Parallel Computing with emory

Dr Paul Ric

http://paulrichmond.shef.ac COM4521/

The University Of Sheffield.

NVIDIA GPU RESEARCH CENTER

Last Week Summary

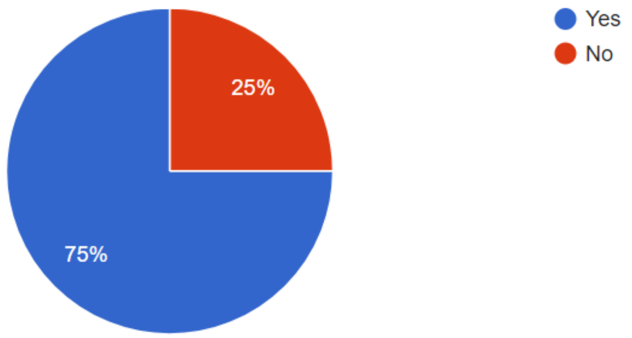
- We learnt about the motivation for using GPUs
- The prevalence of GPUs in HPC
- Begin looking at the C language
- Compiled and built so b
- Demonstrated basic s ipulation
- Compilation and linking

- Now to consider `*` and `&` operators

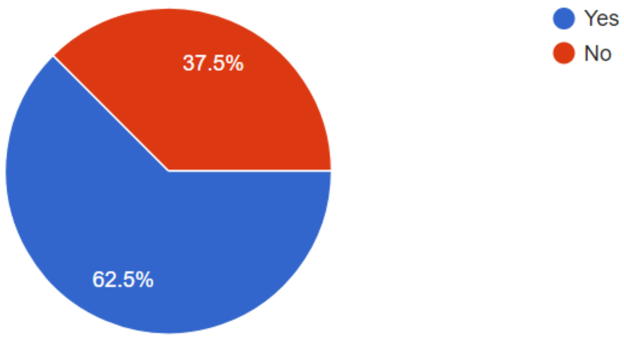
Points from the feedback from

- ❑String concatenation and termination
- ❑Extern keyword
- ❑Transistors != performance (parallelism rules!)
- ❑Unable to complete a specific exercise (incorrect results)
- ❑Setting up my own machine
- ❑Familiarity with VS interfa nd breakpoints)
- ❑REGISTRATION

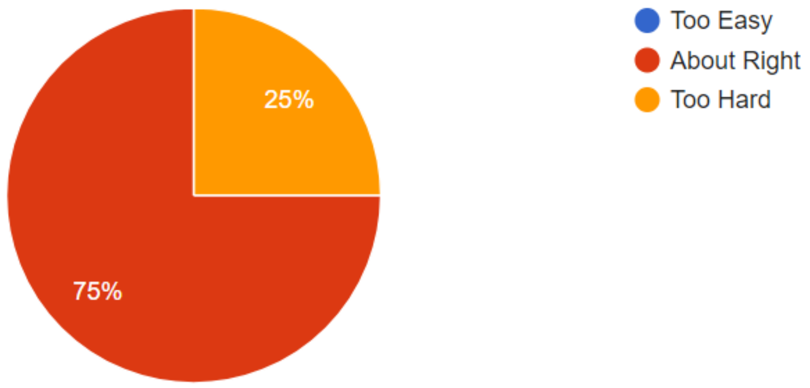
Did you manage to complete all of the lab exercises?



Have you reviewed the exercise solutions?



The difficulty of the Lab class this week was?



This Lecture

- ❑Pointers
- ❑Advanced use of pointers
- ❑Dynamically managed memory
- ❑Structures
- ❑Binary files

Pointers

- A pointer is a variable that contains the address of a variable
- Pointers and arrays are closely related
  - We have already seen some of the syntax with `*` and `&` operators
- The `*` operator can be used to define a pointer variable
- The operator `&` gives t [illegible] le
  - Can not be applied to [illegible]


```c
#include <stdio.h>

void main()
{
    int a;
    int *p;

    a = 8;
    p = &a;
}
```

The University Of Sheffield.

