Advanced arrays and pointers

Stat 580: Statistical Computing

- Theme: Black - White
- Printable version

References

- Part of this slide set is based on *Essential C* by Nick Parlante:

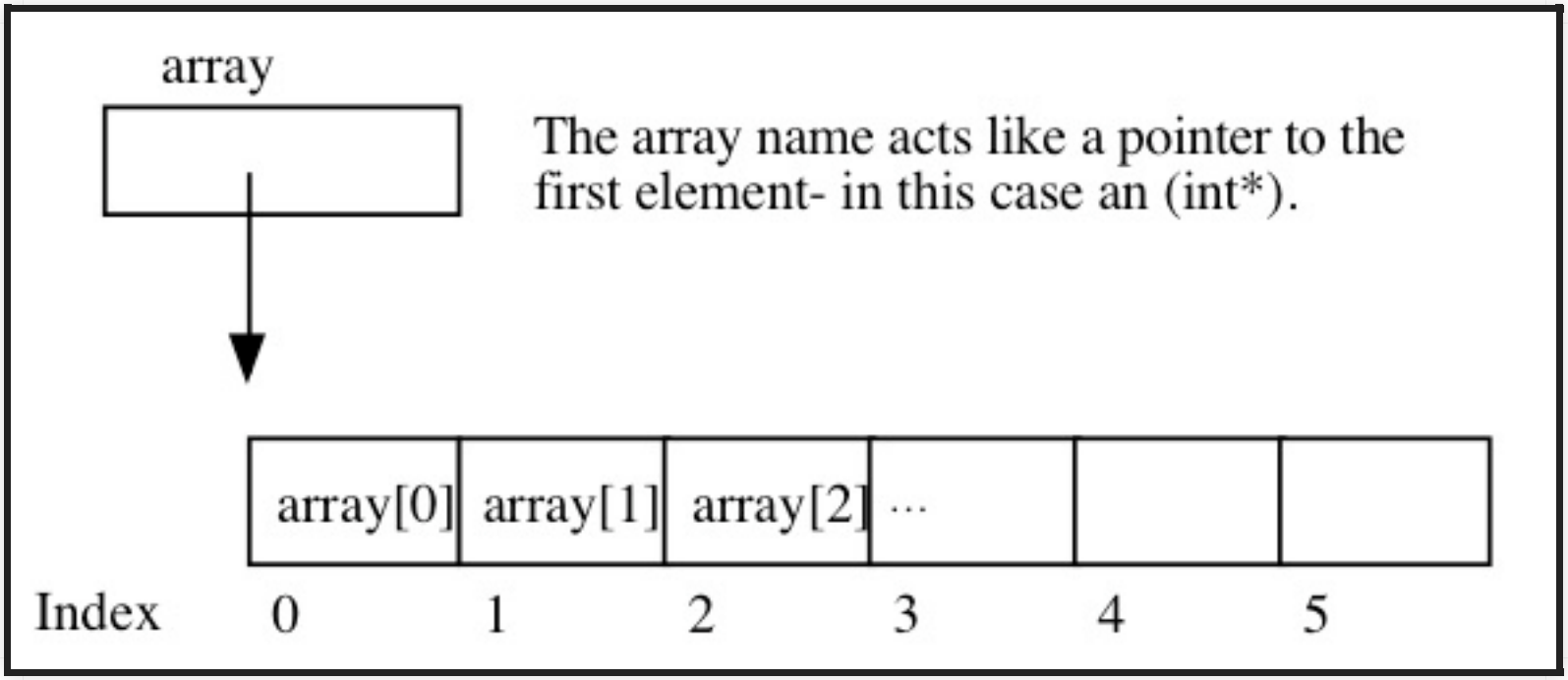
(for copyright reason, this notice is reproduced here.)

