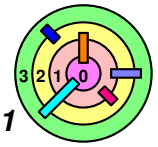
Warmup #1

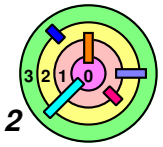
Bill Cheng

http://merlot.usc.edu/cs402-s13



Programming & Good Habbits

- ***Always*** **check return code!**
  - `open(), write()`
  - `malloc()`
  - `switch (errno) { ... }`
- **Initialize *all* variables!**
  - `int i=0;`
  - `struct timeval timeout;`
    `memset(&timeout, 0, sizeof(struct timeval));`
- ***Never*** **leak any resources!**
  - `malloc() and free()`
  - `open() and close()`
  - **Delete temporary files**

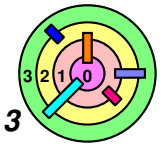


Programming & Good Habbits

- ***Don't*** **assume external input will be short**
  - use `strncpy()` and not `strcpy()`
  - use `snprintf()` and not `sprintf()`
  - use `sizeof()` and not a constant, for example,

```
unsigned char buf[80];

buf[0] = '\0'; /* initialization */
strncpy(buf, sizeof(buf), *argv[1]);
buf[sizeof(buf)-1] = '\0'; /* in case *argv[1] is long */
```

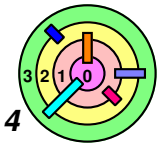
- **Fix your code so that you have *zero* compiler warnings!**
  - use `-Wall` when you compile to get all compiler warnings



Notes on gdb

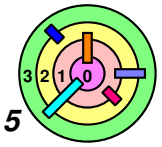
The debugger is your friend! Get to know it!

compile program with:	`-g`
start debugging:	`gdb warmup1`
set breakpoint:	`(gdb) break foo.c:123`
run program:	`(gdb) run`
clear breakpoint:	`(gdb) clear`
stack trace:	`(gdb) where`
print field:	`(gdb) print f.BlockType`
printf():	`(gdb) printf "%02x\n",buf[0]`
single-step at same level:	`(gdb) next`
single-step into a function:	`(gdb) step`
print field after every cmd:	`(gdb) display f.BlockType`
assignment:	`(gdb) set f.BlockType=0`
continue:	`(gdb) cont`
quit:	`(gdb) quit`



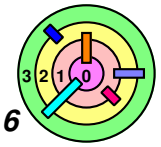
General Requirements

- Some major requirements for all programming assignments
  - severe pentalty for failing `make`
    - we will attempt to fix your Makefile you `make` fails
    - if we cannot get it to work, you need to figure out how to fix it by regrade time
  - severe pentalty for using large memory buffers
  - severe pentalty for any segmentation fault -- you must test your code well
  - if input file is large, you must not read the whole file into into a large memory buffer
    - must learn how to read a large file properly
  - severe pentalty for not using separate compilation or for having all your source code in header files -- you must learn to plan how to write your program



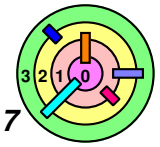
Grading Requirements

- It's important that every byte of your data is read and written correctly.
- For *warmup assignments*, you should run your code against the *grading guidelines*
  - must not change the commands there
    - we will change the data for actual grading, but we will stick to the commands (as much as we can)
  - to be fair to all, running scripts in the grading guidelines is *the only way we will grade*



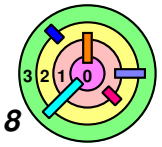
Separate Compilation

- Break up your code into *modules*
  - *compile the modules separately*, at least one rule per module per rule in the `Makefile`
  - a separate rule to *link* all the modules together
    - if your program requites additional libraries, add them to the link stage
- To receive full credit for separate compilation
  - to create an executable, at a minimum, you must run the compiler at least *twice* and the linker *once*



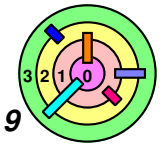
Code Design - Functional vs. Procedural

- Don't design your program "procedurally"
- You need to learn how to write functions!
  - a function has a well-defined interface
    - what are the meaning of the parameters
    - what does it suppose to return
  - pre-conditions
    - what must be true when the function is entered
    - you assume that these are true
      - you can verify it if you want
  - post-conditions
    - what must be true when the function returns
  - you design your program by making designing a sequence of function calls



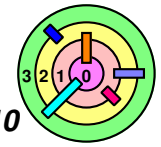
Warmup #1

- 2 parts
  - develop a *doubly-linked circular list* called *My402List*
    - to implement a *linked-list abstraction*
  - use your doubly-linked circular list to implement a command:
    - *sort* - sort a list of bank transactions



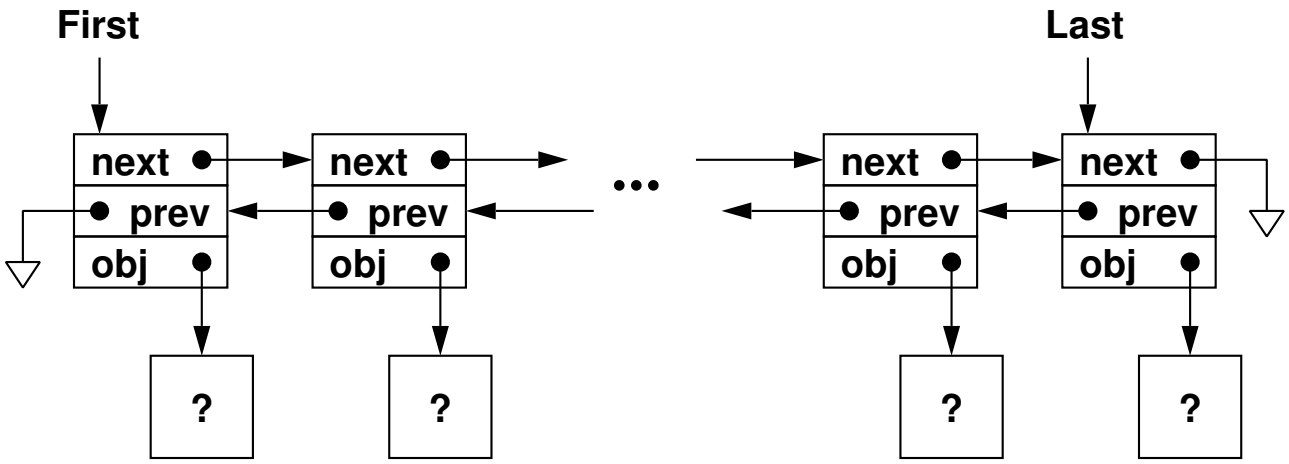
A Linked-List Abstraction

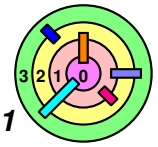
- A list of elements, linked so that you can move from one to the next (and/or previoius)
  - each element holds an object of some sort
- *Functionally:*
  - First()
  - Next()
  - Last()
  - Prev()
  - Insert()
  - Remove()
  - Count()
- Need to have a well-defined interface
  - once you have a good interface, if the implementation is broken, fix the implementation!
    - don't fix the "application"



A Linked-List Abstraction

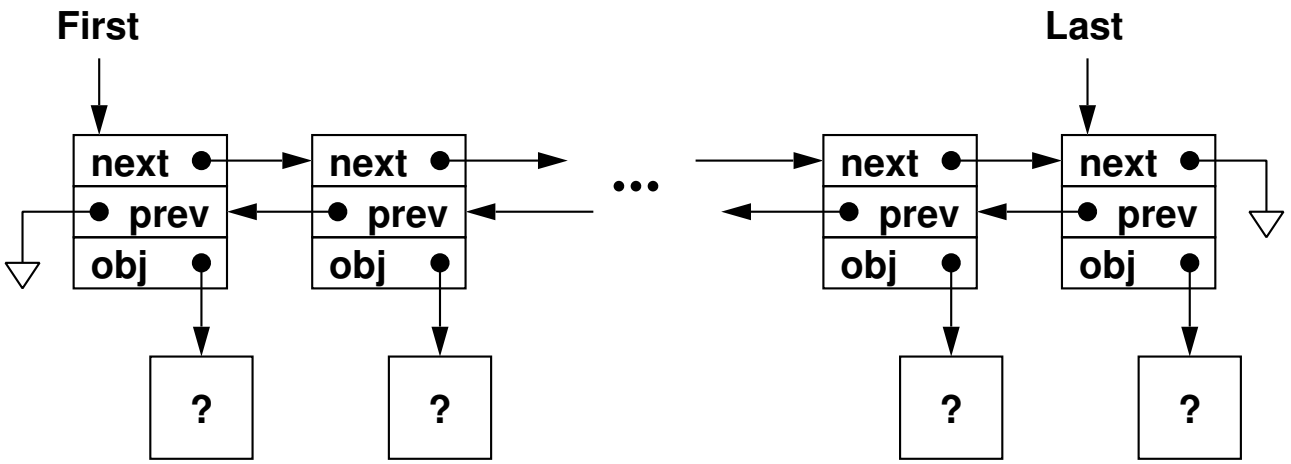
- There are basically two types of lists
  1) next/prev pointers in list items
  2) next/prev pointers outside of list items
- (1) has a major drawback that a list item cannot be inserted into multiple lists
  - We will implement (2)





