C Language V—Dynamic Memory

CMSC 313
Sections 01, 02

Dynamic Memory Allocation

Dynamic Memory

- C allows us to allocate memory in which to store data during program execution.
- Like C++, dynamically allocated memory is take from the heap.
- 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 predetermined arrangement (a data structure)
    - Allows an "infinite" amount of data to be stored

Dynamic Memory Functions

These functions are used to allocate and free dynamically allocated heap memory and are part of the standard C library. To use these functions, include `<stdlib.h>`.

void *malloc(size_t nrBytes);

Returns a pointer to dynamically allocated memory on the heap of size `nrBytes`, or `NULL` if the request cannot be satisfied. The memory is uninitialized.

void *calloc(int nrElements, size_t nrBytes);

Same as `malloc( )`, but the memory is initialized to zero

Note that the 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 the minimum of the old and new size. If the new size is larger, the new space is uninitialized. Returns a pointer to the new memory, or `NULL` if request cannot be satisfied in which case `*p` is unchanged.

void free(void *p)

Deallocates the memory pointed to by `p` which must point to memory previously allocated by calling one of the functions above. Does nothing if `p` is `NULL`.

Adapted from Dennis Frey CMSC 313 Spring 2011

`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 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). It too is often used as a parameter to, or return type from, library functions. By definition, `size_t` is the type that is returned from the `sizeof( )` operator.

Adapted from Dennis Frey CMSC 313 Spring 2011

`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.

```
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 ];
```

Exercise: rewrite this code using `p` as a pointer rather than an array name

Adapted from Richard Chang, CMSC 313 Spring 2013

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]);
```

Adapted from Dennis Frey CMSC 313 Spring 2011

realloc( )

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

The contents will be unchanged up to the minimum of the old and new size. If the new size is larger, the new space is uninitialized.

```
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]);
```

Adapted from Dennis Frey CMSC 313 Spring 2011

Testing the returned pointer

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

```
• char *cp = malloc( 22 * sizeof(

  char ) ); if (cp == NULL) {
    •fprintf( stderr, "malloc failed\n);
   • exit( -12 );
• }
```

Adapted from Dennis Frey CMSC 313 Spring 2011

Testing the returned pointer

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

```
char *cp = malloc(22 * sizeof(char));

if (cp == NULL) {
      fprintf( stderr, "malloc failed\n");
      exit(-12);
}
```

Adapted from Richard Chang, CMSC 313 Spring 2013

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( )` macro.

To use `assert( )`, you must `#include <assert.h>`

```
char *cp = malloc( 22 * sizeof( char ) );
assert( cp != NULL );
```

Adapted from Dennis Frey CMSC 313 Spring 2011

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)

Adapted from Richard Chang, CMSC 313 Spring 2013

free( )

- `free( )` is used to return dynamically allocated memory back to the heap to be reused by later 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

```
int *p = (int *) calloc(42,  sizeof(int));

/* code that uses p */
free( p );
```

Adapted from Dennis Frey CMSC 313 Spring 2011

free( )

- `free( )` is used to return dynamically allocated memory back to the heap to be reused by later 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

```
int *p = (int *) calloc(42, sizeof(int));

/* code that uses p */
free( p );
```


Adapted from Richard Chang, CMSC 313 Spring 2013

Dynamic Memory for structs

In C++

```
class Person
{
    public:
    int age;
    double gpa;
};

// memory allocation
Person bob = new Person( );
bob->age = 42;
Bob->gpa = 3.5;

// bob is eventually freed
delete bob;
```

In C

```
typedef struct person
{
    int age;
    double gpa;
} PERSON ;

/* memory allocation */

PERSON *pbob
    = (PERSON *)malloc(sizeof(PERSON));
pbob->age = 42;
pbob->gpa = 3.5;
...

/* explicitly freeing the memory */

free( pbob );
```

Adapted from Dennis Frey CMSC 313 Spring 2011

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;
}
```

Adapted from Dennis Frey CMSC 313 Spring 2011

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 */

   return 0;
}
```

Adapted from Dennis Frey CMSC 313 Spring 2011

Dynamic Arrays

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