Advanced C arrays

In C

- an array is formed by laying out all the elements contiguously in memory
- the square bracket syntax can be used to refer to the elements in the array
  - `x[0]` refers to the first element of `x`
  - `x[3]` refers to the fourth element of `x`
- the array itself is referred to as the address of the first element
  - known as the "base address" of the whole array
  - `x` is an array, and acts like a pointer to `x[0]`
  - if you pass `x` to other function, you are passing the pointer (achieving "pass by reference")

Advanced C arrays



What [ ] does

> *address of n-th element = address of 0-th element + (n * element size in bytes)*

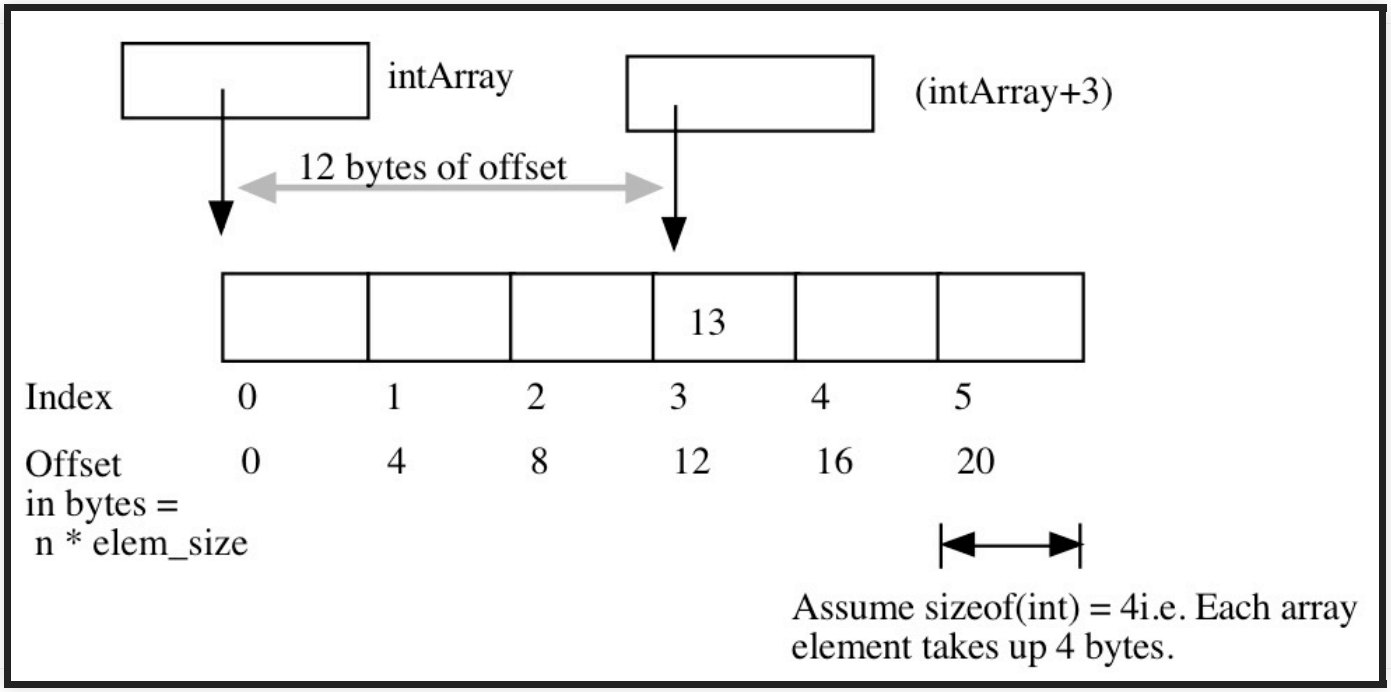
The square bracket syntax [ ] deals with this address arithmetic:

1. takes the integer index,
2. multiplies by the element size,
3. adds the resulting offset to the array base address,
4. finally dereferences the resulting pointer to get to the desired element.

