Advanced Pointers

C Structures, Unions, Example Code
Review

- Introduction to C
- Functions and Macros
- Separate Compilation
- Arrays
- Strings
- Pointers
- Structs and Unions
Reminder

- You can’t use a pointer until it points to something
  - Default value is null
- Therefore, the following will give segmentation fault
  - `char * name;`
  - `strcpy(name,"bobby");`
  - `scanf("%s",name);`
  - `printf("%s\n",name);`
Pointers to Pointers

• Because a pointer is a variable type a pointer may point to another pointer
• Consider the following
  ▫ int age=42;
  ▫ int *pAge=&age;
  ▫ int **ppAge=&pAge;
• ppAge is a pointer variable type, but it points to the memory location of pAge, another pointer
int main ()
{
    /* a double, a pointer to double,  
    ** and a pointer to a pointer to a double */
    double gpa = 3.25, *pGpa, **ppGpa;
    /* make pgpa point to the gpa */
    pGpa = &gpa;
    /* make ppGpa point to pGpa (which points to gpa) */
    ppGpa = &pGpa;
    // what is the output from this printf statement?
    printf( "%0.2f, %0.2f, %0.2f", gpa, *pGpa, **ppGpa);
    return 0;
}

Output = 3.25, 3.25, 3.25
Pointers to Struct

/* define a struct for related student data */
typedef struct student {
    char name[50];
    char major [20];
    double gpa;
} STUDENT_t;

STUDENT_t bob = {"Bob Smith", "Math", 3.77};
STUDENT_t sally = {"Sally", "CSEE", 4.0};
STUDENT_t *pStudent; /* pStudent is a "pointer to struct student" */
/* make pStudent point to bob */
pStudent = &bob;
/* use -> to access the members */
printf ("Bob's name: %s\n", pStudent->name); // a->b is shorthand for (*a).b
printf ("Bob's gpa : %f\n", pStudent->gpa);
/* make pStudent point to sally */
pStudent = &sally;
printf ("Sally's name: %s\n", pStudent->name);
printf ("Sally's gpa: %f\n", pStudent->gpa);
Pointer in a Struct

- Data member of a struct can be any data type, including pointers
- Ex. Person has a pointer to struct name
  ```c
  #define FNSIZE 50
  #define LNSIZE 40
  typedef struct name
  {
    char first[ FNSIZE + 1 ];
    char last[ LNSIZE + 1 ];
  } NAME_t;
  typedef struct person
  {
    NAME_t *pName; // pointer to NAME struct
    int age;
    double gpa;
  } PERSON_t;
  ```
Pointer in a Struct

Given the declarations below, how do we access Bob’s name, last name, and first name?

- `NAME_t bobsName = {"Bob","Smith");`
- `PERSON_t t bob;`
- `bob.age = 42;`
- `bob.gpa = 3.4`
- `bob.pName = &bobsName;`
Self-Referencing Structs

- Powerful data structures can be created when a data member of a struct is a *pointer* to a struct of the same type

```c
typedef struct player
{
    char name[20];
    struct player *teammate; /* can’t use TEAMMATE yet */
} TEAMMATE;
TEAMMATE *team, bob, harry, john;
team = &bob; /* first player */
strncpy(bob.name, "bob", 20);
bob.teammate = &harry; /* next teammate */
strncpy(harry.name, "harry", 20);
harry.teammate = &john; /* next teammate */
strncpy(john.name, "bill", 20);
john.teammate = NULL; /* last teammate */
```
Self-Referencing Structs

• Typical code to print a (linked) list
  ▫ Follow the teammate pointers until NULL is encountered

// start with first player
TEammate *t = team; // t is now equal to &bob
// while there are more players...
while (t != NULL) {
  printf("%s\n", t->name);  // (*t).name
  // next player
  t = t->teammate;       //t=(*t).teammate;
}
Dynamic Memory

