (void)
{
    FILE *fp;
    char line[100];
    int i = 0;

    fp = fopen("mydata2.txt", "r");
    // check if opened correctly
    if(!fp)
    {
        fputs("Error opening file\n", stderr);
        exit(1);
    }

    while(!feof(fp)) //
    {
        i++;
        fgets(line,99,fp);
        printf("%s", line);
    }

    fclose(fp);
    return 0;
}
```

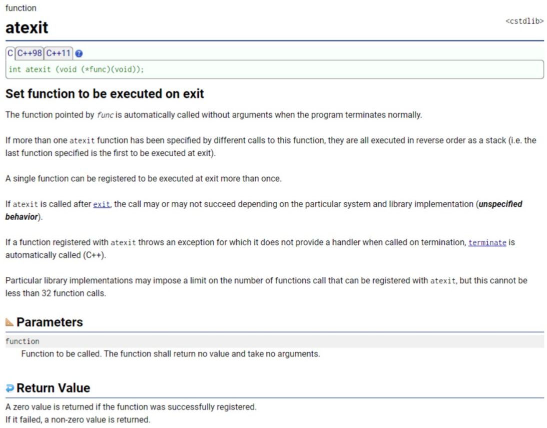


| Mode | Type of file | Read | Write | Create | Truncate |
|---|---|---|---|---|---|
| "r" | text | yes | no | no | no |
| "rb" | binary | yes | no | no | no |
| "r+" | text | yes | yes | no | no |
| "r+b" | binary | yes | yes | no | no |
| "rb+" | binary | yes | yes | no | no |
| "w" | text | no | yes | yes | yes |
| "wb" | binary | no | yes | yes | yes |
| "w+" | text | yes | yes | yes | yes |
| "w+b" | binary | yes | yes | yes | yes |
| "wb+" | binary | yes | yes | yes | yes |
| "a" | text | no | yes | yes | no |
| "ab" | binary | no | yes | yes | no |
| "a+" | text | yes | yes | yes | no |
| "a+b" | binary | no | yes | yes | no |
| "ab+" | binary | no | yes | yes | no |

*Table 9.3. File opening modes*

# Function Pointers

atexit()
uses a function call
as argument



function

## atexit

<cstdlib>

C | C++98 | C++11

```c
int atexit (void (*func)(void));
```

### Set function to be executed on exit

The function pointed by *func* is automatically called without arguments when the program terminates normally.

If more than one `atexit` function has been specified by different calls to this function, they are all executed in reverse order as a stack (i.e. the last function specified is the first to be executed at exit).

A single function can be registered to be executed at exit more than once.

If `atexit` is called after `exit`, the call may or may not succeed depending on the particular system and library implementation (***unspecified behavior***).

If a function registered with `atexit` throws an exception for which it does not provide a handler when called on termination, `terminate` is automatically called (C++).

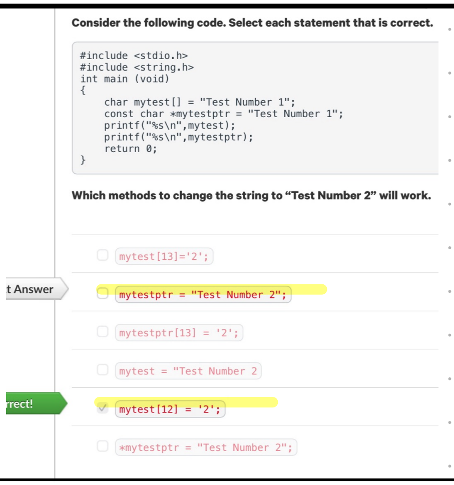
Particular library implementations may impose a limit on the number of functions call that can be registered with `atexit`, but this cannot be less than 32 function calls.

### Parameters

function
Function to be called. The function shall return no value and take no arguments.

### Return Value

A zero value is returned if the function was successfully registered.
If it failed, a non-zero value is returned.

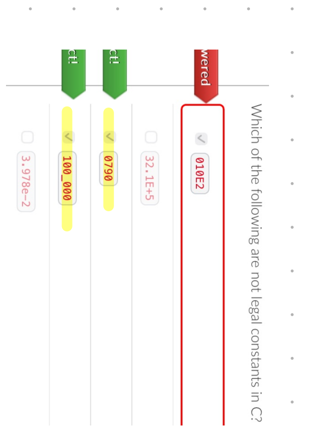


**Consider the following code. Select each statement that is correct.**