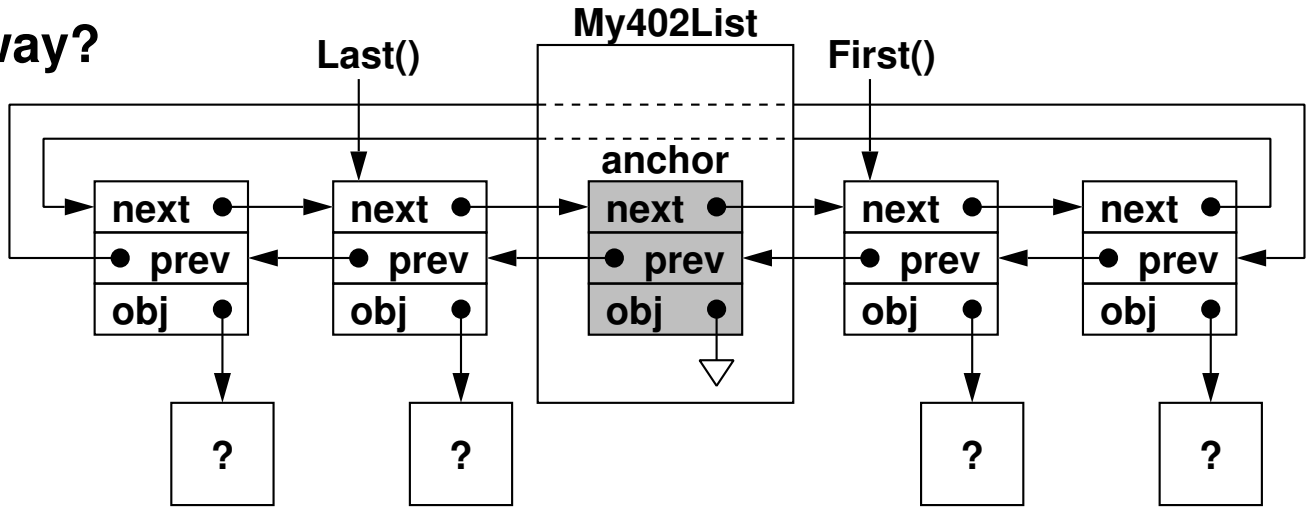
Doubly-linked Circular List

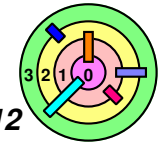
Abstraction



Implementation

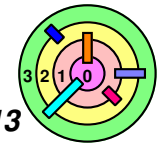
- why this way?





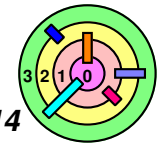
my402list.h

```
#ifndef _MY402LIST_H_
#define _MY402LIST_H_

#include "cs402.h"

typedef struct tagMy402ListElem {
    void *obj;
    struct tagMy402ListElem *next;
    struct tagMy402ListElem *prev;
} My402ListElem;

typedef struct tagMy402List {
    int num_members;
    My402ListElem anchor;

    /* You do not have to set these function pointers */
    int   (*Length)(struct tagMy402List *);
    int   (*Empty)(struct tagMy402List *);

    int   (*Append)(struct tagMy402List *, void*);
    int   (*Prepend)(struct tagMy402List *, void*);
    void  (*Unlink)(struct tagMy402List *, My402ListElem*);
    void  (*UnlinkAll)(struct tagMy402List *);
```

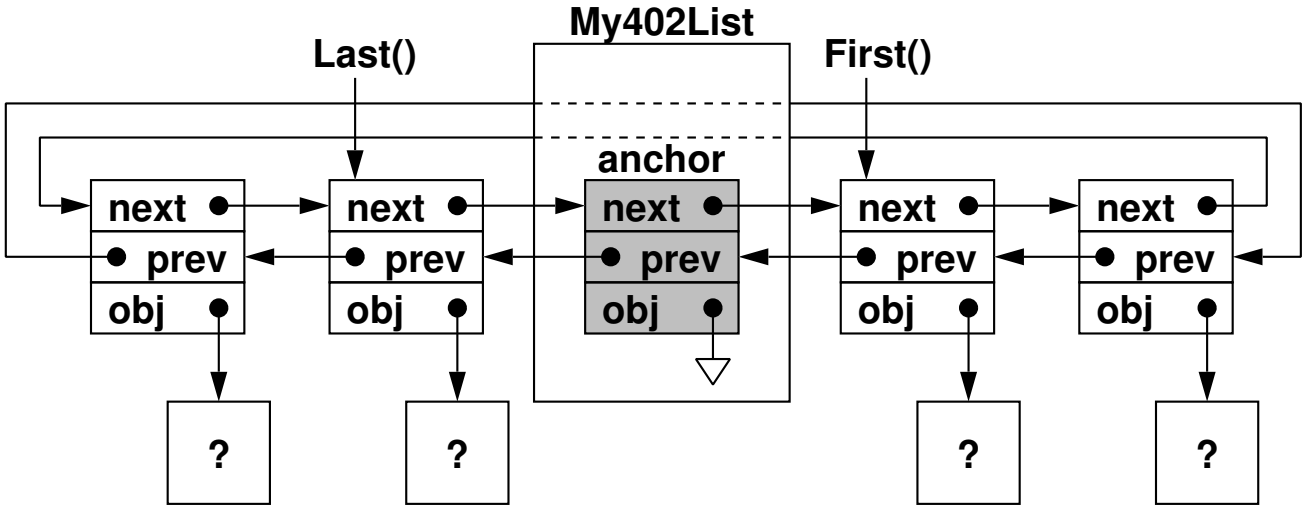


my402list.h

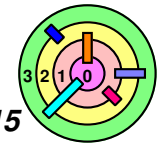
```
    int (*InsertBefore)(struct tagMy402List *, void*, My402ListElem*);
    int (*InsertAfter)(struct tagMy402List *, void*, My402ListElem*);

    My402ListElem *(*First)(struct tagMy402List *);
    My402ListElem *(*Last)(struct tagMy402List *);
    My402ListElem *(*Next)(struct tagMy402List *, My402ListElem *);
    My402ListElem *(*Prev)(struct tagMy402List *, My402ListElem *);

    My402ListElem *(*Find)(struct tagMy402List *, void *obj);
} My402List;

extern int  My402ListLength ARGS_DECL((My402List*));
extern int  My402ListEmpty ARGS_DECL((My402List*));

extern int  My402ListAppend ARGS_DECL((My402List*, void*));
extern int  My402ListPrepend ARGS_DECL((My402List*, void*));
extern void My402ListUnlink ARGS_DECL((My402List*, My402ListElem*));
extern void My402ListUnlinkAll ARGS_DECL((My402List*));
extern int  My402ListInsertAfter ARGS_DECL((My402List*, void*, My402ListElem*));
extern int  My402ListInsertBefore ARGS_DECL((My402List*, void*, My402ListElem*));

extern My402ListElem My402ListFirst ARGS_DECL((My402List*));
extern My402ListElem *My402ListLast ARGS_DECL((My402List*));
extern My402ListElem *My402ListNext ARGS_DECL((My402List*, My402ListElem*));
extern My402ListElem *My402ListPrev ARGS_DECL((My402List*, My402ListElem*));

extern My402ListElem *My402ListFind ARGS_DECL((My402List*, void*));

extern int  My402ListInit ARGS_DECL((My402List*));
#endif /*_MY402LIST_H_*/
```