NVIDIA GPU RESEARCH CENTER

Pointer example

```
int a;
int *p;
```

```
printf("a = %d, p = %d\n", a, p);
printf("a = %d, p = 0x%08X\n", a, p);
```

```
a = 8, p = 2750532
a = 8, p = 0x0045FCE0
```

- Same example using a `char`


```
char b;
char *p;
b = 8;
p = &b;
printf("sizeof(b) = %d, sizeof(p) = %d\n", sizeof(b), sizeof(p));
printf("b = %d, p = 0x%08X\n", b, p);
```


- What is the size of `p`?

Pointer example

```
printf("a = %d, p = %d\n", a, p);
printf("a = %d, p = 0x%08X\n", a, p);
```

```
a = 8, p = 2750532
a = 8, p = 0x0045FCE0
```

- Same example using a `char`


```
char b;
char *p;
b = 8;
p = &b;
printf("sizeof(b) = %d, sizeof(p) = %d\n", sizeof(b), sizeof(p));
printf("b = %d, p = 0x%08X\n", b, p);
```


- What is the size of `p`?

```
sizeof(b) = 1, sizeof(p) = 4
b = 8, p = 0x003BF9A7
```

Pointers

- Pointer size does not change regardless of what it points to
  - The size of a pointer on a 32 bit machine is always 4 bytes
  - The size of a pointer on a 64 bit machine is always 8 bytes
- The operator * is the indirection operator and can be used to dereference a pointer
  - I.e. it accesses the val …
- The macro NULL can be assigned to to give it a value 0
  - This is useful in checking if a pointer has been assigned


```
int x = 1; int y = 0;
int *p;
p = &x; // p now points to x (value is address of x)
y = *p; // y is now equal to the value of what p points to (i.e. x)
x++;    // x is now 2 (y is still 1)
(*p)++; // x is now 3 (y is still 1)
p = NULL// p is now 0
```

Pointers and arguments

- C passes function arguments by value
  - They can therefore only be modified locally

```
void swap (int x, int y){
    int temp;
    temp = x;
    x = y;
    y = temp;
}
```


- This is ineffective
  - Local copies of `x` and `y` are exchanged and then discarded

Pointers and arguments

- C passes function arguments by value
  - They can therefore only be modified locally

```
void swap (int *x, int *y){
     int temp;
     temp = *x;
     *x = *y;
     *y = temp;
}
```

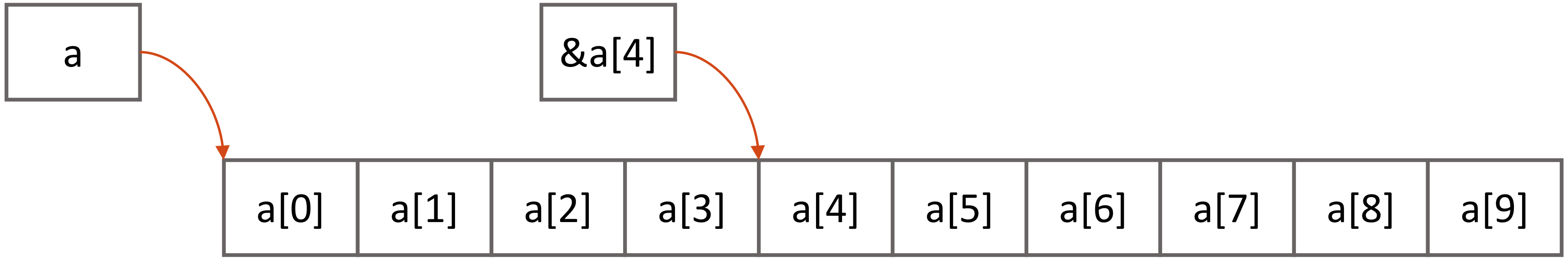

- This swaps the values which x and y point to
- Called by using the & operator

```
swap(&x, &y);
```