```c
#include <stdio.h>
#include <string.h>
int main (void)
{
    char mytest[] = "Test Number 1";
    const char *mytestptr = "Test Number 1";
    printf("%s\n",mytest);
    printf("%s\n",mytestptr);
    return 0;
}
```

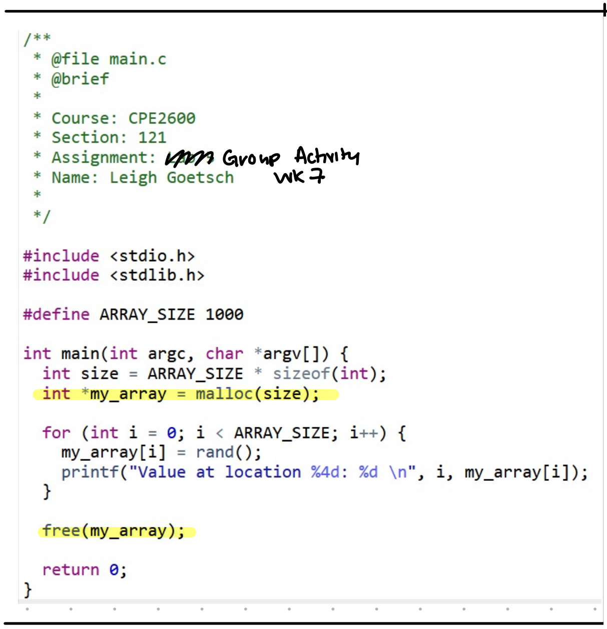
**Which methods to change the string to “Test Number 2” will work.**

- □ `mytest[13]='2';`
- t Answer □ `mytestptr = "Test Number 2";`
- □ `mytestptr[13] = '2';`
- □ `mytest = "Test Number 2`
- rrect! ☑ `mytest[12] = '2';`
- □ `*mytestptr = "Test Number 2";`

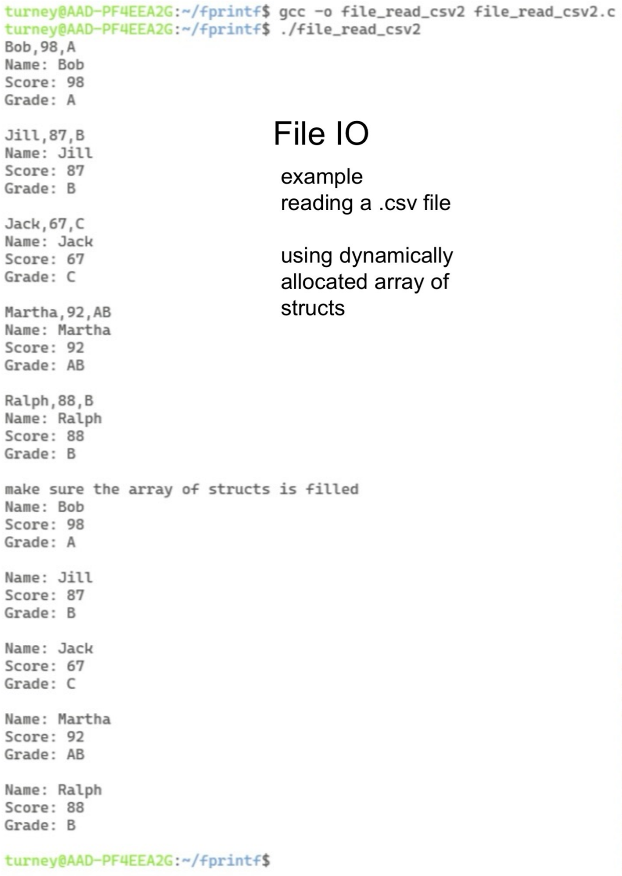


Which of the following are not legal constants in C?

- wered ☑ `010E2`
- □ `32.1E+5`
- ct! ☑ `0790`
- ct! ☑ `100_000`
- □ `3.978e-2`



```c
/**
 * @file main.c
 * @brief
 *
 * Course: CPE2600
 * Section: 121
 * Assignment: Group Activity wk 7
 * Name: Leigh Goetsch
 *
 */