```
char *getCharArray( int size )
{
   char *cp = (char *)malloc( size * sizeof(char));
   assert( cp != NULL);

   return cp;
}
```

Adapted from Dennis Frey CMSC 313 Spring 2011

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.
- `int board[ 8 ] [ 8 ];`
- An 8 x 8 2-d array of int... Not dynamic at all
- `int *board[ 8 ];`
  - An array of 8 pointers to int. Each pointer represents a row whose size is be dynamically allocated.
- `int **board;`
  - A pointer to a pointer of ints. Both the number of rows and the size of each row are dynamically allocated.

Adapted from Dennis Frey CMSC 313 Spring 2011

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:

`int board[8][8];`
- An 8 x 8 2-d array of int... Not dynamic at all

`int *board[8];`
- An array of 8 pointers to int. Each pointer represents a row whose size is be dynamically allocated.

`int **board;`
- A pointer to a pointer of ints. Both the number of rows and the size of each row are dynamically allocated.

Adapted from Richard Chang, CMSC 313 Spring 2013

Perils & Pitfalls

Adapted from Richard Chang, CMSC 313 Spring 2013

Memory-Related Perils and Pitfalls

- Dereferencing bad pointers
- Reading uninitialized memory
- Overwriting memory
- Referencing nonexistent variables
- Freeing blocks multiple times
- Referencing freed blocks
- Failing to free blocks

Adapted from Dennis Frey CMSC 313 Spring 2011

Dereferencing Bad Pointers

The classic `scanf` bug.
Typically reported as an error by the compiler.

```
int val;

...

scanf("%d", val);
```

Adapted from Dennis Frey CMSC 313 Spring 2011

Reading Uninitialized Memory

Assuming that heap data is initialized to zero

```
/* return y = A times x */
int *matvec(int A[N][N], int x[N]) {
   int *y = malloc( N * sizeof(int));
   int i, j;

   for (i = 0; i < N; i++)
      for (j = 0; j<  N; j++)
         y[i] += A[i][j] * x[j];
   return y;
}
```

Adapted from Dennis Frey CMSC 313 Spring 2011

Overwriting Memory

Allocating the (possibly) wrong sized object

```
int i, **p;

p = malloc(N *sizeof(int));

for (i = 0; i < N; i++) {
   p[ i ] = malloc(M * sizeof(int));
}
```

Adapted from Dennis Frey CMSC 313 Spring 2011

Overwriting Memory

Not checking the max string size

```
char s[8];
int i;

gets(s);  /* reads "123456789" from stdin */
```

Modern attacks on Web servers
AOL/Microsoft IM war

Adapted from Dennis Frey CMSC 313 Spring 2011

Overwriting Memory

Misunderstanding pointer arithmetic

```
int *search(int *p, int val) {

   while (*p != NULL && *p != val)
      p += sizeof(int);

   return p;
}
```

Adapted from Dennis Frey CMSC 313 Spring 2011

Referencing Nonexistent Variables

Forgetting that local variables disappear when a function returns

```
int *foo () {
   int val;

   return &val;
}
```

Adapted from Dennis Frey CMSC 313 Spring 2011

Freeing Blocks Multiple Times

Nasty!

```
x = malloc(N * sizeof(int));
        <manipulate x>
free(x);

y = malloc( M * sizeof(int));
        <manipulate y>
free(x);
```

Adapted from Dennis Frey CMSC 313 Spring 2011

Referencing Freed Blocks

Evil!

```
x = malloc(N * sizeof(int));
  <manipulate x>
free(x);
  ...
y = malloc(M *sizeof(int));
for (i = 0; i < M; i++)
   y[ i ] = x[ i ]++;
```

Adapted from Dennis Frey CMSC 313 Spring 2011

Failing to Free Blocks (Memory Leaks)

Slow, long-term killer!

```
foo() {
   int *x = malloc(N * sizeof(int));
   ...
   return;
}
```

Adapted from Dennis Frey CMSC 313 Spring 2011

Failing to Free Blocks (Memory Leaks)

Freeing only part of a data structure

```
struct list {
   int val;
   struct list *next;
};

foo() {
   struct list *head = malloc(sizeof(struct list));
   head->val = 0;
   head->next = NULL;
   <create and manipulate the rest of the list>
   ...
   free(head);
   return;
}
```

Adapted from Dennis Frey CMSC 313 Spring 2011

Dealing With Memory Bugs

- Conventional debugger (`gdb`)
  - Good for finding bad pointer dereferences
  - Hard to detect the other memory bugs
- Some malloc implementations contain checking code
  - Linux glibc malloc: `setenv MALLOC_CHECK_ 2`

Adapted from Dennis Frey CMSC 313 Spring 2011

Dealing With Memory Bugs (cont.)

- Binary translator: valgrind (Linux)
  - Powerful debugging and analysis technique
  - Rewrites text section of executable object file
  - Can detect all errors as debugging `malloc`
  - Can also check each individual reference at runtime
    - Bad pointers
    - Overwriting
    - Referencing outside of allocated block
- Garbage collection (Boehm-Weiser Conservative GC)
  - Let the system free blocks instead of the programmer.

Adapted from Dennis Frey CMSC 313 Spring 2011