Pointers and Arrays

- In the last lecture we saw pointer being used for arrays
  - `char *name` is equivalent to `char name []`
- When we declare an array at compile time the variable **is a pointer** to the starting address of the array
  - E.g. `int a[10];`


```
int a[10] = {1,2,3,4,5,
int *p;
p = &a[4];
printf("*p=%d, p[0]=%d\n", *p, p[0]);
```

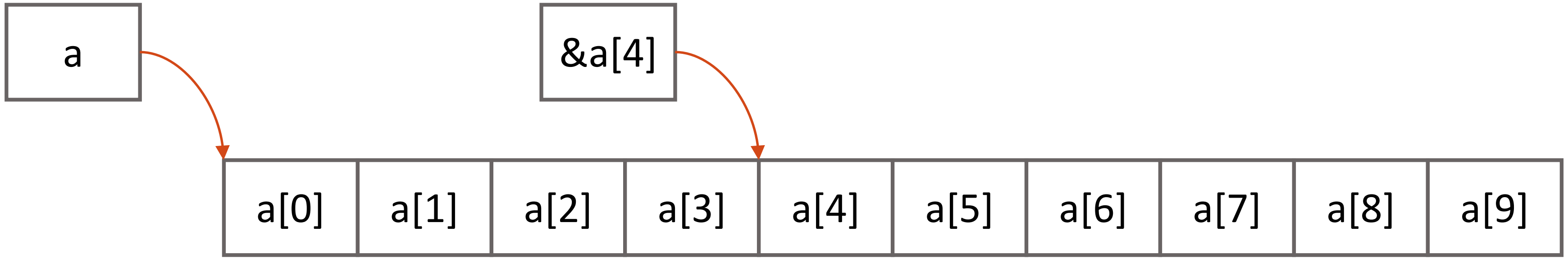


What is the output?

Pointers and Arrays

- In the last lecture we saw pointer being used for arrays
  - `char *name` is equivalent to `char name []`
- When we declare an array at compile time the variable **is a pointer** to the starting address of the array
  - E.g. `int a[10];`


```
int a[10] = {1,2,3,4,5,
int *p;
p = &a[4];
printf("*p=%d, p[0]=%d\n", *p, p[0]);
```




```
*p=5, p[0]=5
```

Pointer and Arrays

- There is however an important distinction between `char *name` and `char name []`
- Consider the following
  - The pointer may be modified
  - The array can only ref


```
char a[] = "hello world 1";
char *b = "hello world 2";
char *temp;
temp = b;
b = a;
a = temp; //ERROR
```

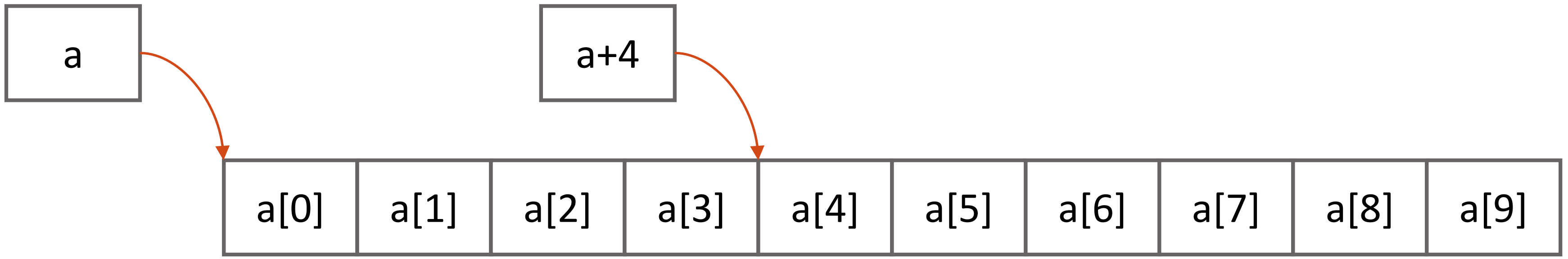
The University Of Sheffield.