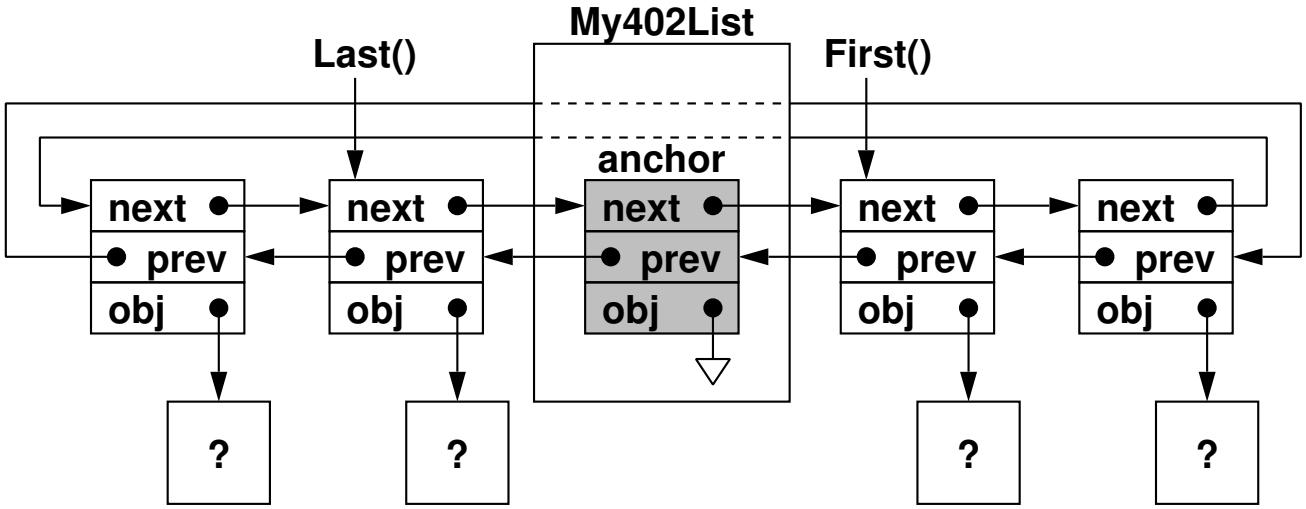
Implementation




```
int Length() { return num_members; }
int Empty() { return num_members<=0; }

int Append(void *obj);
int Prepend(void *obj);
void Unlink(My402ListElem*);
void UnlinkAll();
int InsertBefore(void *obj, My402ListElem *elem);
int InsertAfter(void *obj, My402ListElem *elem);

My402ListElem *First();
My402ListElem *Last();
My402ListElem *Next(My402ListElem *cur);
My402ListElem *Prev(My402ListElem *cur);

My402ListElem *Find(void *obj);
```



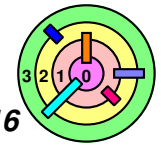
Usage - Traversing the List




```
void Traverse(My402List *list)
{
  My402ListElem *elem=NULL;

  for (elem=My402ListFirst(list);
       elem != NULL;
       elem=My402ListNext(list, elem)) {
    Foo *foo=(Foo*)(elem->obj);

    /* access foo here */
  }
}
```

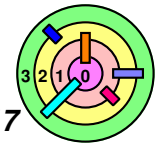
⇨ This is how an *application* will use `My402List`

- you must support your application



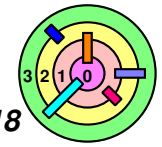
listtest

- Use provided `listtest.c` and `Makefile` to create `listtest`
  - `listtest` must run without error and you must not change `listtest.c` and `Makefile`
  - They specifies how your code is expected to be used
- You should learn how to run `listtest` under `gdb`



Sort Command

- `warmup1 sort [tfile]`
  - Produce a sorted transaction history for the transaction records in `tfile` (or `stdin`) and compute balances
- Input is an ASCII text file
  - Each line in a `tfile` contains 4 fields delimited by `<TAB>`
    - transcation type (single character)
      - `"+"` for deposit
      - `"-"` for withdrawal
    - transcation time (UNIX time)
      - `man -s 2 time`
    - amount (a number, a period, two digits)
    - transcation description (textual description)
      - cannot be empty



Reading Text Input

- Read in an entire line using `fgets()`
  - especially since we know the maximum line length, according to the spec
- If a filename is given, use `fopen()` to get a file pointer (`FILE*`)

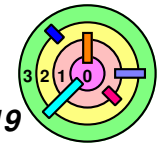
```
FILE *fp = fopen(..., "r");
```

  - read man pages of `fopen()`
  - if a filename is not given, you will be reading from "standard input" (i.e., file descriptor 0)

```
FILE *fp = stdin;
```

  - pass the file pointer around so that you run the same code whether you input comes from a file or `stdin`

```
My420List list;
if (!My402ListInit(&list)) { /* error */ }
if (!ReadInput(fp, &list)) { /* error */ }
if (fp != stdin) fclose(fp);
SortInput(&list);
PrintStatement(&list);
```



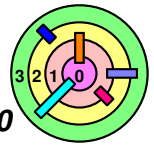
Parsing Text Input

- Read a line

```
char buf[1026];
if (fgets(buf, sizeof(buf), fp) == NULL) {
  /* end of file */
} else {
  /* parse it */
}
```

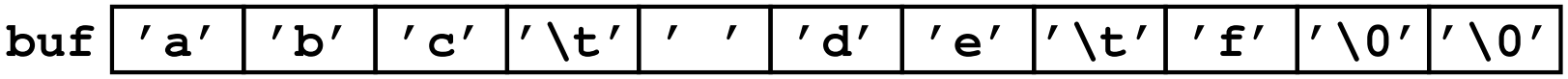
- Parse a line according to the spec
  - find an ***easy*** and ***correct*** way to parse the line
    - according to the spec, each line must have exactly 3 <TAB> characters
    - I think it's easy and correct to go after this

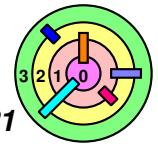
```
char *start_ptr = buf;
char *tab_ptr = strchr(start_ptr, '\t');
if (tab_ptr != NULL) {
  *tab_ptr++ = '\0';
}
/* start_ptr now contains a
   "null-terminated string" */
```



Parsing Text Input

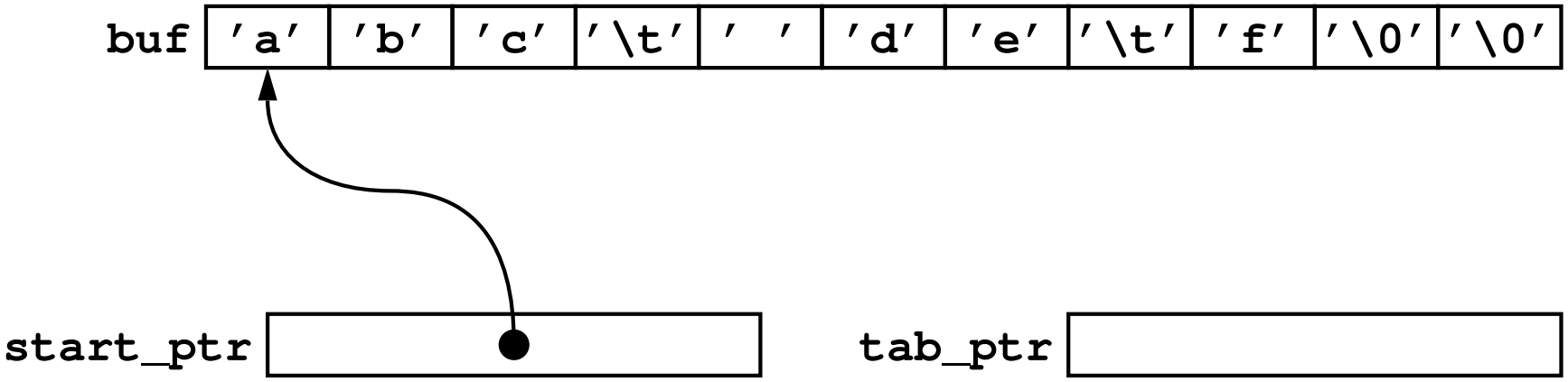
```
char *start_ptr = buf;
char *tab_ptr = strchr(start_ptr, '\t');
if (tab_ptr != NULL) {
  *tab_ptr++ = '\0';
}
/* start_ptr now contains a
   "null-terminated string" */
```

