Applied Programming

Debugging Code

gdb

- **gdb** http://www.gnu.org/software/gdb/
- The **GNU debugger** is a general debugger. Reference: Guide to Faster, Less Frustrating Debugging

  http://heather.cs.ucdavis.edu/~matloff/UnixAndC/CLanguage/Debug.html

Notes:

- To use **gdb** you must **compile** with **–g**

  gcc –ansi –wall **–g** mycode.c

Basic commands

Command	Comment
gdb code	Start the gdb debugging the program "code"
gdb -args code var1 var2	Start the gdb debugging the program "code" passing the parameters var1 and var2
l	List out the source lines of "code"
b lineNum	Set a break point at the line number in the current file
b function	Set a break point at the start of the function specified, in any file.
r var1 var2	Run "code" as if var1 and var2 were specified on the command line. Code will run until the break point is hit
c	Continue from the current line to the next break point
n	Run the next line of code
print var	Print out the value of "var". Only simple types (int, float, strings, ptrs)
q	Quit the debugger
where	Shows the code line that caused a crash

Example 1

You REALLY need to learn GDB

Gdb -args TestDarray_hw short

Reading symbols from TestDarray_hw...done

(gdb) b 44

Breakpoint 1 at 0x400997: file TestDarray_hw.c, line 45.

(gdb) r

Breakpoint 1, main (argc=2, argv=0x7fffffffe358) at TestDarray_hw.c:45

45 int ErrorCode = 0; /* Application error code - 0 is OK */

(gdb) print ErrorCode

$1 = 0

(gdb) n

51 if (2 == argc) /* note that argc 2 means one argument given */

(gdb) q

See the GDB cheat sheet in the References on MyCourses

Example 2

Same as before but now I passed the parameters on the "run" line

gdb TestDarray_hw

Reading symbols from TestDarray_hw...done

(gdb) b 44

Breakpoint 1 at 0x400997: file TestDarray_hw.c, line 45.

(gdb) r american-English-short

Breakpoint 1, main (argc=2, argv=0x7fffffffe358) at TestDarray_hw.c:45

45 int ErrorCode = 0; /* Application error code - 0 is OK */

(gdb) print ErrorCode

$1 = 0

(gdb) n

51 if (2 == argc) /* note that argc 2 means one argument given */

(gdb) q

See the GDB cheat sheet in the References on MyCourses

Debugging: gdb

```
/* debug_ex01.c – using gdb
 * Example of simple segfaulting program */
#include <stdio.h>
int main()
  {
   int lcv=10;
  printf("%s is %d\n", lcv, lcv);
  return 0;
  }
```

- gcc –Wall –ansi –g debug_ex01.c –o dex01
  gdb ./dex01
- Commands: list, break 9, where, run

Segfault Example

gdb ./dex01

(gdb) r

Program received signal SIGSEGV, Segmentation fault.

0x0000003e57247e2c in _IO_vfprintf_internal (s=<value optimized out>,
format=<value optimized out>, ap=<value optimized out>) at vfprintf.c:1641

1641 process_string_arg (((struct printf_spec *) NULL));

(gdb) where

#0 0x0000003e57247e2c in _IO_vfprintf_internal (s=<value optimized out>,
format=<value optimized out>, ap=<value optimized out>) at vfprintf.c:1641

#1 0x0000003e5724f18a in __printf (format=<value optimized out>)
at printf.c:35

#2 0x00000000004004ed in main () at debug_ex01.c:9


Valgrind

Valgrind - reminder

- A **FAMILY of tools to detect *memory management bugs, threading bugs* and profile programs in detail.**
  - Valgrind.org

- **valgrind** tools (**-tool= the_tool**)
  - **memcheck** – checks memory usage
  - **cachegrind** – counts cache misses
  - **massif** – tracks overall heap usage

You always need to make sure that there are no memory leaks

Important: You must *compile with* the **–g** *option* (in **gcc**)

valgrind

- Valgrind can be used to find general programming errors:

```
/* valgrind_ex01.c  Warning: This program is wrong on purpose. */
#include <stdio.h>
int main() {
int age = 380;
int height;
printf("I am %d years old.\n");
printf("I am %d inches tall.\n", height);
return 0;   }
```