Pointer arithmetic

A + between a pointer and an integer does the same offset computation, but leaves the result as a pointer

- ++ syntax still works



Pointer arithmetic

Any `[ ]` expression can be written with the + syntax instead

- need the pointer dereference
- `intArray[3]` is exactly equivalent to `*(intArray + 3)`.

```
int a[5]={1,2,3,4,5}; /* a is in fact the base address */
int i;

for (i=0; i<5; i++){
  printf("a[%d] = %d and *(a + %d) = %d and a + %d = %p \n",
         i, a[i], i, *(a+i), i, (a+i));
}
```

Example

```c
#include<stdio.h>

int product(int *a, int n);

int main() {
  int a[5]={2,3,4,5,6}; /* a is in fact the base address */
  int *p;

  /* iterating the address */

  for (p=a; p<&a[5]; p++){ /* note a[4] is the end of the array */
    printf("%d\n", *p);
  }

  printf("product of a is %d\n", product(a, 5));

  return 0;
}

int product(int *a, int n) {
  int i, out = 1;
  for (i=0; i<n; i++){
    out *= a[i];
  }
  return out;
}
```

Example

```c
#include<stdio.h>

int product(int *a, int n);

int main() {
  int a[5]={2,3,4,5,6}; /* a is in fact the base address */
  int *p;

  /* iterating the address */

  for (p=a; p<&a[5]; p++){ /* note a[4] is the end of the array */
    printf("%d\n", *p);
  }

  printf("product of a is %d\n", product(a, 5));
  printf("product of a[1],a[2],a[3] is %d\n", product(&a[1], 3)); /* 3*4*5 */
  return 0;
}

int product(int *a, int n) {
  int i, out = 1;
  for (i=0; i<n; i++){
    out *= a[i];
  }
  return out;
}
```

Four versions of string copy function

```
void strcpy1(char *dest, char *source) {
  int i = 0;

  while (1) {
    dest[i] = source[i];
    if (dest[i] == '\0') break;  /* we're done */
    i++;
  }
}


/* Move the assignment into the test */
void strcpy2(char *dest, char *source) {
  int i = 0;

  while ((dest[i] = source[i]) != '\0') {
    i++;
  }
}
```

Four versions of string copy function

```
/* Get rid of i and just move the pointers. */
/* Relies on the precedence of * and ++.    */
void strcpy3(char *dest, char *source)
{
  while ((*dest++ = *source++) != '\0') ;
}


/* Rely on the fact that the integer representation of '\0' is 0 */
/* (equivalent to FALSE)                                         */
void strcpy4(char *dest, char *source)
{
  while ((*dest++ = *source++)) ;
}
```

Pointer type effects

```
int *p;
p = p + 12;     /* at run-time, what does this add to p? 12? */
```

- The code above increments `p` by 12 ints (each int probably takes e.g. 4 bytes)
- The compiler figures all this out based on the type of the pointer.
- You can change the size of increments using casts.
  - `p = (int*) ( (char*)p + 12);`
  - This increments `p` by exactly 12 bytes since `sizeof(char)` is always 1.

