CS2002
Computer Systems
Lecture 7 – 8

Dynamic Memory Allocation

Jon Lewis (JC 0.26)

School of Computer Science

University of St Andrews

Overview

- Storage Classes
  - Storage classes for variables
    - static
    - automatic
- Dynamic storage (not storage class for variables)
  - for data, accessed via pointers held in variables
- Dynamic Allocation
  - malloc, free
  - Creating equivalent to variable size arrays
  - Creating other dynamic objects

So-called "storage class"

Variables are classified according to their behaviour with regard to …

scope	intrafile: local, global
linkage	interfile: extern, static
duration	storage allocation: static, automatic

Static Allocation

Memory ***statically*** allocated when program is run

- the amount is decided at time of writing the program or at the latest at compile time
- can be used for variables local to a block or external to all blocks
- so scope can be local to block/function, entire file, or global
- in either case they retain their value across exit from and re-entry to blocks and functions, i.e. they persist for lifetime of program
- automatically zeroed
- default for global variables, defined outside all blocks, without having used static specifier
- global variables declared with static specifier are local (private) to file or translation unit
- within block or function need static specifier to make variable static
- global variables declared extern signify that they are statically defined elsewhere – external linkage

Automatic (‘stack’) Allocation

- Memory automatically allocated when function is executed.
- fixed size of memory automatically allocated at run-time.
  - used for variables that are local to a block
  - scope is the block within which the variable is declared.
  - size of memory is known at compile time (apart from arrays such as `int array[n]`)
  - automatic variables are discarded on exit from the block, i.e. they only persist for lifetime of the block.

Dynamic (‘heap’) Allocation

- Memory allocated at run-time.
  - Done manually by the programmer.
  - Lifetime of the memory is handled by the programmer.
- `malloc` (and friends) allocate a new block of raw memory.
  - Memory does not have a type, you can put anything you like in it.
- `free` returns a block of memory to the OS.
  - There is no automatic memory management!
    - Memory can be leaked if the program "loses" it.
    - Undefined behaviour if you read or write from a pointer into a block of already freed memory.

malloc/free example

```
int* new_int(int i) {
  int* ptr = malloc(sizeof(int));
  *ptr = i;
  return ptr;
}

int add_ints() {
  int* i = new_int(2);
  int* j = new_int(3);
  int z = *i + *j;
  free(i); free(j);
  return z;
}
```

malloc (2)

- malloc returns NULL if there is insufficient memory left in the OS.

- Memory returned has no type, you can use it for whatever you like, or reuse it for different things.

- Memory will be filled with random data, just like automatically allocated memory.

- You have to remember the size of an allocated block yourself.

Malloc Return Value

- It is good practice to check the return value of malloc (and related functions).

- If nothing else, consider:

```
int* ptr = malloc(sizeof(int));
if(!ptr) abort(); // or if(ptr == NULL)
```

- Can even put this into a method.

Malloc Example (2)

Use malloc to allocate a block of memory of a certain size

```
#include <stdlib.h>
// Above #include gives you malloc as follows:
void *malloc(size_t size);
```

The size is a number of bytes

```
/* Create a new "permanent" copy of string src */
// Defined in <string.h>
char *strdup(const char *src) {
  char *copy = malloc( strlen(src)+1 );
  strcpy(copy, src);
  return copy;
}
```

For the '\0'

Releasing Dynamic Memory

- Memory is automatically freed when your program finishes, but it's a good habit to clean up after yourself.

- `free` accepts a pointer given by malloc, and frees the memory. Make sure to only call it once per **`malloc`**ed memory block!

- Remember:
  - You **`malloc`** it, you **`free`** it. You didn't, don't.

Releasing Dynamic Memory

- programmer must explicitly release memory

```
 #include <stdlib.h>
// Above #include also provides free() as follows
 void free(void *ptr);
```

- is always good practice to release memory, or else you have memory leak
- Do not call free on any pointer values except ones you got from malloc.

```
int i = 1;
int* j = malloc(20); // is 20 enough?
int* k = j + 1;
free(&i);     // BOOM!
free(k);      // BOOM!
free(k - 1); // O.K.
```

Malloc and Arrays

- We have seen before that arrays are just stored as a list of objects, contiguous in memory.
- If we want to create an array, we just make some memory for it!

```
float* make_float_array(int n) {
  int i;
  float* a = malloc(sizeof(float) * n);
  for(i = 0; i < n; ++i)
    a[i] = i;
  return a;
}
```

Structs and malloc

- Allocating space for structs is the same as for anything else.

```
typedef struct { int x,y; } Point;

Point* new_Point(int x, int y) {
  Point* this = malloc(sizeof(Point));
  // OR above could be written as
  // Point *this;
  // this = malloc(sizeof(Point));
  this->x = x;
  this->y = y;
  return this;
}
```