Used for general programming errors

- To do:

```
gcc -Wall -ansi -g valgrind_ex01.c -o vex01
valgrind ./vex01
```

Go through Warnings and address each of them in sequence. Cycle until program is bug free

valgrind

```
==12587== Command: ./val
I am -16776312 years old.
==12587== Use of uninitialized value of size 8
==12587==    at 0x3E57243A9B: _itoa_word (_itoa.c:195)
==12587==    by 0x3E57246652: vfprintf (vfprintf.c:1640)
==12587==    by 0x3E5724F189: printf (printf.c:35)
==12587==    by 0x4004FB: main (valgrind_ex01.c:24)
==12587==
==12587== Conditional jump or move depends on uninitialized value(s)
==12587==    at 0x3E57243AA5: _itoa_word (_itoa.c:195)
==12587==    by 0x3E57246652: vfprintf (vfprintf.c:1640)
==12587==    by 0x3E5724F189: printf (printf.c:35)
==12587==    by 0x4004FB: main (valgrind_ex01.c:24)
==12587==
==12587== Conditional jump or move depends on uninitialized value(s)
==12587==    at 0x3E572450E3: vfprintf (vfprintf.c:1640)
==12587==    by 0x3E5724F189: printf (printf.c:35)
==12587==    by 0x4004FB: main (valgrind_ex01.c:24)
==12587==
```

valgrind

```
==12587== Conditional jump or move depends on uninitialized value(s)
==12587==    at 0x3E57245101: vfprintf (vfprintf.c:1640)
==12587==    by 0x3E5724F189: printf (printf.c:35)
==12587==    by 0x4004FB: main (valgrind_ex01.c:24)
==12587==
I am 0 inches tall.
==12587==
==12587== HEAP SUMMARY:
==12587==     in use at exit: 0 bytes in 0 blocks
==12587==   total heap usage: 0 allocs, 0 frees, 0 bytes allocated
==12587==
==12587== All heap blocks were freed -- no leaks are possible
==12587==
==12587== For counts of detected and suppressed errors, rerun with: -v
==12587== Use --track-origins=yes to see where uninitialized values come from
==12587== ERROR SUMMARY: 4 errors from 4 contexts (suppressed: 6 from 6)
```

memory leaks with `valgrind`

- Tutorial: http://cs.ecs.baylor.edu/~donahoo/tools/valgrind/

```
valgrind --tool=memcheck --leak-check=yes ./leak
```

  - Does not trigger on out-of-bounds index errors for arrays on the stack

```
/* leak.c */
#include <stdio.h>
#include <stdlib.h>
int main() {
   char *p;
   p = (char *) malloc(19); /* Allocation #1 of 19 bytes, line 6*/
   p = (char *) malloc(12); /* Allocation #2 of 12 bytes, line 7*/
   free(p);                /* No null, using p again */
   p = (char *) malloc(16); /* Allocation #3 of 16 bytes, line 9*/
   return 0;
}
```

Important: You must *compile with* the **`-g`** *option* (in **`gcc`**)

valgrind --tool=memcheck --leak-check=yes ./leak

```
p = (char *)malloc(19); /*line 6*/
p = (char *)malloc(12); /*line 7*/
free(p);
p = (char *)malloc(16); /*line 9*/
```

```
Memcheck, a memory error detector
HEAP SUMMARY:
    in use at exit: 35 bytes in 2 blocks
   total heap usage: 3 allocs, 1 frees, 47 bytes allocated
 16 bytes in 1 blocks are definitely lost in loss record 1 of 2
    at 0x4A06A2E: malloc (vg_replace_malloc.c:270)
    by 0x40053D: main (leak.c:10)
 19 bytes in 1 blocks are definitely lost in loss record 2 of 2
    at 0x4A06A2E: malloc (vg_replace_malloc.c:270)
    by 0x400515: main (leak.c:7)
==14390== LEAK SUMMARY:
    definitely lost: 35 bytes in 2 blocks
    indirectly lost: 0 bytes in 0 blocks
      possibly lost: 0 bytes in 0 blocks
    still reachable: 0 bytes in 0 blocks
         suppressed: 0 bytes in 0 blocks
 For counts of detected and suppressed errors, rerun with: -v
 ERROR SUMMARY: 2 errors from 2 contexts (suppressed: 6 from 6)
```

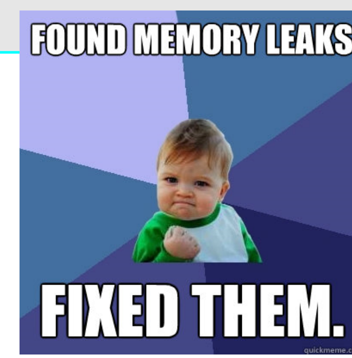
Leaks Fixed!