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

#define ARRAY_SIZE 1000

int main(int argc, char *argv[]) {
  int size = ARRAY_SIZE * sizeof(int);
  int *my_array = malloc(size);

  for (int i = 0; i < ARRAY_SIZE; i++) {
    my_array[i] = rand();
    printf("Value at location %4d: %d \n", i, my_array[i]);
  }

  free(my_array);

  return 0;
}
```

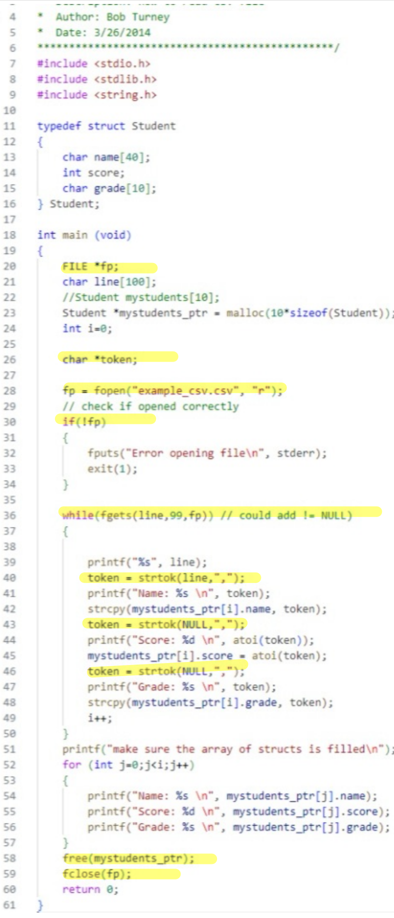


```
turney@AAD-PF4EEA2G:~/fprintf$ gcc -o file_read_csv2 file_read_csv2.c
turney@AAD-PF4EEA2G:~/fprintf$ ./file_read_csv2
Bob,98,A
Name: Bob
Score: 98
Grade: A

Jill,87,B
Name: Jill
Score: 87
Grade: B

Jack,67,C
Name: Jack
Score: 67
Grade: C

Martha,92,AB
Name: Martha
Score: 92
Grade: AB

Ralph,88,B
Name: Ralph
Score: 88
Grade: B

make sure the array of structs is filled
Name: Bob
Score: 98
Grade: A

Name: Jill
Score: 87
Grade: B

Name: Jack
Score: 67
Grade: C

Name: Martha
Score: 92
Grade: AB

Name: Ralph
Score: 88
Grade: B

turney@AAD-PF4EEA2G:~/fprintf$
```

# File IO

example
reading a .csv file

using dynamically
allocated array of
structs



```c
*  Author: Bob Turney
*  Date: 3/26/2014
************************************************/
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

typedef struct Student
{
    char name[40];
    int score;
    char grade[10];
} Student;

int main (void)
{
    FILE *fp;
    char line[100];
    //Student mystudents[10];
    Student *mystudents_ptr = malloc(10*sizeof(Student));
    int i=0;

    char *token;

    fp = fopen("example_csv.csv", "r");
    // check if opened correctly
    if(!fp)
    {
        fputs("Error opening file\n", stderr);
        exit(1);
    }