NVIDIA GPU RESEARCH CENTER

Pointer arithmetic

- Pointer can be manipulated like any other value
  - `p++:` advances the pointer the next element
  - Pointer arithmetic must not go beyond the bounds of an array
- Incrementing a pointer increments the memory location depending on the pointer type
  - An single integer *point* to the next integer


```
int a[10] = {10,9,8,7,6,5,4,3,2,1};
int *p = a;
p+=4;
printf("*p=%d, p[0]=%d\n", *p, p[0]);
```

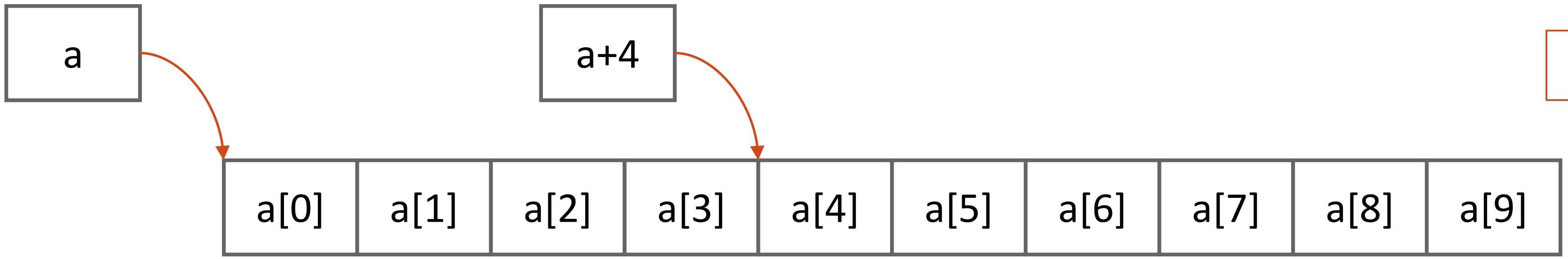


What is the output?

Pointer arithmetic

- Pointer can be manipulated like any other value
  - `p++:` advances the pointer the next element
  - Pointer arithmetic must not go beyond the bounds of an array
- Incrementing a pointer increments the memory location depending on the pointer type
  - An single integer *pointer* increments to the next integer


```c
int a[10] = {10,9,8,7,6,5,4,3,2,1};
int *p = a;
p+=4;
printf("*p=%d, p[0]=%d\n", *p, p[0]);
```




```
*p=6, p[0]=6
```

This Lecture

- ❑Pointers
- ❑Advanced use of pointers
- ❑Dynamically managed memory
- ❑Structures
- ❑Binary files

General Purpose Pointer

- A General purpose pointer can be defined using `void` type
  - A void type can not be dereferenced
  - Arithmetic on a void pointer will increment/decrement by 1 byte


```
void *p;
char c;
int i;
float f;
p = &c;  // ptr has address of chara
p = &i;  // ptr has address of integer data
p = &f;  // ptr has address of float data
```