- All code going forward must be leak free!

```
Memcheck, a memory error detector
 Copyright (C) 2002-2012, and GNU (
 Using Valgrind-3.8.1 and LibVEX;
info
 Command: ./noleak
 HEAP SUMMARY:
     in use at exit: 0 bytes in 0 blocks
   total heap usage: 3 allocs, 3 frees, 47 bytes allocated
 All heap blocks were freed -- no leaks are possible

 For counts of detected and suppressed errors, rerun with: -v
 ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 6 from 6)
```

```
p = (char *)malloc(19); /*line 6*/
free(p);
p = (char *)malloc(12); /*line 7*/
free(p);
p = (char *)malloc(16); /*line 9*/
Free(p);
```



heap usage with `valgrind`

For more details about **massif** consult its manual:

http://valgrind.org/docs/manual/ms-manual.html#ms-manual.running-massif

- Compile code with **–g** option
- Invoke **valgrind** with the **massif** **(heap profiler)** tool:

```
valgrind --tool=massif ./TestDarray
```

Result: files with names **massif.out.####** (some number)

- Examine results of using **ms_print**

```
ms_print massif.out.23721 | more
```

  - **ms_print** will produce:
    - A ***graph*** of program use of memory during execution
    - Details about allocation at various points in the execution

```
Tip: Massif uses "instructions executed" as
     unit of time (see option --time-unit)
```

Partial valgrind example

The program dynamically reads 206,590 words into a 256 byte long string.
Expected memory usage = 50.4 MB

```
valgrind --tool=massif ./TestDarray
ms_print massif.out.23721 | more
```

```
    MB
51.23^                                                                       #
     |                                                                    :::#
     |                                                                :@:::::#
     |                                                             ::::@:::::#
     |                                                          :@:::::@:::::#
     |                                                      @::::@:::::@:::::#
     |                                                  ::::@::::@:::::@:::::#
     |                                               ::@:: :@::::@:::::@:::::#
     |                                           ::::::@:: :@::::@:::::@:::::#
     |                                        @::: ::::@:: :@::::@:::::@:::::#
     |                                    ::::@::: ::::@:: :@::::@:::::@:::::#
     |                                 ::@:: :@::: ::::@:: :@::::@:::::@:::::#
     |                             ::::: @:: :@::: ::::@:: :@::::@:::::@:::::#
     |                         :::::: :: @:: :@::: ::::@:: :@::::@:::::@:::::#
     |                     ::::: :::: :: @:: :@::: ::::@:: :@::::@:::::@:::::#
     |                  :::::::: :::: :: @:: :@::: ::::@:: :@::::@:::::@:::::#
     |               :::: :::::: :::: :: @:: :@::: ::::@:: :@::::@:::::@:::::#
     |          :@@@::::: :::::: :::: :: @:: :@::: ::::@:: :@::::@:::::@:::::#
     |       ::@:@ @::::: :::::: :::: :: @:: :@::: ::::@:: :@::::@:::::@:::::#
     |    :@@: @:@ @::::: :::::: :::: :: @:: :@::: ::::@:: :@::::@:::::@:::::#
   0 +----------------------------------------------------------------------->
```

Applied Programming

Dynamic Arrays

Dynamic Memory Allocation in C

- Java (Python) have built in garbage collection C does not.
- The C programmer is responsible for allocation and de-allocation of (heap) memory.
- The primary memory ***allocation/deallocation functions*** are:
  - **`malloc()`, `calloc()`, `realloc()`**
  - **`free()`** (used to free memory back to the heap).

When programming in C, I promise that I will free all the heap memory that I use

Malloc()

Primary function for memory allocation

- Prototype:

```
void * malloc (size_t size);
```

  - Returns **NULL** (pointer) if unsuccessful

- Example: Allocate memory for 20 floats

```
float* Numbers;                    /* pointer variable */
Numbers = ( float* ) malloc(20 * sizeof(float));
```

  - Note typecasting of generic pointer `void`
  - Allocated memory is NOT automatically ZERO “0”

Tip: Always check if your allocation request was granted (e.g., check that Numbers!=NULL is true)

Calloc()

Sets memory to “binary zero”

- Prototype:

```
void* calloc (size_t nelements, size_t elementSize);
```

  - Used to ***allocate and initialize*** the elements to 0 simultaneously
  - Returns **NULL** (pointer) if unsuccessful

Example:

  - Create an array of 10 integers initialized to 0

```
int* ptr;
ptr = ( int* ) calloc(10, sizeof(int));
```

  - Note: don’t forget to check that your request was granted.