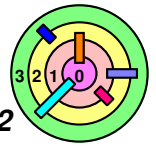




Parsing Text Input

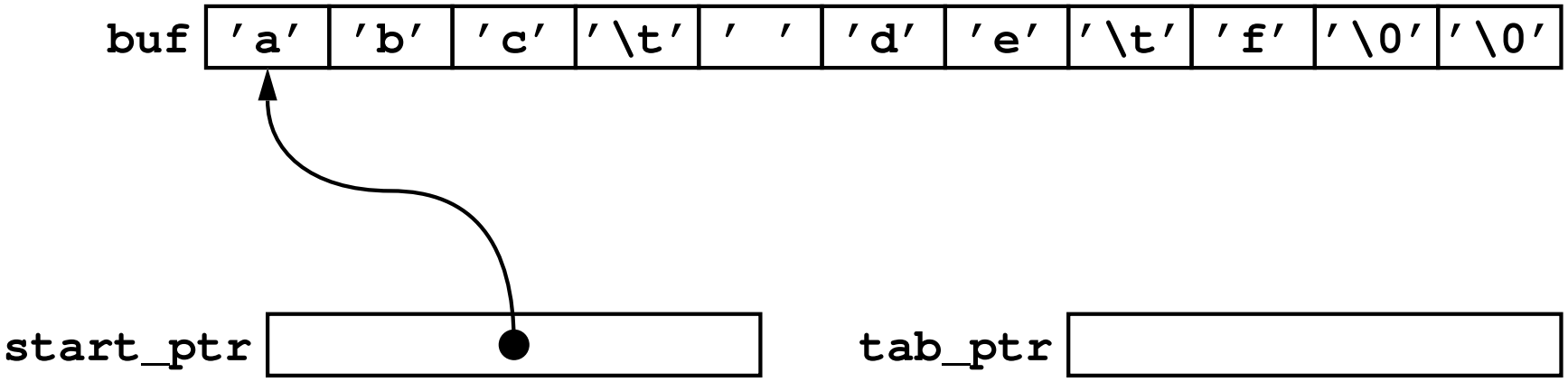
```
char *start_ptr = buf;
char *tab_ptr = strchr(start_ptr, '\t');
if (tab_ptr != NULL) {
  *tab_ptr++ = '\0';
}
/* start_ptr now contains a
   "null-terminated string" */
```

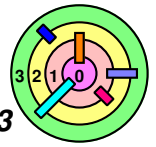




Parsing Text Input

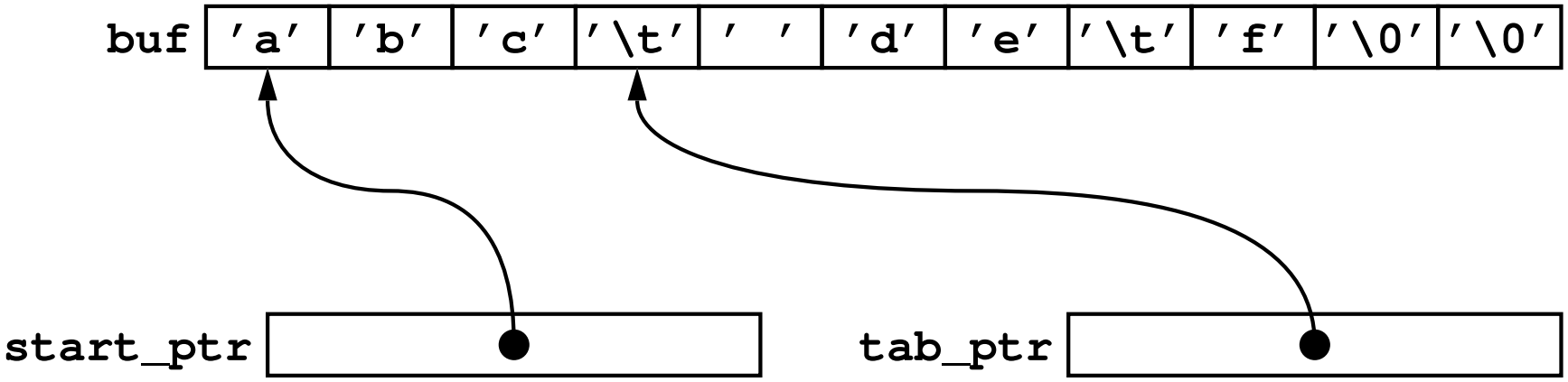
```
char *start_ptr = buf;
char *tab_ptr = strchr(start_ptr, '\t');
if (tab_ptr != NULL) {
  *tab_ptr++ = '\0';
}
/* start_ptr now contains a
   "null-terminated string" */
```

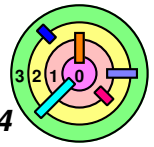




Parsing Text Input

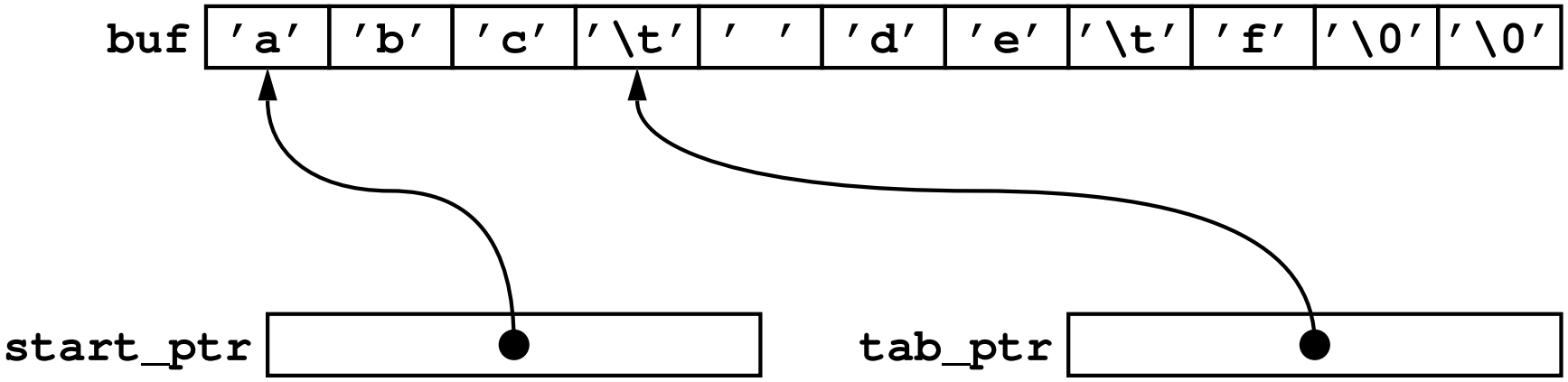
```
char *start_ptr = buf;
char *tab_ptr = strchr(start_ptr, '\t');
if (tab_ptr != NULL) {
  *tab_ptr++ = '\0';
}
/* start_ptr now contains a
   "null-terminated string" */
```

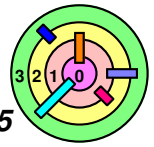




Parsing Text Input

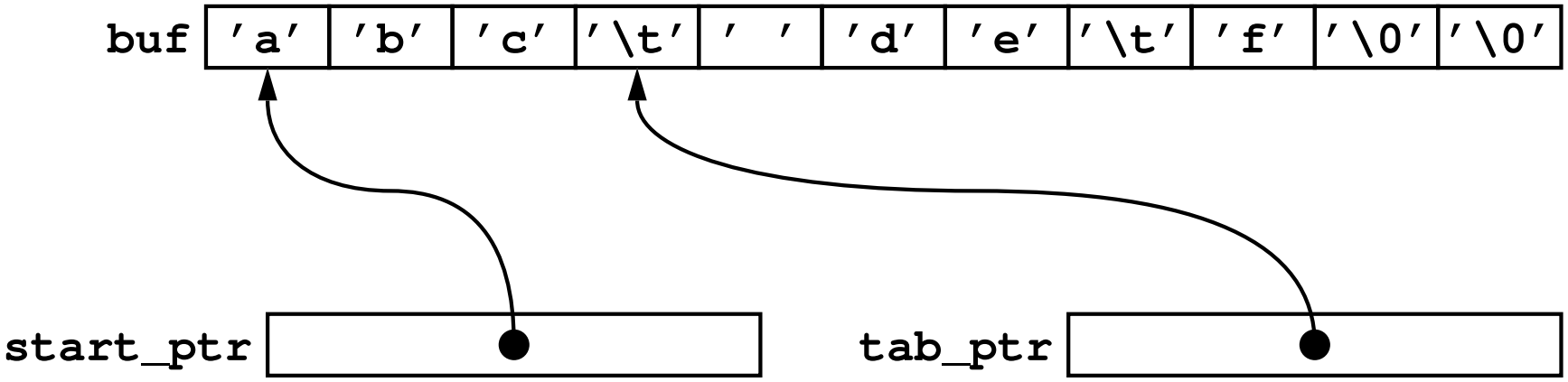
```
char *start_ptr = buf;
char *tab_ptr = strchr(start_ptr, '\t');
if (tab_ptr != NULL) {
  *tab_ptr++ = '\0';
}
/* start_ptr now contains a
   "null-terminated string" */
```