Allocating dynamic structs

```
typedef struct Person {
        char *name;          // Pointer to start of name string
        int age;
} Person;


Person *new_Person(const char *name, int age) {
    Person *this = malloc(sizeof(Person));
    this->name = strdup(name); // uses malloc internally
    this->age = age;
    return this;
}

void Person_free(Person *this) {
    free(this->name);
    free(this);
}
```

Allocating dynamic structs

```
void Person_setAge(Person *this, int age) {
    this->age = age;
}

int Person_getAge(Person *this) {
    return this->age;
}

void Person_printDetails(Person *this) {
    printf("Person (name = %s, age = %d)\n", this->name, this->age);
}
```

Allocating dynamic structs

```
Person *p1 = new_Person("Jeff Jones", 23);
Person *p2 = new_Person("Jon Lewis", 32);
Person_printDetails(p1);
Person_printDetails(p2);

Person_setAge(p1, 101);
Person_printDetails(p1);

Person_free(p2);
p2 = p1;

Person_printDetails(p1);
Person_printDetails(p2);

Person_free(p1);
```

EXAMPLES

Examples/L07-08/PersonStructDynamic

Examples/L07-08/

other struct examples (with functions, …)

Other Memory Functions

- calloc provides a simple wrapper around malloc.

```
void* calloc(size_t num, size_t size);
```

does `malloc(num*size)`, then fill with zeroes.

- Intended to be used like:

```
int* p = calloc(10, sizeof(int));
// All ints set to zero.
```

- But there is no requirement, it just gives you back num*size bytes to do with as you will.

System independent pointer comparison.

- Converting a pointer, or the difference of two pointers, to an int can cause truncation if
  `sizeof(int) < sizeof(int*)`.
- which is the case for 64 bit
- The compiler has a type called **`size_t`**, which is an unsigned type big enough to fit a pointer (64 bits on the labs)

- Also, **`ptrdiff_t`**, which holds the difference between two pointers.

- They both live in **`<stddef.h>`**.
- **`"%zu"`** - printf a **`size_t`**. - **`"z"`** = sizeof(size_t), **`"u"`** = unsigned
- **`"%td"`** - printf a **`ptrdiff_t`**. - **`"t"`** = sizeof(ptrdiff_t)

2D arrays in C

- Actual 2D arrays are a bit strange, of limited use, and need care:

```
int a[3][3] = {{1,2,3},{4,5,6},{7,8,9}};
a[1][2] = -1;

void f(int x[3][3]); // Pass array to function
```

2D arrays

- **`char array[2][3] = {{3,4,5},{6,7,8}}`**
- Remember: array of T acts like a pointer to T.
- So array of arrays acts like a pointer to an array.
- So in this case, **`array[1]`** points to **`array[1][0]`**

9000	9001	9002	9003	9004	9005
3	4	5	6	7	8

array+1 starts at 9003

2D arrays

- An actual 2D array is an array of arrays.
  - this is not the same as a pointer to pointer!

- Passing 2-D arrays to functions is either of limited use or a little awkward
  - Contiguous storage behaves like 1-D array
    - so need to know array dimensions to access elements

Passing 2-D Array to Function

```
void alterArray(int array[3][5], int row, int column, int val) {
   array[row][column] = val;
}

void alterArrayByPntr(int *array, int columns, int row, int column, int val) {
   array[row*columns + column] = val;
}

int main(void) {
  int array[3][5] = {{1, 2, 3, 4, 5}, {6, 7, 8, 9, 10}, {11, 12, 13, 14, 15}};
  int* p2array = (int *) array;

  alterArray(array, 2, 0, -1);
  alterArrayByPntr(p2array, 5, 2, 0, -1);

  return 0;
}
```

- Both alter Array calls will alter the value 11 at index [2][0] to -1
- alterArray is of limited use (as array dimensions is included in function decl)
- alterArrayByPntr overcomes that issue, but still needs to know column size (5 in this case) and is a little awkward

Better to have pointer to pointer

```
char x[3] = {1,2,3};
char y[4] = {4,5};
char* a[2] = {x,y}; // a decays to char**
```

y:

0x466	0x467	…
4	5	

x:

0x100	0x101	0x102
1	2	3

a:

0x502	0x503
0x100	0x466

Pointer to Pointer

```
int x[3] = {1,2,3};
int y[4] = {4,5};
int* a[2] = {x,y};
```

a[0] equal **x;**

a[0][1] equal **x[1]** equal **2;**

a[1][0] equal **y[0]** equal **4;**

Pointer to Pointer

- Can be accessed as if they were 2-D arrays
- Can use dynamic memory allocation
- Storage no longer contiguous
  - indirection cost on access
- Easier to pass to functions
- Example
  - malloc_arrays.c
  - how to dynamically allocate a proper pointer to pointer which can be accessed like a 2-D array