Arrays and pointers

The compiler does not distinguish meaningfully between arrays and pointers

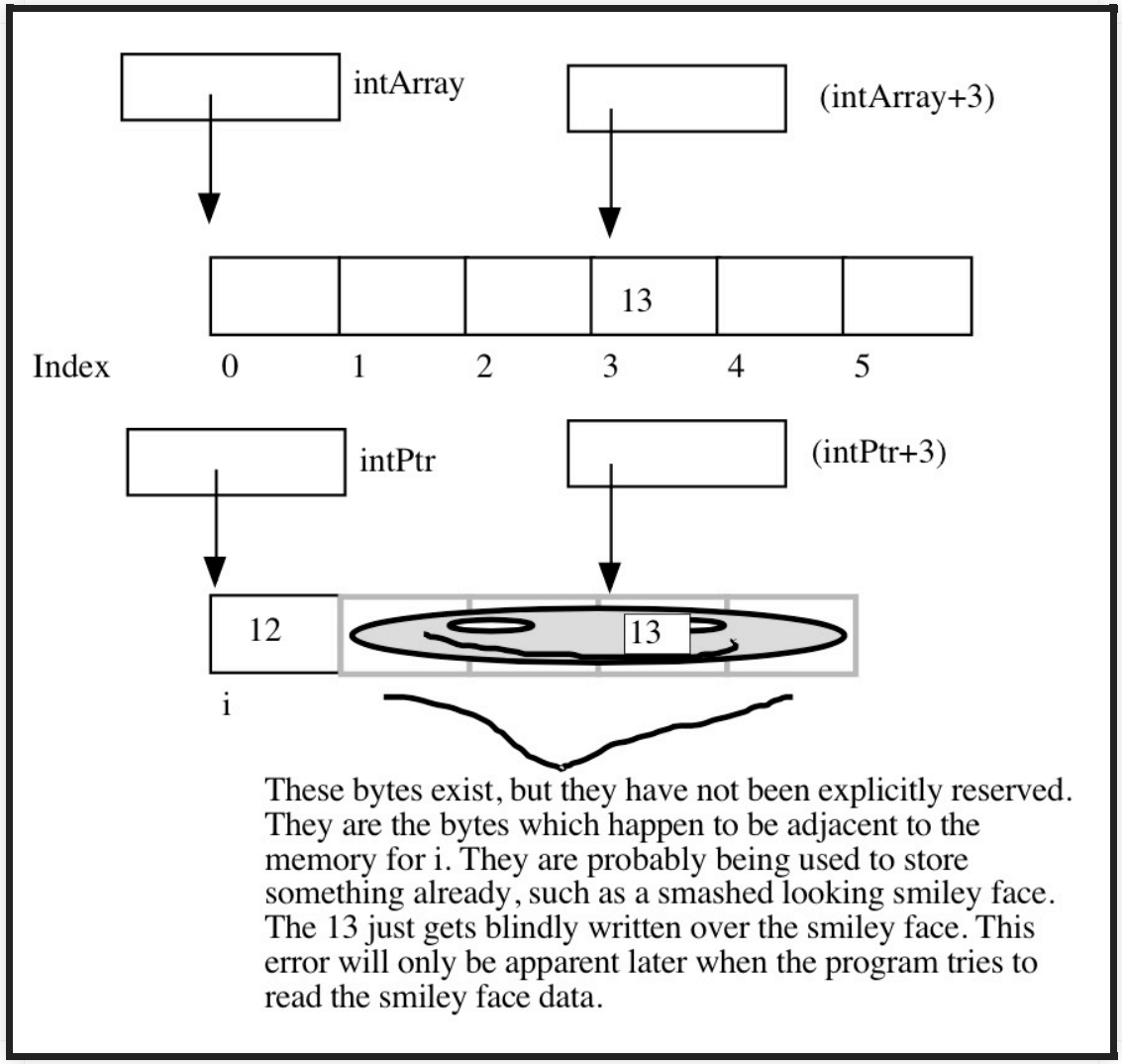
- they both just look like pointers
- the value of an integer array is a pointer to the first element in the array so it's an `int*`
- the value of the integer pointer is also `int*`
- the compiler is perfectly happy to apply the `[ ]` or + syntax to either

```
int intArray[6];
int *intPtr;
int i;

intPtr = &i;

intArray[3] = 13; /* ok */
intPtr[0] = 12;   /* odd, but ok. Changes i. */
intPtr[3] = 13;   /* BAD! There is no integer reserved here! */
```

Arrays and pointers



Array "names" are const

One subtle distinction between an array and a pointer, is that the pointer which represents the base address of an array cannot be changed in the code.

- The array base address behaves like a const pointer.

```
int ints[100]
int *p;
int i;

ints = NULL;      /* NO, cannot change the base addr ptr */
ints = &i;        /* NO */
ints = ints + 1; /* NO */
ints++;           /* NO */

p = ints;         /* OK, p is a regular pointer which can be changed */
                  /* here it is getting a copy of the ints pointer */
p++;              /* OK, p can still be changed (and ints cannot) */
p = NULL;         /* OK */
p = &i;           /* OK */
```

Passing array

Array parameters are passed as pointers.

```
/* The following two definitions of foo look different, */
/* but to the compiler they mean exactly the same thing. */

void foo(int arrayParam[]) {
  /* content */
}

void foo(int *arrayParam) {
  /* content */
}
```

Memory

C manages memory in three ways: statically, automatically and dynamically.