    while(fgets(line,99,fp)) // could add != NULL)
    {

        printf("%s", line);
        token = strtok(line,",");
        printf("Name: %s \n", token);
        strcpy(mystudents_ptr[i].name, token);
        token = strtok(NULL,",");
        printf("Score: %d \n", atoi(token));
        mystudents_ptr[i].score = atoi(token);
        token = strtok(NULL,",");
        printf("Grade: %s \n", token);
        strcpy(mystudents_ptr[i].grade, token);
        i++;
    }
    printf("make sure the array of structs is filled\n");
    for (int j=0;j<i;j++)
    {
        printf("Name: %s \n", mystudents_ptr[j].name);
        printf("Score: %d \n", mystudents_ptr[j].score);
        printf("Grade: %s \n", mystudents_ptr[j].grade);
    }
    free(mystudents_ptr);
    fclose(fp);
    return 0;
}
```

## C Type modifiers

example
static const unsigned long int x = 4.2;

[storage class] [type qualifier] [type specifiers] [declarator(s)] = [initializer]

storage classes: auto, static, extern, register

type qualifier: const, volatile, restrict

type specifiers: void, char, short, int, long, float, double, signed, unsigned, combinations and user defined types (typedef)

# C Reference Card (ANSI)

## Program Structure/Functions

| | |
|---|---|
| *type fnc*(*type*$_1$, ...); | function prototype |
| *type name*; | variable declaration |
| int main(void) { | main routine |
|   *declarations* | local variable declarations |
|   *statements* | |
| } | |
| *type fnc*(*arg*$_1$, ...) { | function definition |
|   *declarations* | local variable declarations |
|   *statements* | |
|   return *value*; | |
| } | |
| /* */ | comments |
| int main(int argc, char *argv[]) | main with args |
| exit(*arg*); | terminate execution |

## C Preprocessor

| | |
|---|---|
| include library file | #include <*filename*> |
| include user file | #include "*filename*" |
| replacement text | #define *name text* |
| replacement macro | #define *name*(*var*) *text* |
|   *Example.* #define max(A,B) ((A)>(B) ? (A) : (B)) | |
| undefine | #undef *name* |
| quoted string in replace | # |
|   *Example.* #define msg(A) printf("%s = %d", #A, (A)) | |
| concatenate args and rescan | ## |
| conditional execution | #if, #else, #elif, #endif |
| is *name* defined, not defined? | #ifdef, #ifndef |
| *name* defined? | defined(*name*) |
| line continuation char | \ |

## Data Types/Declarations

| | |
|---|---|
| character (1 byte) | char |
| integer | int |
| real number (single, double precision) | float, double |
| short (16 bit integer) | short |
| long (32 bit integer) | long |
| double long (64 bit integer) | long long |
| positive or negative | signed |
| non-negative modulo $2^m$ | unsigned |
| pointer to int, float,... | int*, float*,... |
| enumeration constant | enum *tag* {*name*$_1$=*value*$_1$,...}; |
| constant (read-only) value | *type* const *name*; |
| declare external variable | extern |
| internal to source file | static |
| local persistent between calls | static |
| no value | void |
| structure | struct *tag* {...}; |
| create new name for data type | typedef *type name*; |
| size of an object (type is size_t) | sizeof *object* |
| size of a data type (type is size_t) | sizeof(*type*) |

## Initialization

| | |
|---|---|
| initialize variable | *type name*=*value*; |
| initialize array | *type name*[]={*value*$_1$,...}; |
| initialize char string | char *name*[]="*string*"; |

## Constants

| | |
|---|---|
| suffix: long, unsigned, float | 65536L, -1U, 3.0F |
| exponential form | 4.2e1 |
| prefix: octal, hexadecimal | 0, 0x or 0X |
|   *Example.* 031 is 25, 0x31 is 49 decimal | |
| character constant (char, octal, hex) | 'a', '\*ooo*', '\x*hh*' |
| newline, cr, tab, backspace | \n, \r, \t, \b |
| special characters | \\\\, \?, \', \" |
| string constant (ends with '\0') | "abc...de" |

## Pointers, Arrays & Structures

| | |
|---|---|
| declare pointer to *type* | *type* **name*; |