Realloc()

Resizes and copies memory

- Prototype: `void* realloc (void* pointer, size_t size);`
  - Used to ***grow or shrink*** a block of memory
  - "old" pointer is no longer valid on success
  - Returns NULL (pointer) if unsuccessful
    - Old pointer still good if unsuccessful

Example: Extend the array to hold 20 integers

```
int* ptr;
ptr = ( int* ) calloc(10, sizeof(int));
. . .
/* oops need larger array :-) */
ptr = ( int* ) realloc(ptr, 20*sizeof(int));
```

Free()

Return allocated memory back to the heap

- Prototype: `void free (void* pointer);`
    - <u>Note that this function does not return anything</u>

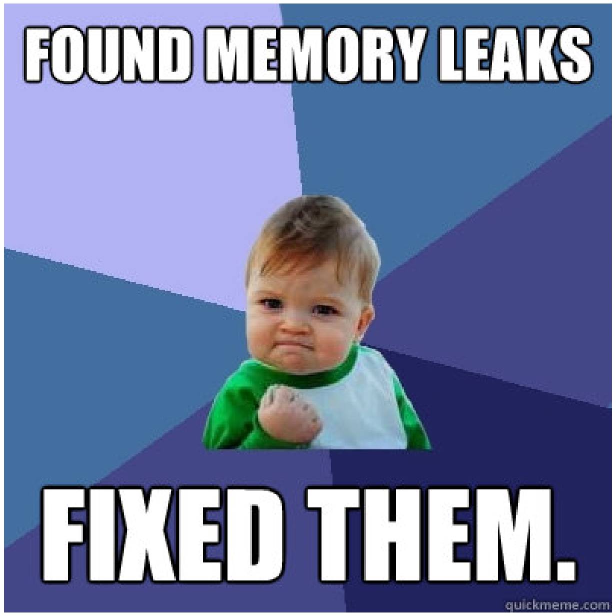
Example:

```
float* Numbers;                          /* pointer variable */
Numbers = ( float* ) malloc(20 * sizeof(float));
. . .
/* done using Numbers, now return memory to heap */
free (Numbers);
Numbers = NULL;                          /* Good programming */
```

Tip: After **`free`** the pointer **`Numbers`** becomes ***dangling pointer.*** To avoid using it unintendedly set it to point to nothing, e.g.,
`Numbers = NULL;`

Memory Leaks

- When you malloc() memory, you reserved a block of memory that will NOT be reused by ANYONE until you free() it!
  - A memory leak occurs when you reserve memory and never return it.
  - Causes your program memory size to grow until your program, or another program, crashes.
  - Easy to cause, harder to fix



Memcpy()

Copies “n” bytes from srt to dest

- Prototype:

```
void *memcpy(void *dest, void *src, size_t n);
```

  - Note that this function returns a pointer to dest

Example:

```
char *src = “Some bytes”;     /* pointer variable */
char dest [80];

  memcpy(dst, src, sizeo(dest));
```

Tip: You will want to use this function to copy data structures in the home work.

Static vs. Dynamic arrays

- ***Static Arrays*** are defined at compile time and whose ***size cannot be changed during execution .***   e.g. int array [20];
  - Memory space is allocated in the stack or in the heap, depending on where the array is defined.
- ***Simple Dynamic Arrays*** are allocated once and ***never change size*** during program execution
- ***(Fully) Dynamic Arrays*** are ***can grow/shrink in size "on demand"*** during program execution

sizeof()

- Sizeof ONLY returns compile time sizes. It can't "see" the real size of dynamic arrays!

```
int array1 [100];
int *array2;
array2 = (int *)malloc(100);
```

- sizeof(array1) is 400
  sizeof(array2) is 8 -the size of any POINTER

- You can't always get the size of a pointer from a pointer ☹

sizeof() from Pointers

- If you are REALLY careful you CAN get the size from SOME pointer variables

  struct test { int x; int y; int z;};

  struct test *ptr;

  sizeof(**struct test**) is 12

  sizeof(**(*ptr)**) is 12

- Why doesn't it work in the previous case?
  - Because we didn't dereference the pointer
  - But if we did, we still get the wrong answer, because we really wanted the array SIZE not the base object size

Static Array Failures

- What are the sizes of array1 & array2?

```
int array1 [100];
int num = 100;
int array2 [num];
```

- array1 is 400
  array2 is 4 -WHAT???
  a) C can't dynamically allocate memory this way!
  b) Some C compilers will generate an error, most just GIVE YOU ONE ENTRY!