- static and automatic allocations
  - are done automatically
  - require the size of allocation to be known in the compilation.
- dynamic allocation
  - is done manually (useful when memory control is needed)
  - determines the size during run-time

Memory

When a program is loaded into the memory, the memory can be divided into five segments

- text segment
  - where the compiled program (executable instructions) stays
- data segment
  - for static or global variables that are initialized in the source code
- bss (Block Started by Symbol) segment
  - for static or global variables that are uninitialized in the source code

Memory

- heap segment
  - dynamical allocation
- stack segment
  - for automatic variables created by functions (including `main()`)
  - when function exits, the memory is cleared and reused

Heap memory

C provides access to the heap features

- need the standard library with header file "stdlib.h"
- the three functions of interest are
  - `void* malloc(size_t size)`
  - `void free(void* block)`
  - `void* realloc(void* block, size_t size);`

`void* malloc(size_t size)`

- `malloc()` requests a contiguous block of memory of the given size in the heap.
- `malloc()` returns a pointer to the heap block or `NULL` if the request could not be satisfied.
- the type `size_t` is essentially an unsigned long which indicates how large a block the caller would like measured in bytes.
- because the block pointer returned by `malloc()` is a `void*`, a cast will probably be required when storing the `void*` pointer into a regular typed pointer.
  - `int *b; b = (int*) malloc( sizeof(int) * 1000);`

`void free(void* block)`

- `free()` takes a pointer to a heap block earlier allocated by `malloc()` and returns that block to the heap for re-use.
- after the `free()`, the client should not access any part of the block or assume that the block is valid memory.
- The block should not be freed a second time.

`void* realloc(void* block, size_t size);`

- take an existing heap block and try to relocate it to a heap block of the given size which may be larger or smaller than the original size of the block.
- returns a pointer to the new block, or NULL if the relocation was unsuccessful.
- remember to catch and examine the return value of `realloc()` -- it is a common error to continue to use the old block pointer.
- `realloc()` takes care of moving the bytes from the old block to the new block.

Example

```c
#include<stdio.h>
#include<stdlib.h>
#include<assert.h>

int main(){
  int a[10];
  int i;

  int *b;
  b = (int*) malloc(sizeof(int) * 10);
  assert(b != NULL);

  for (i=0; i<10; i++){
    a[i] = i;
    b[i] = i;
  }

  b = realloc(b, sizeof(int) * 20);
  assert(b != NULL);
  for (i=0; i<20; i++){
    printf("%d ", b[i]);
  }
  printf("\n");

  return 0;
}
```

`void free(void *ptr)`

All of a program's memory is deallocated automatically when it exits, so a program only needs to use `free()` during execution if it is important for the program to recycle its memory while it runs -- typically because

- it uses a lot of memory
- or it runs for a long time.

The pointer passed to `free()` must be exactly the pointer which was originally returned by `malloc()` or `realloc()` (or the pointer to the same address), not just a pointer into *somewhere within* the heap block.

- `malloc()` and `realloc()` actually allocates a little bit more memory to store information (e.g. size of the allocated block)
- `free()` is designed to uses the corresponding address to access these information

Disadvantage of being in the heap

- You have to remember to allocate the array, and you have to get it right.
- You have to remember to deallocate it exactly once when you are done with it, and you have to get that right.
- The above two disadvantages have the same basic profile: if you get them wrong,
  - your code still looks right.
  - It compiles fine.
  - It even runs for small cases,
  - but for some input cases it just crashes unexpectedly because random memory is getting overwritten somewhere like the smiley face.

Dynamic strings

The common local variable technique such as char string[1000];

- allocates way too much space most of the time,
- wasting the unused bytes
- and yet fails if the string ever gets bigger than the variable's fixed size.

```
#include <string.h>
/*
   Takes a c string as input, and makes a copy of that string
   in the heap. The caller takes over ownership of the new string
   and is responsible for freeing it.
 */
char* MakeStringInHeap(const char* source) {
  char* newString;
  newString = (char*) malloc(strlen(source) + 1); // +1 for the '\0'
  assert(newString != NULL);
  strcpy(newString, source);
  return(newString);
}
```