- C allows us to allocate memory in which to store data during program execution
- Dynamic memory has two primary applications:
  - Dynamically allocating an array
    - Based on some user input or file data
    - Better than guessing and defining the array size in our code since it can’t be changed
  - Dynamically allocating structs to hold data in some arrangement (a data structure)
    - Allows an “infinite” amount of data to be stored
Dynamic Memory Functions

- Part of the standard C library (`stdlib.h`)
  - `void *malloc(size_t nrBytes);`
    - Returns pointer to (uninitialized) dynamically allocated memory of size `nrBytes`, or NULL if request cannot be satisfied
  - `void *calloc(int nrElements, size_t nrBytes);`
    - Same as `malloc()` but memory is initialized to 0
    - Parameter list is different
  - `void *realloc(void*p, size_t nrBytes);`
    - Changes the size of the memory pointed to by `p` to `nrBytes`. The contents will be unchanged up to minimum of old and new size
    - If new size is larger, new space is uninitialized
    - Copies data to new location if necessary
    - If successful, pointer to new memory location is provided or NULL if cannot be satisfied
  - `void free(void *p)`
    - Deallocates memory pointed to by `p` which must point to memory previously allocated by calling by calling one of the above functions
void* and size_t

• The void* type is C’s generic pointer. It may point to any kind of variable, but may not be dereferenced

  ▫ Any other pointer type may be converted to void* and back again without any loss of information
  ▫ Void* is often used as parameter types to, and return types from, library functions

• size_t is an unsigned integral type that should be used (rather than int) when expressing “the size of something”

  ▫ E.g. an int, array, string, or struct
  ▫ Often used as parameter for library functions
malloc() for arrays

- malloc() returns a void pointer to uninitialized memory
- Good programming practice is to cast the void* to the appropriate pointer type
- Note the use of sizeof() for portable coding
- As we’ve seen, the pointer can be used as an array name
  ```c
  int *p = (int*)malloc(42*sizeof(int));
  for(k=0;k<42;k++)
      p[k] = k;
  for(k=0;k<42;k++)
      printf("%d\n",p[k]);
  ```
- p may be rewritten as a pointer rather than an array name
calloc() for arrays

- calloc() returns a void pointer to memory that is initialized to zero
- Note that the parameters to calloc() are different than the parameters for malloc()
  - int * p = (int*)calloc(42,sizeof(int));
  - for(k=0;k<42;k++);
  - printf(“%d\n”,p[k]);
- Try rewriting this code using p as a pointer rather than array name
realloc()

- realloc() changes the size of a dynamically allocated memory previously created by malloc() or calloc(), returns a void pointer to the new memory

```c
int *p = (int *)malloc( 42 * sizeof(int));
for (k = 0; k < 42; k++)
p[k] = k;
p = (int *)realloc( p, 99 * sizeof(int));
for (k = 0; k < 42; k++)
    printf( “p[ %d ] = %d\n”, k, p[k]);
for (k = 0; k < 99; k++)
p[k] = k * 2;
for(k=0; k < 99; k++)
    printf(“p[ %d ] = %d\n”, k, p[k]);
```
Testing the returned pointer

- `malloc()`, `calloc()`, and `realloc()` all return NULL if unable to fulfill the requested memory allocation.
- Good programming practice (i.e. points for your homework) dictates that the pointer returned should be validated.

```c
char *cp = malloc( 22 * sizeof( char ) );
if (cp == NULL) {
    fprintf( stderr, "malloc failed\n" );
    exit( -12 );
}
```
Assert()

- Since dynamic memory allocation shouldn’t fail unless there is a serious programming mistake, such failures are often fatal.
- Rather than using ‘if’ statements to check the return values from malloc() we can use the assert() function.
- To use assert():
  - `#include <assert.h>`
  - `char *cp = malloc(22*sizeof(char));`
  - `assert(cp!=NULL);`
How assert() works