| declare function returning pointer to *type* | *type* *f(); |
| declare pointer to function returning *type* | *type* (*pf)(); |
| generic pointer type | void * |
| null pointer constant | NULL |
| object pointed to by *pointer* | **pointer* |
| address of object *name* | &*name* |
| array | *name*[*dim*] |
| multi-dim array | *name*[*dim*$_1$][*dim*$_2$]... |

### Structures

| | |
|---|---|
|   struct *tag* { | structure template |
|     *declarations* | declaration of members |
|   }; | |

| | |
|---|---|
| create structure | struct *tag name* |
| member of structure from template | *name*.*member* |
| member of pointed-to structure | *pointer* -> *member* |
|   *Example.* (*p).x and p->x are the same | |
| single object, multiple possible types | union |
| bit field with *b* bits | unsigned *member*: *b*; |

## Operators (grouped by precedence)

| | |
|---|---|
| struct member operator | *name*.*member* |
| struct member through pointer | *pointer*->*member* |
| increment, decrement | ++, -- |
| plus, minus, logical not, bitwise not | +, -, !, ~ |
| indirection via pointer, address of object | **pointer*, &*name* |
| cast expression to type | (*type*) *expr* |
| size of an object | sizeof |
| multiply, divide, modulus (remainder) | *, /, % |
| add, subtract | +, - |
| left, right shift [bit ops] | <<, >> |
| relational comparisons | >, >=, <, <= |
| equality comparisons | ==, != |
| and [bit op] | & |
| exclusive or [bit op] | ^ |
| or (inclusive) [bit op] | \| |
| logical and | && |
| logical or | \|\| |
| conditional expression | *expr*$_1$ ? *expr*$_2$ : *expr*$_3$ |
| assignment operators | +=, -=, *=, ... |
| expression evaluation separator | , |

Unary operators, conditional expression and assignment operators group right to left; all others group left to right.

## Flow of Control

| | |
|---|---|
| statement terminator | ; |
| block delimiters | { } |
| exit from switch, while, do, for | break; |
| next iteration of while, do, for | continue; |
| go to | goto *label*; |
| label | *label*: statement |
| return value from function | return *expr* |

### Flow Constructions

| | |
|---|---|
| if statement | if (*expr*$_1$) *statement*$_1$<br>else if (*expr*$_2$) *statement*$_2$<br>else *statement*$_3$ |
| while statement | while (*expr*)<br>  *statement* |
| for statement | for (*expr*$_1$; *expr*$_2$; *expr*$_3$)<br>  *statement* |
| do statement | do *statement*<br>while(*expr*); |
| switch statement | switch (*expr*) {<br>    case *const*$_1$: *statement*$_1$ break;<br>    case *const*$_2$: *statement*$_2$ break;<br>    default: *statement*<br>  } |

## ANSI Standard Libraries

| | | | | |
|---|---|---|---|---|
| <assert.h> | <ctype.h> | <errno.h> | <float.h> | <limits.h> |
| <locale.h> | <math.h> | <setjmp.h> | <signal.h> | <stdarg.h> |
| <stddef.h> | <stdio.h> | <stdlib.h> | <string.h> | <time.h> |

## Character Class Tests <ctype.h>

| | |
|---|---|
| alphanumeric? | isalnum(c) |
| alphabetic? | isalpha(c) |
| control character? | iscntrl(c) |
| decimal digit? | isdigit(c) |
| printing character (not incl space)? | isgraph(c) |
| lower case letter? | islower(c) |
| printing character (incl space)? | isprint(c) |
| printing char except space, letter, digit? | ispunct(c) |
| space, formfeed, newline, cr, tab, vtab? | isspace(c) |
| upper case letter? | isupper(c) |
| hexadecimal digit? | isxdigit(c) |
| convert to lower case | tolower(c) |
| convert to upper case | toupper(c) |

## String Operations <string.h>

s is a string; cs, ct are constant strings

| | |
|---|---|
| length of s | strlen(s) |
| copy ct to s | strcpy(s,ct) |
| concatenate ct after s | strcat(s,ct) |
| compare cs to ct | strcmp(cs,ct) |
|   only first n chars | strncmp(cs,ct,n) |
| pointer to first c in cs | strchr(cs,c) |
| pointer to last c in cs | strrchr(cs,c) |
| copy n chars from ct to s | memcpy(s,ct,n) |