Simple Dynamic Array

```
   /* Example: simple dynamic array of 100 doubles   simpleDyn.c */
       #include <stdlib.h>
       #define N (100)        /* Size of array          */
int main(void)
{
       int i;                    /* Index for access       */
       double* array;         /* Pointer to array "head" */

       /* Allocate memory for static array from heap     */
       array = (double *) malloc (N * sizeof (double));

       /* Initialize static array by indexing  */
       for (i=0; i<N; i++) {array [i] = i;}

       /* Free memory used, return back to heap */
       free (array);
       array = NULL; /* defensive programming, ground it */
   return(0);
}
```

- What's wrong with this program?

Dynamic Arrays

- A ***(Fully) Dynamic Array*** is an array that ***can grow in size "on demand"*** during program execution
- This "*data structure*" is useful when:
  1. Amount of *data cannot be determined a priory* (e.g., before running the program)
  2. Data is ***accessed sequentially*** or by an index
  3. Access time variability must be minimized
  4. Data will *not be searched (ordering unimportant)*
- To implement dynamic arrays we need
  - Code to manage the data structure
  - Provide access functions to data (via pointers)
  - Keep track of "state" of array (array header)

Homework Hint 1

- Most assignments will use data structures of the form:

```
typedef {
  int num;       /* The number of things */
  Data* data_p; /* Pointer to storage for the things */
} Thing;
```

- Note: "Thing" does not store anything, it is just an container variable.
  - It holds the thing that holds the data
  - data_p (after you malloc) really holds the "Thing"

Homework Hint 2

```
typedef {
  int num;       /* The number of things */
  Data* data_p; /* Pointer to storage for the things */
} Thing;
```

- Generally your code should look like:

```
Thing *Mything_p;
Mything_p = malloc(sizeof(Thing));   /* the container */

/* Now make space for the actual data */
Mything_p->num = 99;
Mything_p->data_p = malloc(sizeof(Data)* Mything_p->num);
                              ……
Free(Mything_p->data_p);                 /* In this order */
Free(Mything_p);
```

Good Example

- What does this do?
- Main.c

```
Thing *createThing(Thing *p, int num);  /* Prototype */
Thing mainThing;
createThing(&mainThing, 99);
```

- What does mainThing contain?

- Thing *createThing(Thing *p, int num){

```
p->num = num;
p->data_p = malloc(sizeof(Data)* num);
Return(p);
}
```

Bad Example

- Create a Thing
- Main.c

```
Thing *createThing(Thing *p, int num); /* Prototype */
Thing *mainThing_p;
createThing(mainThing_p, 99);
```

- What does mainThing contain?

- Thing *createThing(Thing *p, int num){

```
p->num = num;
p->data_p = malloc(sizeof(Data)* num);
Return(p);
}
```

Good? Example

- Create a Thing
- Main.c

```
Thing *createThing(Thing *p, int num); /* Prototype */
Thing *mainThing_p;
mainThing_p = createThing(mainThing_p, 99);
```

- What does mainThing contain?

- Thing *createThing(Thing *p, int num){

```
p = malloc(sizeof(Thing));     /* just destroyed p */
p->num = num;
p->data_p = malloc(sizeof(Data)* num);
Return(p);
}
```

Dynamic Arrays: Header Structure

- The data structure, **`DynamicArray`** is used to implement a *generic dynamic array*

```
typedef {
  unsigned int Capacity;
  unsigned int EntriesUsed;
  Data* Payload;
} DynamicArray;
```

- **`Capacity`** is used to determine if we run out of space and we need extra memory to be allocated.
- **`EntriesUsed`** stores the index of the *last element* added.
- **`Payload`** is a ***pointer to the array that holds the data***. The data type of the array is determined by the typemark **`Data`** (*e.g.*, if we have an array of doubles then we would use **`typedef double Data`**)

Dynamic Arrays: Management

- Now we need to implement functions to "manage" (initialize, update, etc.) the dynamic array.