- The parameter to assert is any Boolean expression -- `assert(expression);`
  - If the Boolean expression is true, nothing happens and execution continues on the next line
- If the Boolean expression is false, a message is output to stderr and your program terminates
  - The message includes the name of the .c file and the line number of the assert() that failed
- assert() may be disabled with the preprocessor directive `#define NDEBUG`
- assert() may be used for any condition including
  - Opening files
  - Function parameter checking (preconditions)
free()

- free() is used to return dynamically allocated memory back to the heap to be reused later by calls to malloc(), calloc(), or realloc()
- The parameter to free() must be a pointer previously returned by one of malloc(), calloc(), or realloc()
- Freeing a NULL pointer has no effect
- Failure to free memory is known as a “memory leak” and may lead to program crash when no more heap memory is available

```c
int *p = (int *)calloc(42, sizeof(int));
/* code that uses p */
free( p );
```
Dynamic Memory for Structs

typedef struct person{
    char name[ 51 ];
    int age;
    double gpa;
} PERSON;

/* memory allocation */
PERSON *pbob = (PERSON *)malloc(sizeof(PERSON));
pbob->age = 42; //same as (*pbob).age = 42;
pbob->gpa = 3.5; //same as (*pbob).gpa = 3.5;
strcpy( pbob->name, “bob”); //same as strcpy((*pbob).name, “bob”);

... /* explicitly freeing the memory */
free( pbob );
Dynamic Memory for Structs

Java Comparison

```java
public class Person {
    String name;
    public int age;
    public double gpa;
}

// memory allocation
Person bob = new Person();
bob.age = 42;
bob.gpa = 3.5;
bob.name = "bob"
// bob is eventually freed
// by garbage collector
```
Dynamic Teammates

typedef struct player{
    char name[20];
    struct player *teammate;
} PLAYER;

PLAYER *getPlayer() {
    char *name = askUserForPlayerName();
    PLAYER *p = (PLAYER *)malloc(sizeof(PLAYER));
    strncpy( p->name, name, 20 );
    p->teammate = NULL;
    return p;
}
Dynamic Teammates (2)

int main (){  
    int nrPlayers, count = 0;  
    PLAYER *pPlayer, *pTeam = NULL;  
    nrPlayers = askUserForNumberOfPlayers( );  
    while (count < nrPlayers){  
        pPlayer = getPlayer( );  
        pPlayer->teammate = pTeam;  
        pTeam = pPlayer;  
        ++count;  
    }  
    /* do other stuff with the PLAYERs */  
    /* Exercise --write code to free ALL the PLAYERs */  
    return 0;  
}
typedef struct player
{
    char name[20];
    struct player *nextteammate; /* can’t use TEAMMATE yet */
    struct player *prevteammate; /* can’t use TEAMMATE yet */
} TEAMMATE;
...
TEAMMATE *team, bob, harry, john;
team = &bob; /* first player */
strncpy(bob.name, “bob”, 20);
bob.nextteammate = &harry; /* next teammate */
bob.prevteammate = NULL; // or &john for circular
strncpy(harry.name, “harry”, 20);
harry.nextteammate = &john; /* next teammate */
harry.prevteammate = &bob;
strncpy(john.name, “john”, 20);
john.nextteammate = NULL; // &bob for circular linked list
john.prevteammate = &harry;
Dynamic Arrays

- As we noted, arrays cannot be returned from functions
- However, pointers to dynamically allocated arrays may be returned

```c
char *getCharArray( int size ){
    char *cp = (char *)malloc( size * sizeof(char));
    assert( cp != NULL);
    return cp;
}
```
Dynamic 2-D Arrays

- There are now three ways to define a 2-D array, depending on just how dynamic you want them to be.
  ```c
  int board[8][8];
  ```
- An 8 x 8 2-d array of int... Not dynamic at all
  ```c
  int *board[8];
  ```
- An array of 8 pointers to int. Each pointer represents a row whose size is be dynamically allocated.
  ```c
  int **board;
  ```
- A pointer to a pointer of ints. Both the number of rows and the size of each row are dynamically allocated.