| copy n chars from ct to s (may overlap) | memmove(s,ct,n) |
| compare n chars of cs with ct | memcmp(cs,ct,n) |
| pointer to first c in first n chars of cs | memchr(cs,c,n) |
| put c into first n chars of s | memset(s,c,n) |

# C Reference Card (ANSI)

## Input/Output `<stdio.h>`

### Standard I/O

| | |
|---|---|
| standard input stream | `stdin` |
| standard output stream | `stdout` |
| standard error stream | `stderr` |
| end of file (type is `int`) | `EOF` |
| get a character | `getchar()` |
| print a character | `putchar(`*chr*`)` |
| print formatted data | `printf("`*format*`",`$arg_1$`,...)` |
| print to string `s` | `sprintf(s,"`*format*`",`$arg_1$`,...)` |
| read formatted data | `scanf("`*format*`",&`$name_1$`,...)` |
| read from string `s` | `sscanf(s,"`*format*`",&`$name_1$`,...)` |
| print string `s` | `puts(s)` |

### File I/O

| | |
|---|---|
| declare file pointer | `FILE *`*fp*`;` |
| pointer to named file | `fopen("`*name*`","`*mode*`")` |
| modes: `r` (read), `w` (write), `a` (append), `b` (binary) | |
| get a character | `getc(`*fp*`)` |
| write a character | `putc(`*chr*`,`*fp*`)` |
| write to file | `fprintf(`*fp*`,"`*format*`",`$arg_1$`,...)` |
| read from file | `fscanf(`*fp*`,"`*format*`",`$arg_1$`,...)` |
| read and store `n` elts to `*ptr` | `fread(*ptr,eltsize,n,`*fp*`)` |
| write `n` elts from `*ptr` to file | `fwrite(*ptr,eltsize,n,`*fp*`)` |
| close file | `fclose(`*fp*`)` |
| non-zero if error | `ferror(`*fp*`)` |
| non-zero if already reached EOF | `feof(`*fp*`)` |
| read line to string `s` (< `max` chars) | `fgets(s,max,`*fp*`)` |
| write string `s` | `fputs(s,`*fp*`)` |

**Codes for Formatted I/O**: `"%-+ 0`*w.pmc*`"`

| | |
|---|---|
| `-` | left justify |
| `+` | print with sign |
| *space* | print space if no sign |
| `0` | pad with leading zeros |
| *w* | min field width |
| *p* | precision |
| *m* | conversion character: |
| | `h` short, `l` long, `L` long double |
| *c* | conversion character: |

| | | | |
|---|---|---|---|
| `d,i` | integer | `u` | unsigned |
| `c` | single char | `s` | char string |
| `f` | double (`printf`) | `e,E` | exponential |
| `f` | float (`scanf`) | `lf` | double (`scanf`) |
| `o` | octal | `x,X` | hexadecimal |
| `p` | pointer | `n` | number of chars written |
| `g,G` | same as `f` or `e,E` depending on exponent | | |

## Variable Argument Lists `<stdarg.h>`

| | |
|---|---|
| declaration of pointer to arguments | `va_list` *ap*`;` |
| initialization of argument pointer | `va_start(`*ap*`,`*lastarg*`);` |
| *lastarg* is last named parameter of the function | |
| access next unnamed arg, update pointer | `va_arg(`*ap*`,`*type*`)` |
| call before exiting function | `va_end(`*ap*`);` |

## Standard Utility Functions `<stdlib.h>`

| | |
|---|---|
| absolute value of `int n` | `abs(n)` |
| absolute value of `long n` | `labs(n)` |
| quotient and remainder of `int`s `n,d` | `div(n,d)` |
| returns structure with `div_t.quot` and `div_t.rem` | |
| quotient and remainder of `long`s `n,d` | `ldiv(n,d)` |
| returns structure with `ldiv_t.quot` and `ldiv_t.rem` | |
| pseudo-random integer `[0,RAND_MAX]` | `rand()` |
| set random seed to `n` | `srand(n)` |
| terminate program execution | `exit(status)` |
| pass string `s` to system for execution | `system(s)` |

### Conversions

| | |
|---|---|
| convert string `s` to double | `atof(s)` |
| convert string `s` to integer | `atoi(s)` |
| convert string `s` to long | `atol(s)` |
| convert prefix of `s` to `double` | `strtod(s,&endp)` |
| convert prefix of `s` (base `b`) to `long` | `strtol(s,&endp,b)` |
| same, but `unsigned long` | `strtoul(s,&endp,b)` |

### Storage Allocation

| | |
|---|---|
| allocate storage | `malloc(size), calloc(nobj,size)` |
| change size of storage | `newptr = realloc(ptr,size);` |
| deallocate storage | `free(ptr);` |

### Array Functions

| | |
|---|---|
| search `array` for `key` | `bsearch(key,array,n,size,cmpf)` |