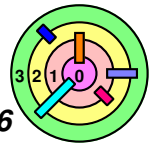




Parsing Text Input

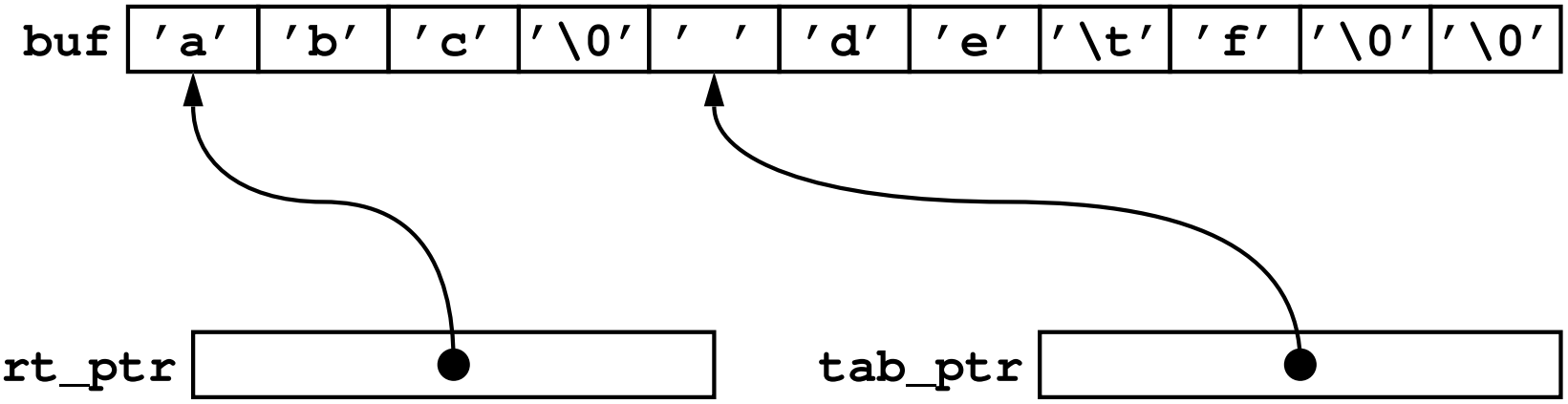
```
char *start_ptr = buf;
char *tab_ptr = strchr(start_ptr, '\t');
if (tab_ptr != NULL) {
  *tab_ptr++ = '\0';
}
/* start_ptr now contains a
   "null-terminated string" */
```

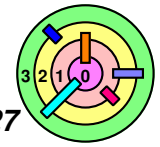




Parsing Text Input

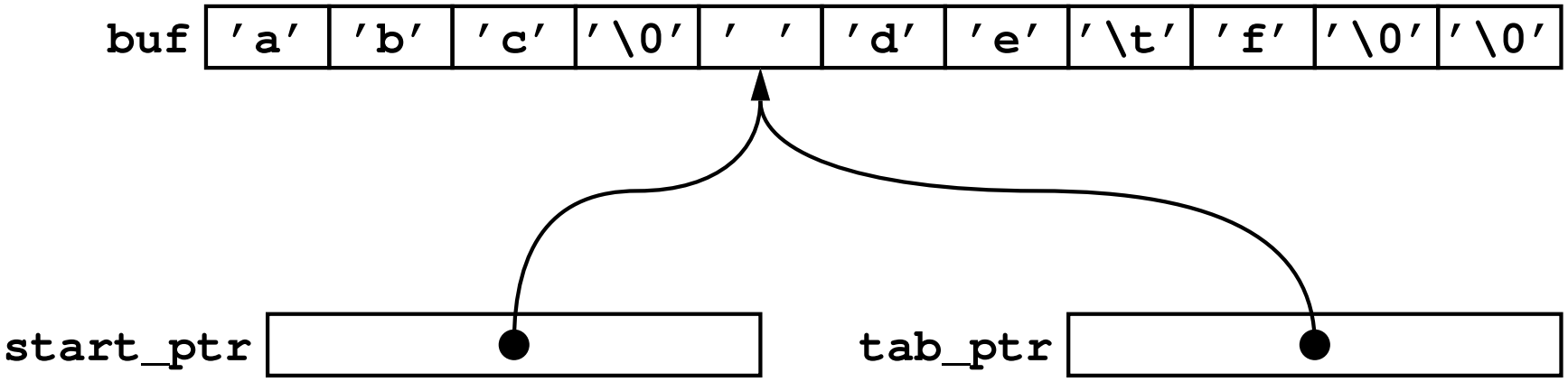
```
char *start_ptr = buf;
char *tab_ptr = strchr(start_ptr, '\t');
if (tab_ptr != NULL) {
  *tab_ptr++ = '\0';
}
/* start_ptr now contains a
   "null-terminated string" */
```

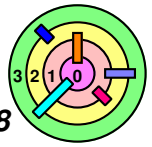




Parsing Text Input (2nd Iteration)

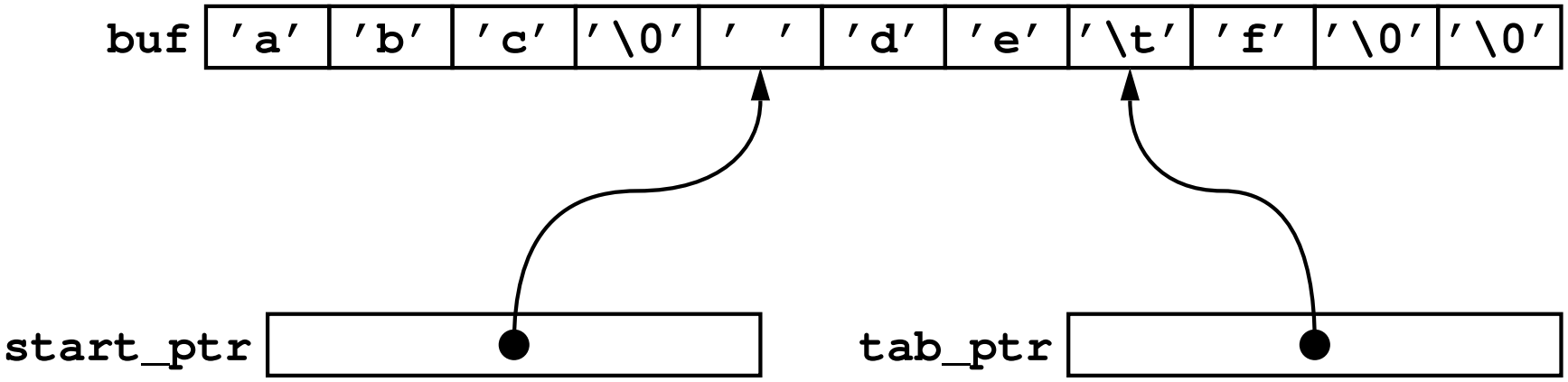
```
start_ptr = tab_ptr;
tab_ptr = strchr(start_ptr, '\t');
if (tab_ptr != NULL) {
  *tab_ptr++ = '\0';
}
/* start_ptr now contains a
   "null-terminated string" */
```

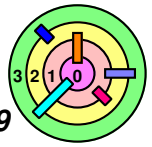




Parsing Text Input (2nd Iteration)