- Example: Pseudo-code to Create and Initialize a Dynamic Array

```
CreateDynamicArray
   Input: DynamicArray header, InitialSize
   Initialize dynamic array as empty:
      Set EntriesUsed to zero, and
      Set storage Capacity to a InitialSize
   if Capacity is non-zero then
      Allocate heap memory for Data array
      Set Payload pointer to address of Data array
   else
      Set Payload pointer to NULL
   endif
```

Dynamic Arrays: Growth

- The key function of a dynamic array is the one that ***manages the size of the array as data is added to it***. Here is one possible algorithm

- Example: Pseudo-code to add data to Dynamic Array,

```
PushToDynamicArray
Input: DynamicArray header, Payload pointer
Return: Index of last element inserted
if DynamicArray is full
   Determine new size for the array (based on growth policy)
   Set array Capacity to the new size
   Allocate [realloc()] memory storage in the heap
end
Insert new data element at end of the array [memcpy()]
Increment EntriesUsed by one
```

Dynamic Arrays: Growth Policy

- The growth policy determines ***how to increase the size of the array*** if we run out of space
- The most ***common growth policies*** are:
  - By a fixed value (simplest of all)
    - Add a constant number of additional entries
  - By a fixed percentage of its current size
    - **`new size = 1.x * old size`**
    - Double the size of the array (efficient and simple)
- The “best growth policy” depends on the application.

Reallocating Memory

- Our objective is to "grow" the array dynamically using ***contiguous chunks of memory*** (in the heap)
- In practice this may not always be possible, the (heap) memory may already be in use.
- In general we may need to do the following:
  - *Allocate new* (contiguous) chunk of memory from the heap. (**`malloc`**)
  - *Copy data* from existing (smaller) array to new array (**`memcpy`**)
  - *Release memory* space of old array back into heap (**`free`**)
- **`realloc`** does the three steps above for you
  - found in **`<stdlib.h>`**

Dynamic Arrays: Prototypes

- To implement this abstract data type in C you will create a module.
- You will place the prototypes for the array management functions in the header
- Example:

```
/* These function prototypes should be placed in a *
 * module header file                             */
void CreateDynamicArray
  (DynamicArray* DArrayPtr, unsigned int InitialSize);
unsigned int PushToDynamicArray
  (DynamicArray* DArrayPtr, Data* PayloadPtr);
void DestroyDynamicArray
  (DynamicArray* DArrayPtr);
```

- *Reference:* A similar concept called "infinite arrays" is given in chapter 18 of Oualline, Practical C programming book.

Example:Dynamic Arrays

- Reads in list of words, without knowing the size ahead of time.

```
TestDynamicArray  american-english-words
First 5 elements:
      1, dermatitis
      2, toxins
      3, wisted
      4, benedictions
      5, Tera
      6, petrochemistry's
Last 5 elements:
 206585, harken
 206586, Pict's
 206587, Sidman's
 206588, intercohort
 206589, pressurize
 206590, besotting
Number of Words Read = 206590
```

Example: Dynamic Array Module

- C Module for Dynamic Arrays

  `DynamicArrays.h` (public interface)
  `DynamicArrays.c` (implementation)

- Note: you will have a chance to implement a dynamic array module in homework #2

Dynamic Arrays: Header

Example of header file `DynamicArrays.h` (all comments removed)

```
#ifndef _DYNAMIC_ARRAYS_H_
#define _DYNAMIC_ARRAYS_H_
typedef struct Data {
   int Num;           /* Sequence number */
   char String[256];  /* word, (less than 255 chars -not checked!) */
  } Data;

#define GROWTH_AMOUNT (100)
typedef struct Darray {
   unsigned int Capacity;     /* Max Number of elements array can hold */
   unsigned int EntriesUsed;  /* Number of array elements "used"       */
   Data *Payload;     /* Pointer to array that actually holds the data  */
  } DArray;

void CreateDArray(DArray *DArrayPtr, unsigned int InitialSize);

unsigned int PushToDArray(
          DArray *DArrayPtr, Data *PayloadPtr);

void DestroyDArray(DArray *DArrayPtr);

#endif /* _DYNAMIC_ARRAYS_H_ */
```

Example 1

- Given: **int main(int argv, char * argv[])** and the compiler message::

  hw5.c: In function ‘main’:

  hw5.c:89: **warning: control reaches end of non-void function**
- What is wrong?
- Functions with a declared return **type MUST ALWAYS return a value** (e.g. return(4)).
  Not ALL paths in this code ARE returning values.
  In some paths junk (random values) will be returned!
  ALWAYS fix all warning!