| sort `array` ascending order | `qsort(array,n,size,cmpf)` |

## Time and Date Functions `<time.h>`

| | |
|---|---|
| processor time used by program | `clock()` |
| *Example.* `clock()/CLOCKS_PER_SEC` is time in seconds | |
| current calendar time | `time()` |
| $\texttt{time}_2$`-`$\texttt{time}_1$ in seconds (`double`) | `difftime(`$\texttt{time}_2$`,`$\texttt{time}_1$`)` |
| arithmetic types representing times | `clock_t,time_t` |
| structure type for calendar time comps | `struct tm` |

| | |
|---|---|
| `tm_sec` | seconds after minute |
| `tm_min` | minutes after hour |
| `tm_hour` | hours since midnight |
| `tm_mday` | day of month |
| `tm_mon` | months since January |
| `tm_year` | years since 1900 |
| `tm_wday` | days since Sunday |
| `tm_yday` | days since January 1 |
| `tm_isdst` | Daylight Savings Time flag |

| | |
|---|---|
| convert local time to calendar time | `mktime(tp)` |
| convert time in `tp` to string | `asctime(tp)` |
| convert calendar time in `tp` to local time | `ctime(tp)` |
| convert calendar time to GMT | `gmtime(tp)` |
| convert calendar time to local time | `localtime(tp)` |
| format date and time info | `strftime(s,smax,"`*format*`",tp)` |
| `tp` is a pointer to a structure of type `tm` | |

## Mathematical Functions `<math.h>`

Arguments and returned values are `double`

| | |
|---|---|
| trig functions | `sin(x), cos(x), tan(x)` |
| inverse trig functions | `asin(x), acos(x), atan(x)` |
| arctan($y/x$) | `atan2(y,x)` |
| hyperbolic trig functions | `sinh(x), cosh(x), tanh(x)` |
| exponentials & logs | `exp(x), log(x), log10(x)` |
| exponentials & logs (2 power) | `ldexp(x,n), frexp(x,&e)` |
| division & remainder | `modf(x,ip), fmod(x,y)` |
| powers | `pow(x,y), sqrt(x)` |
| rounding | `ceil(x), floor(x), fabs(x)` |

## Integer Type Limits `<limits.h>`

The numbers given in parentheses are typical values for the constants on a 32-bit Unix system, followed by minimum required values (if significantly different).

| | | |
|---|---|---|
| `CHAR_BIT` | bits in `char` | (8) |
| `CHAR_MAX` | max value of `char` | (`SCHAR_MAX` or `UCHAR_MAX`) |
| `CHAR_MIN` | min value of `char` | (`SCHAR_MIN` or 0) |
| `SCHAR_MAX` | max `signed char` | (+127) |
| `SCHAR_MIN` | min `signed char` | (−128) |
| `SHRT_MAX` | max value of `short` | (+32,767) |
| `SHRT_MIN` | min value of `short` | (−32,768) |
| `INT_MAX` | max value of `int` | (+2,147,483,647) (+32,767) |
| `INT_MIN` | min value of `int` | (−2,147,483,648) (−32,767) |
| `LONG_MAX` | max value of `long` | (+2,147,483,647) |
| `LONG_MIN` | min value of `long` | (−2,147,483,648) |
| `UCHAR_MAX` | max `unsigned char` | (255) |
| `USHRT_MAX` | max `unsigned short` | (65,535) |
| `UINT_MAX` | max `unsigned int` | (4,294,967,295) (65,535) |
| `ULONG_MAX` | max `unsigned long` | (4,294,967,295) |

## Float Type Limits `<float.h>`

The numbers given in parentheses are typical values for the constants on a 32-bit Unix system.

| | | |
|---|---|---|
| `FLT_RADIX` | radix of exponent rep | (2) |
| `FLT_ROUNDS` | floating point rounding mode | |
| `FLT_DIG` | decimal digits of precision | (6) |
| `FLT_EPSILON` | smallest $x$ so $1.0\text{f} + x \neq 1.0\text{f}$ | (1.1E − 7) |
| `FLT_MANT_DIG` | number of digits in mantissa | |
| `FLT_MAX` | maximum `float` number | (3.4E38) |
| `FLT_MAX_EXP` | maximum exponent | |
| `FLT_MIN` | minimum `float` number | (1.2E − 38) |
| `FLT_MIN_EXP` | minimum exponent | |
| `DBL_DIG` | decimal digits of precision | (15) |
| `DBL_EPSILON` | smallest $x$ so $1.0 + x \neq 1.0$ | (2.2E − 16) |
| `DBL_MANT_DIG` | number of digits in mantissa | |
| `DBL_MAX` | max `double` number | (1.8E308) |
| `DBL_MAX_EXP` | maximum exponent | |
| `DBL_MIN` | min `double` number | (2.2E − 308) |
| `DBL_MIN_EXP` | minimum exponent | |

Send comments and corrections to J.H. Silverman, Math. Dept., Brown Univ., Providence, RI 02912 USA. ⟨jhs@math.brown.edu⟩