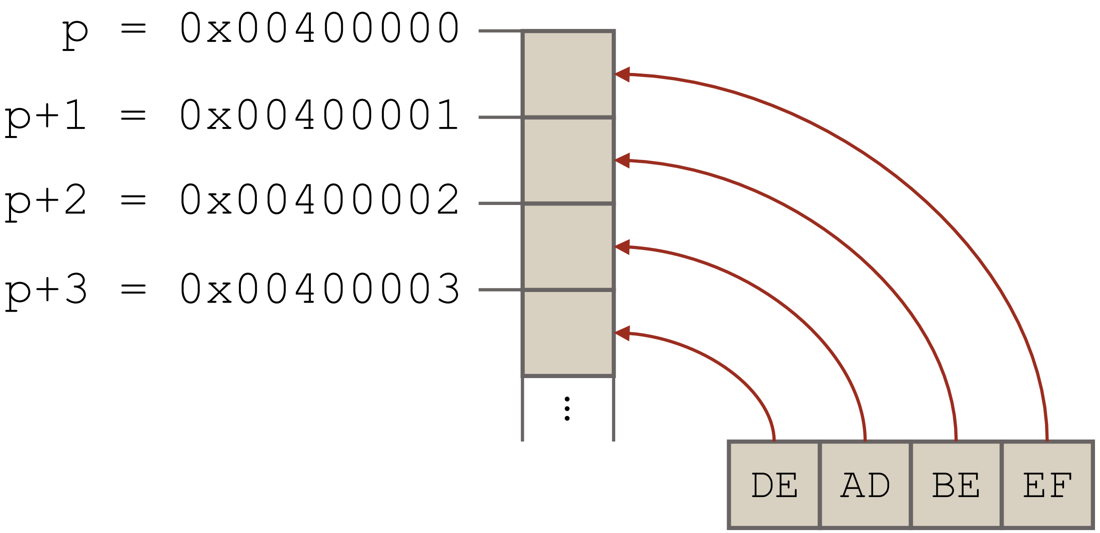
Endianness

- X86 uses little endian format
  - Memory is stored from least significant byte stored at the **lowest** memory

```
unsigned int a = 0xDEADBEEF;
char* p;
p = (char*)&a;
printf("0x%08X, 0x%08X, 0x%08X, 0x%08X\n", p, p+1, p+2, p+3);
printf("0x%02X, 0x%02X, 0x%02                          *(p+2), *(p+3));
```

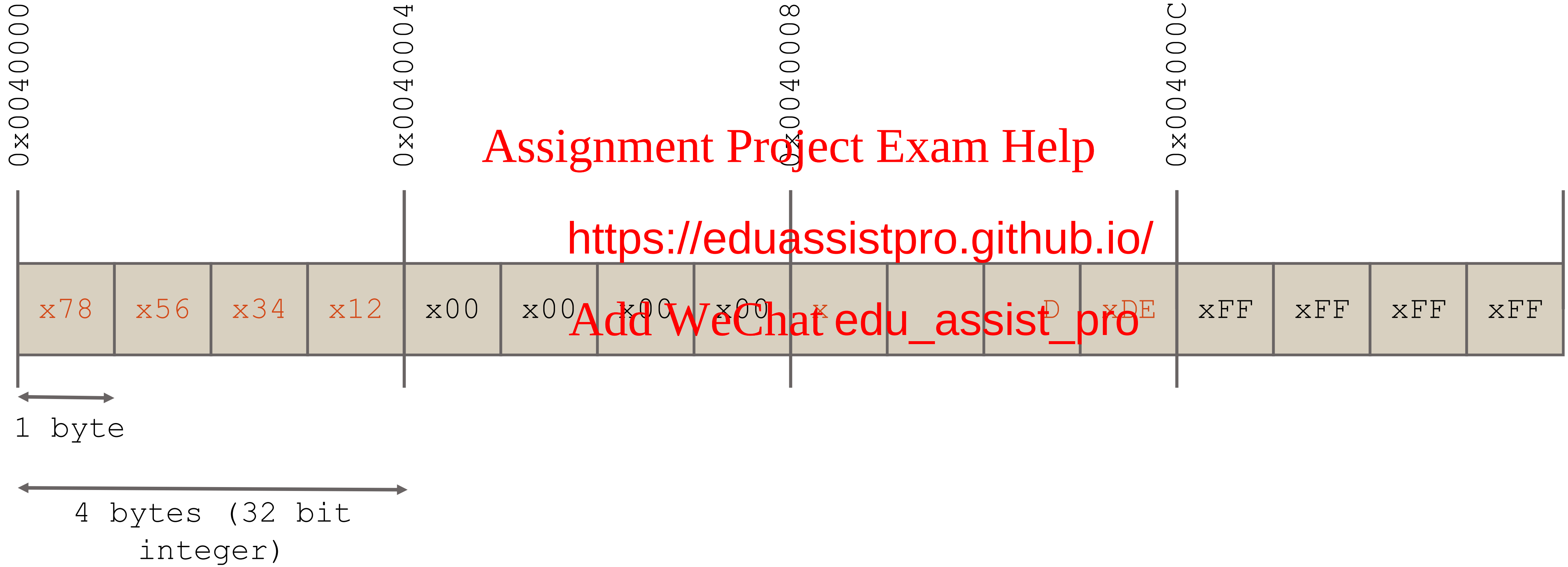


```
0x00400000, 0x00400001, 0x004
0xEF, 0xBE, 0xAD, 0xDE
```