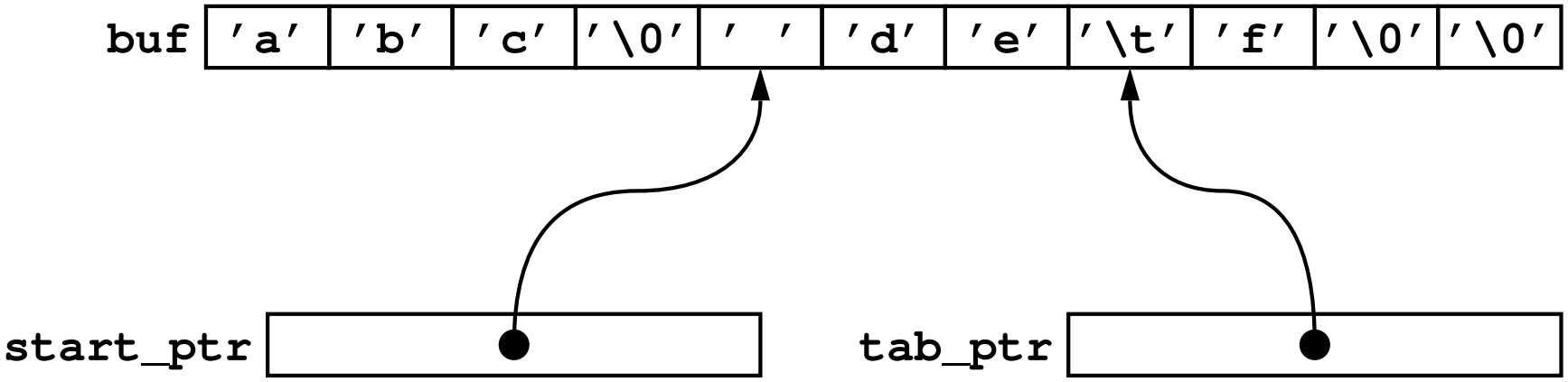
```
start_ptr = tab_ptr;
tab_ptr = strchr(start_ptr, '\t');
if (tab_ptr != NULL) {
  *tab_ptr++ = '\0';
}
/* start_ptr now contains a
   "null-terminated string" */
```

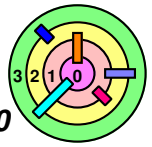




Parsing Text Input (2nd Iteration)

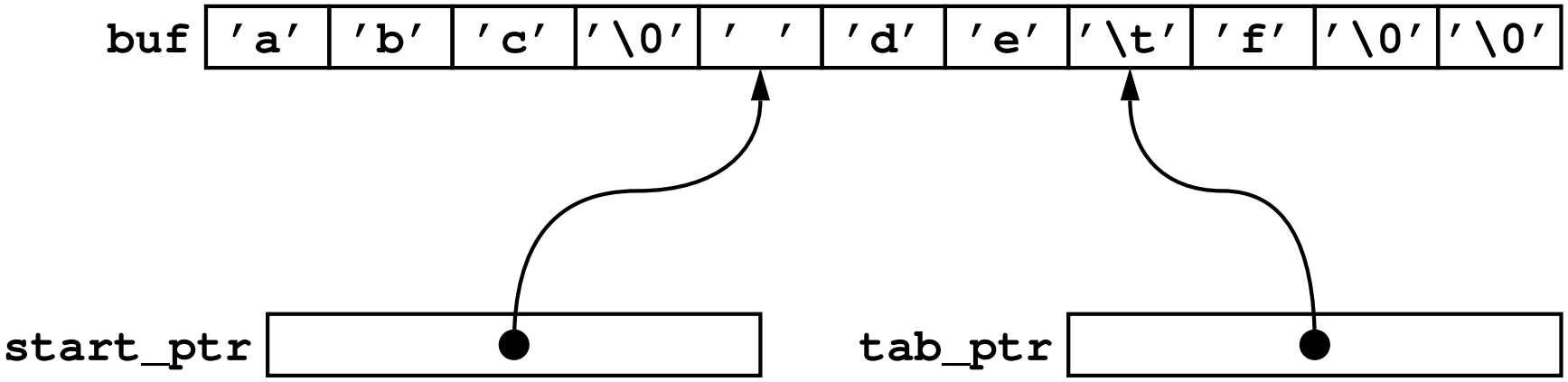
```
start_ptr = tab_ptr;
tab_ptr = strchr(start_ptr, '\t');
if (tab_ptr != NULL) {
  *tab_ptr++ = '\0';
}
/* start_ptr now contains a
   "null-terminated string" */
```

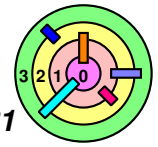




Parsing Text Input (2nd Iteration)

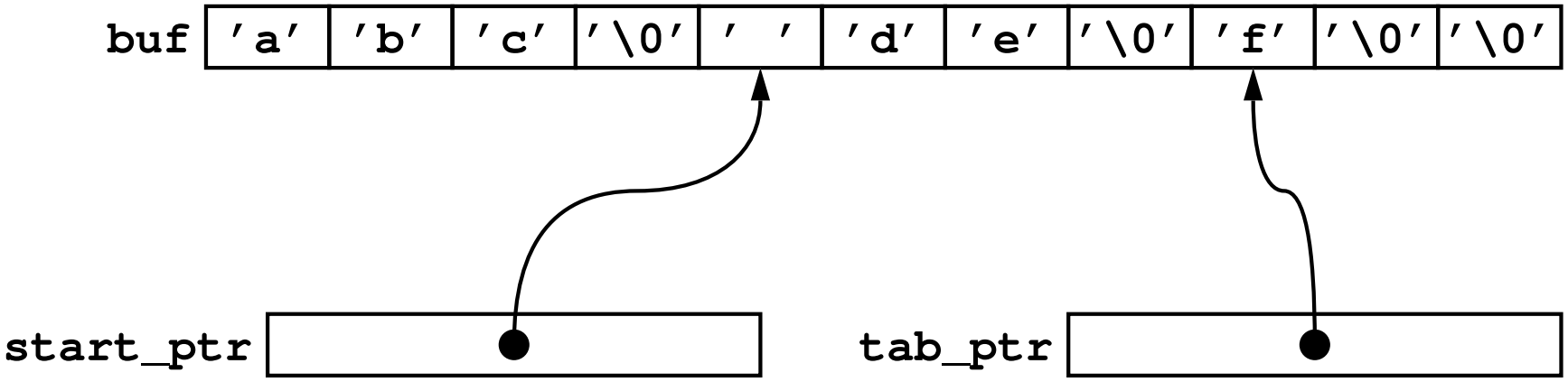
```
start_ptr = tab_ptr;
tab_ptr = strchr(start_ptr, '\t');
if (tab_ptr != NULL) {
  *tab_ptr++ = '\0';
}
/* start_ptr now contains a
   "null-terminated string" */
```

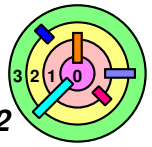




Parsing Text Input (2nd Iteration)

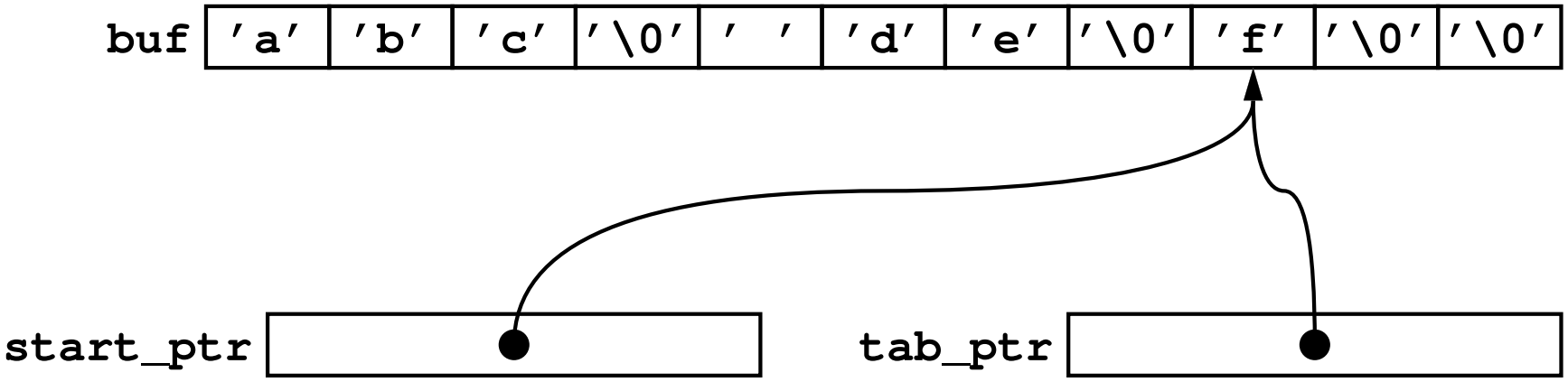
```
start_ptr = tab_ptr;
tab_ptr = strchr(start_ptr, '\t');
if (tab_ptr != NULL) {
  *tab_ptr++ = '\0';
}
/* start_ptr now contains a
   "null-terminated string" */
```

