Structures Containing Strings: two examples; Memory Reallocation

CS2023 Winter 2004

Structures Containing Pointers

- Structures can also include pointers, which can point to dynamically allocated memory

```
typedef struct{
  char *string;
  int count;
}StringTabT;
```

- Use this structure to create table of strings and their occurence

String Table Module

- Create module to store strings and their occurence in a table

```
StringTabT *initTable(void);
```

  - allocates memory for table and initializes it to zero; returns NULL if can't allocate. Use preprocessor macro in module to define size of table.

```
int addStringTable(char *string, StringTabT
  *table);
```

  - If string already in table, increments count and returns 1
  - If string not in table, adds it to table and returns 0
  - If error occurs, returns -1

String Table Module

```
void printTable(StringTabT *table);
```

– Prints table to stdout, showing each string and its count

```
void clearTable(StringTabT **ptable);
```

– Deletes table, freeing all memory that was allocated

Header File

```
typedef struct{
  char *string;
  int count;
}StringTabT;

#define SIZE 5000

StringTabT *initTable(void);

int addStringTable(char *string, StringTabT
*table);

void printTable(StringTabT *table);

void clearTable(StringTabT **ptable);
```

```c
#include "stringTable.h"

StringTabT *initTable(void)
{
  StringTabT *p;
  p = calloc(SIZE,sizeof(StringTabT));
  return p;
}
```

```
Void printTable(StringTabT *table)
{
  StringTabT *p;
  for(p = table; p < table + SIZE &&
              p->string != NULL; p++)
    printf("%s: %d\n", p->string, p->count);
}
```

```
int addStringTable(char *string, StringTabT
  *table)
{
  StringTabT *p;
  for(p = table; p < table + SIZE &&
              p->string != NULL; p++)
    if(strcmp(string,p->string) == 0){
      p->count++;
      return 1;
    }
  if(p == table + SIZE)
    return -1;
  if((p->string = strdup(string)) == NULL)
    return -1;
  p->count = 1;
  return 0;
}
```

```
void clearTable(StringTabT **ptable)
{
  StringTabT *p;

  for(p = *ptable; p < *ptable + SIZE &&
                p->string != NULL; p++)
    free(p->string);
  free(*ptable);
  *ptable = NULL;
}
```

Application of String Table

```
#include "stringTab.h"
...
int main(){
  char word[50];
  StringTabT *wordTable;
  wordTable = initTable();
  while(scanf("%50s",word) == 1){
    if(addStringTable(word, wordTable) == -1){
      fprintf(stderr,"Error creating table\n");
      return 1;
    }
  }
  printTable(wordTable);
  clearTable(&wordTable);
  return 0;
```

Comments on String Table

- Exposing the `StringTabT` type in the header file allows client to create multiple tables
  - Details of implementation exposed to client
- Could hide the data structure by creating a *singleton* module (textbook, 7.4.7), at the cost of only allowing one table to be created at a time

Header File: Singleton String Table

```
/* initTable: allocates memory for table
 * (of length size) and initializes it to
 * zero. Returns 0 if allocation fails, 1 if
 * successful
 */
int initTable(int size);

int addStringTable(char *string);

void printTable(void);

void clearTable(void);
```

```c
#include "stringTable.h"

typedef struct{
  char *string;
  int count;
}StringTabT;
StringTabT *table;
int tableSize;

int initTable(int size)
{
  if((table = calloc(size,sizeof(StringTabT))) ==
  NULL)
    return 0;
  tableSize = size;
  return 1;
}
```

```
void printTable(void){
  StringTabT *p;

  for(p = table; p < table + tableSize && p-
  >string != NULL; p++)
    printf("%s: %d\n", p->string, p->count);
}

void clearTable(void){
  StringTabT *p;

  for(p = table; p < table + tableSize && p-
  >string != NULL; p++)
    free(p->string);
  free(table);
  table = NULL;
}
```

```
int addStringTable(char *string)
{
  StringTabT *p;

  for(p = table; p < table + tableSize &&
                    p->string != NULL; p++)
    if(strcmp(string,p->string) == 0){
      p->count++;
      return 1;
    }
  if(p == table + tableSize)
    return -1;
  if((p->string = strdup(string)) == NULL)
    return -1;
  p->count = 1;
  return 0;
}
```

Memory Reallocation

`void *realloc(void *p, size_t size)`

- **`p`** must point to memory block obtained by previous call to **`mallloc`**, **`callloc`**, or **`realloc`**.
- **`size`** represents the new size of the block, which may be smaller or larger than the original size
- When block is expanded, bytes that are added to the block are not initialized
- If reallocation fails, null pointer is returned
- If it succeeds, pointer to expanded/shrunk block is returned
  - may not be in same location as original block!

Dynamic String Table

- Modify our singleton string table module to allow table to grow in size, using `realloc`.
- No change required in interface
  - Only `addStringTable` needs to be modified

```
int addStringTable(char *string){
  StringTabT *p;
  int newTableSize;

  for(p = table; p < table + tableSize && p->string != NULL; p++)
    if(strcmp(string,p->string) == 0){
      p->count++;
      return 1;
    }
  if(p == table + tableSize){
    newTableSize = 2*tableSize;
    if((p =
  realloc(table,newTableSize*sizeof(StringTabT)))==NULL)
      return -1;
    table = p;
    p = table+tableSize;
    tableSize = newTableSize;
  }
... /* as before */
}
```