Endianness

```
int a[] = {0x12345678, 0x0000000, 0xDEADBEEF, 0xFFFFFFFF};
```



Endianess is very stange without an example

ssenaidnE si yrev egnats tuohtiw na elpmaxe

Pointers to pointers

- Consider the following
  - `int a[10][20]`
  - `int *b[10]`
- a is a two-dimensional array
  - 200 int sized location
- b is single dimensional array
  - 10 pointers to integers are reserved
  - `B[?]` must be initialised or allocated (later in this lecture)
  - The pointers in `b` may be initialised to arrays of different length


```
char names[][10] = {"Paul", "Bob", "Emma", "Jim", "Kathryn"};
char *p_names[]  = {"Paul", "Bob", "Emma", "Jim", "Kathryn"};
```

Which of the above is better?

Function Pointers

- It is possible to define pointers to functions
  - Functions are however **not** variables

```
int (*f_p)(int, int);
```


- `f_p` is a pointer to a function taking two integer arguments and returning an integer.
  - If `f` is a function then `&f` is
  - Just in the same way that if it to an integer


```
int add(int a, int b);
int sub(int a, int b);

void main()
{
    int (*f_p)(int, int);
    f_p = &add;
    return;
}
```

Using function pointers

- ❑Treat the function pointer like it is the function you want to call.
  - ❑There is no need to dereference (`*f_p`) but you may if you wish

```
f_p = &add;
printf("add = %d\n", f_p(10, 4));
f_p = &sub;
printf("sub = %d\n", f_p(
```



```
add = 14
sub = 6
```

- ❑Care is needed with parenthesis
  - ❑What is `f`?
  - ❑What is `g`?

```
int *f();
int (*g)();
```

Using function pointers

- Treat the function pointer like it is the function you want to call.
  - There is no need to dereference (`*f_p`) but you may if you wish

```
f_p = &add;
printf("add = %d\n", f_p(10, 4));
f_p = &sub;
printf("sub = %d\n", f_p(
```



```
add = 14
sub = 6
```

- Care is needed with parenthesis
  - What is `f`? function returning pointer to int
  - What is `g`? pointer to a function returning int

```
int *f();
int (*g)();
```

const pointers

- Remember the definition of `const`?
  - Not unintentionally modifiable

- What then is the meaning of the following?


```
char * const ptr;
```

```
const char * ptr;
```

```
char const * const ptr;
```

const pointers

- Remember the definition of `const`?
  - Not unintentionally modifiable
  - Read from right to left

https://cdecl.org/ - C Gibberish to English

- What then is the meaning of the following?


```
char * const ptr;
```

The pointer is constant but the data
i.e. declare ptr as const pointer to char

```
char const * ptr;
```

```
const char * ptr;
```

The pointed to data is constant but the pointer is not
i.e. declare ptr as pointer to const char

```
char const * const ptr;
```

The pointer is constant and the data it points to is also constant
i.e. declare ptr as const pointer to const char