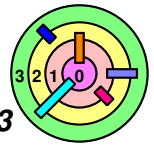




Parsing Text Input (3rd Iteration)

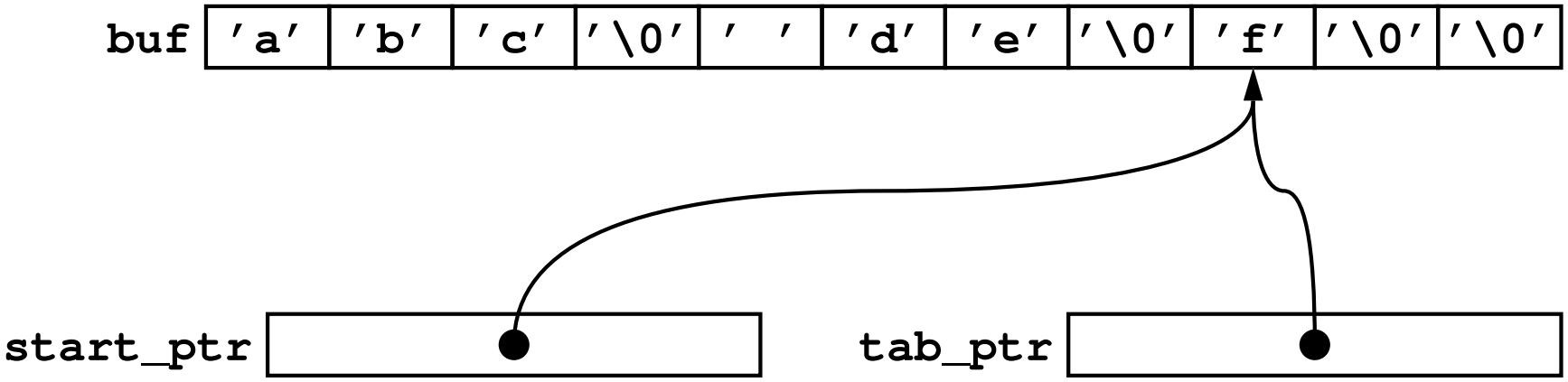
```
start_ptr = tab_ptr;
tab_ptr = strchr(start_ptr, '\t');
if (tab_ptr != NULL) {
  *tab_ptr++ = '\0';
}
/* start_ptr now contains a
   "null-terminated string" */
```

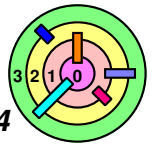




Parsing Text Input (3rd Iteration)

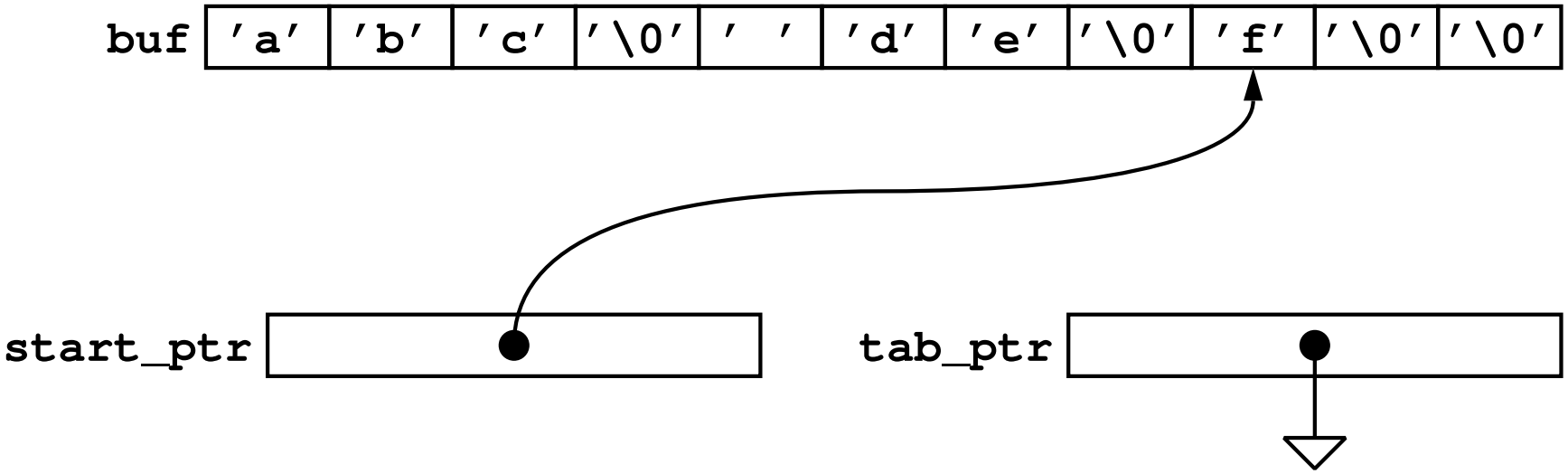
```
start_ptr = tab_ptr;
tab_ptr = strchr(start_ptr, '\t');
if (tab_ptr != NULL) {
  *tab_ptr++ = '\0';
}
/* start_ptr now contains a
   "null-terminated string" */
```

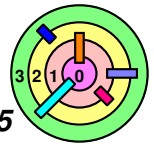




Parsing Text Input (3rd Iteration)

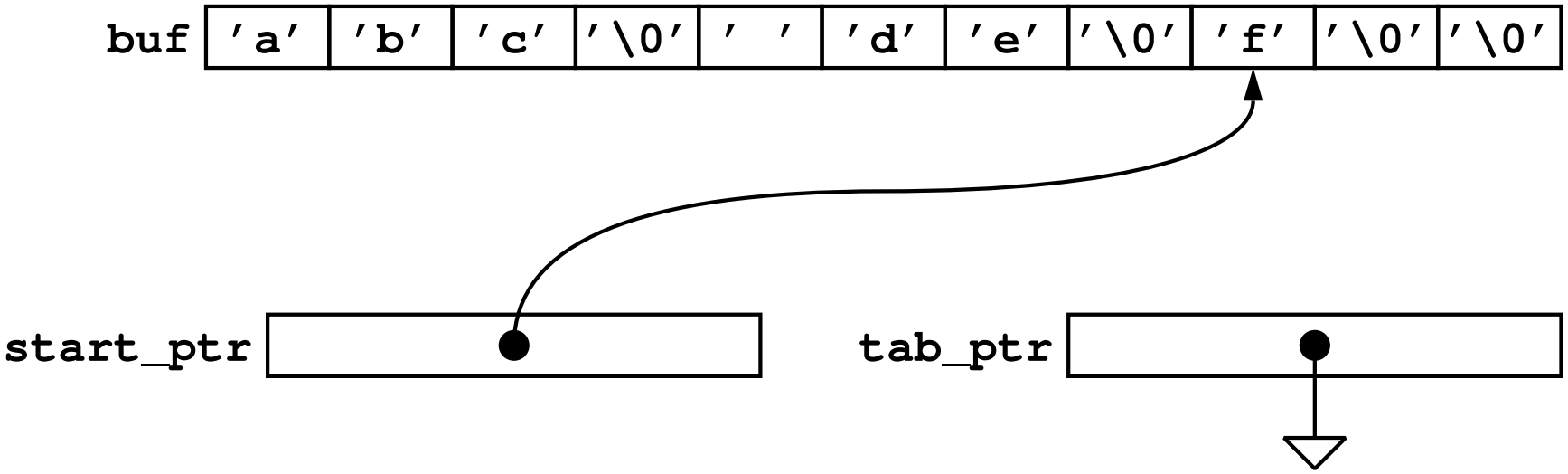
```
start_ptr = tab_ptr;
tab_ptr = strchr(start_ptr, '\t');
if (tab_ptr != NULL) {
  *tab_ptr++ = '\0';
}
/* start_ptr now contains a
   "null-terminated string" */
```

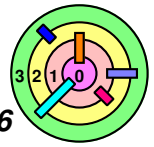




Parsing Text Input (3rd Iteration)

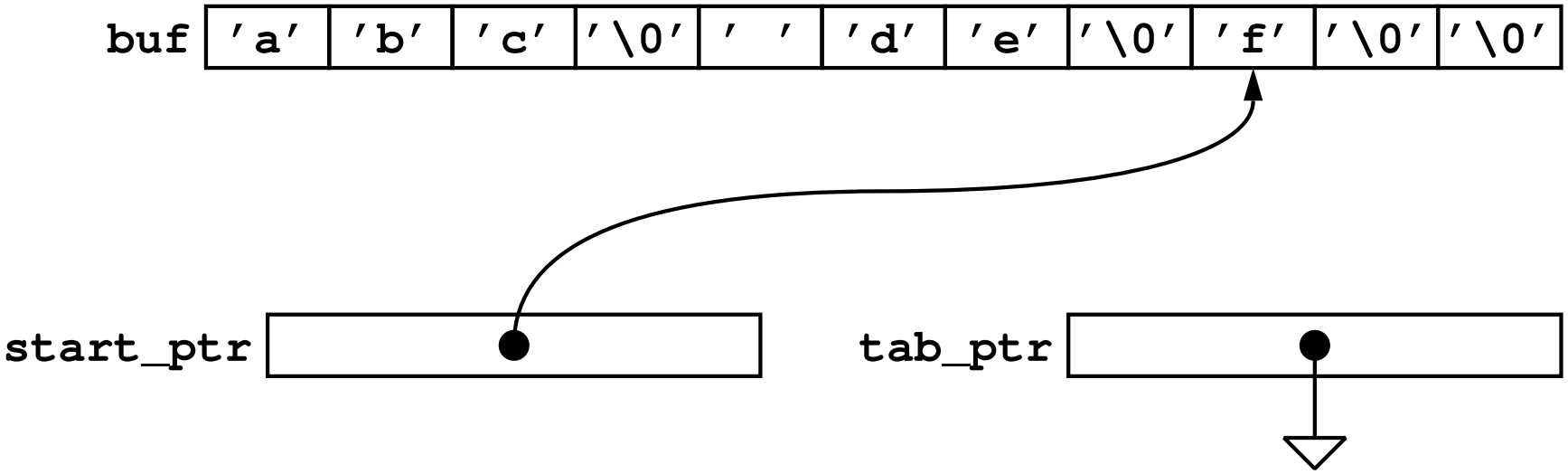
```
start_ptr = tab_ptr;
tab_ptr = strchr(start_ptr, '\t');
if (tab_ptr != NULL) {
  *tab_ptr++ = '\0';
}
/* start_ptr now contains a
   "null-terminated string" */
```

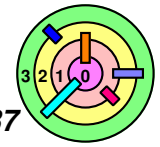




Parsing Text Input (3rd Iteration)

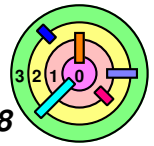
```
start_ptr = tab_ptr;
tab_ptr = strchr(start_ptr, '\t');
if (tab_ptr != NULL) {
  *tab_ptr++ = '\0';
}
/* start_ptr now contains a
   "null-terminated string" */
```





Validate Input

- Make sure every null-terminated string contains the right kind of value
  - if incorrect, print a *reasonable error message* and *quit* your program
    - ideally, you should clean up all your data structures (not required for an assignment like this one)
- After all fields are validated, you can put them in *one* data structure
  - allocate memory for this data structure and copy the fields into it
  - *append* pointer to this data structure to `list`
    - any pointer is compatible with (`void*`)
    - alternately, you can perform insertion sort by finding the right place to insert this pointer and call one of the insert functions of `My402List`



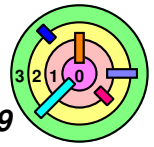
Sort Command

Output

```
000000000111111111122222222223333333333444444444455555555556666666666777777777 8
123456789012345678901234567890123456789012345678901234567890123456789012345678 90

+----------------+-------------------------+----------------+----------------+
|       Date     | Description             |         Amount |        Balance |
+----------------+-------------------------+----------------+----------------+
| Thu Aug 21 2008 | ...                    |       1,723.00 |       1,723.00 |
| Wed Dec 31 2008 | ...                    | (        45.33) |       1,677.67 |
| Mon Jul 13 2009 | ...                    |     10,388.07  |      12,065.74 |
| Sun Jan 10 2010 | ...                    | (       654.32) |      11,411.42 |
+----------------+-------------------------+----------------+----------------+
```

How to keep track of balance

- First thing that comes to mind is to use `double`
- The weird thing is that if you are not very careful with double, your output will be wrong (by 1 penny) once in a while
- Recommendation: keep the balance in cents, not dollars
  - No precision problem with integers!

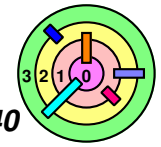


Sort Command

```
000000000111111111122222222223333333333444444444455555555556666666666777777777 8
123456789012345678901234567890123456789012345678901234567890123456789012345678 90

+----------------+--------------------------+----------------+----------------+
|       Date     | Description              |         Amount |        Balance |
+----------------+--------------------------+----------------+----------------+
| Thu Aug 21 2008 | ...                     |      1,723.00  |      1,723.00  |
| Wed Dec 31 2008 | ...                     | (       45.33) |      1,677.67  |
| Mon Jul 13 2009 | ...                     |     10,388.07  |     12,065.74  |
| Sun Jan 10 2010 | ...                     | (      654.32) |     11,411.42  |
+----------------+--------------------------+----------------+----------------+
```

- The spec requires you to call `ctime()` to convert a Unix timestamp to string
  - then pick the right characters to display as date
  - e.g., `ctime()` returns `"Thu Aug 30 08:17:32 2012\n"`
    - becareful, `ctime()` returns a pointer that points to a *global variable*, so you must *make a copy*

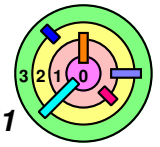
```
char date[16];
char buf[26];
strncpy(buf, sizeof(buf), ctime(...));
date[0] = buf[0];
date[1] = buf[1];
...
date[15] = '\0';
```



Sort Command

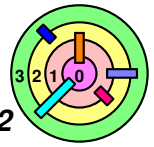
```
0000000001111111111222222222233333333334444444444555555555566666666667777777778
1234567890123456789012345678901234567890123456789012345678901234567890123456789

+-----------------+--------------------------+----------------+----------------+
|       Date      | Description              |         Amount |        Balance |
+-----------------+--------------------------+----------------+----------------+
| Thu Aug 21 2008 | ...                      |       1,723.00 |       1,723.00 |
| Wed Dec 31 2008 | ...                      | (        45.33) |       1,677.67 |
| Mon Jul 13 2009 | ...                      |      10,388.07 |      12,065.74 |
| Sun Jan 10 2010 | ...                      | (       654.32) |      11,411.42 |
+-----------------+--------------------------+----------------+----------------+
```

- Format your data in your own buffer
  - write a function to "format" numeric fields into null-terminated strings
    - it's a little more work, but you really should have this code isolated
      - in case you have bugs, just fix this function
  - you can even do the formatting when you append or insert your data structure to your list
    - need more fields in your data structure
  - this way, you can just print things out easily
  - use `printf("%s", ...)` to print a field to `stdout`



Warmup #1

- I'm giving you a lot of details on how to do things in C
  - this is the first and last assignment that I will do this!
  - you must learn C on your own
- Read man pages
- Ask questions in class Google Group
  - or send e-mail to me
- Come to office hours, especially if you are stuck



Warmup #1 - Miscellaneous Requirements

- Run your code against the *grading guidelines*
  - must not change the test program
- You must not use any *external code fragments*
- You must not use *array* to implement any list functions
  - must use pointers
- If input file is large, you must not read the whole file into into a large memory buffer
- It's important that every byte of your data is read and written correctly.
  - `diff` commands in the grading guidelines must *not* produce *any* output or you will not get credit
- Please see Warmup #1 spec for additional details
  - please read the *entire* spec *yourself*