This Lecture

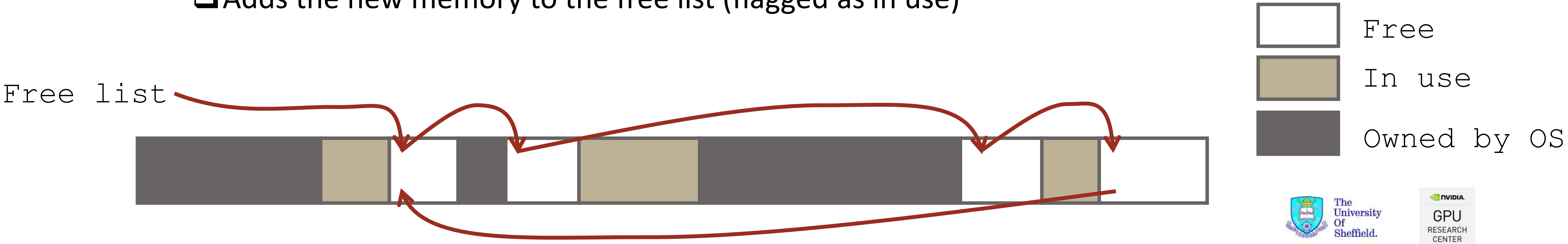
- ❑Pointers
- ❑Advanced use of pointers
- ❑Dynamically managed memory
- ❑Structures
- ❑Binary files

Reminder: Heap vs. Stack

- Stack
  - Memory is managed for you
  - When a function declares a variable it is pushed onto the stack
  - When a function exists all variables on the stack are popped
  - Stack variables are th
  - The stack has size limi
- Heap
  - You must manage memory
  - No size restrictions (except available memory)
  - Accessible by any function

Dynamically allocated memory

- What if we can't specify an array size at compile time (static allocation)
  - The size might not be known until runtime
- We can use the `malloc` system function to get a block of memory on the heap.
  - `malloc` keeps a list of free blocks of memory on the heap
  - `malloc` returns the fi ough "first fit"
  - If a block is too big it is s
    - Part is returned to the user and the remain e free list
  - If no suitable block is found `malloc` wil rger block from the OS
    - Increases the size of the heap
    - Adds the new memory to the free list (flagged as in use)



malloc

- `void *malloc(size_t size)`
  - Returns a pointer to void which must therefore be cast

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

void main()
{
    int *a;
    a = (int*) malloc(sizeof(int) * 10);
}
```


- Use `sizeof` function to ensure correct number of bytes per element
- `a` can now be used as an array (as in the previous examples)
- Result of `malloc` can be implicitly cast

Memory leaks

- Consider the following
  - b is on the stack and is free'd on return
  - a points to an area of memory which is allocated
  - a then points to b, there is no pointer to the area of memory that was allocated

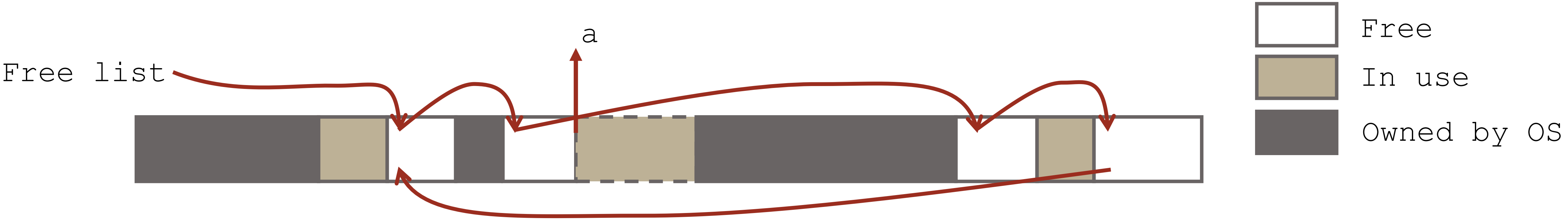

```
void main()
{
    int b[10] = {1,2,3,4,5,6,7,8,9,10};
    int *a;
    a = (int*) malloc(sizeof(int) * 10);
    a = b;

    return;
}
```

- This is known as a memory leak
  - Where we allocate memory we must also free it

free

- The `free` function will add a previous used area of memory to the free list
  - If it is adjacent to another free block these will be coalesced into a larger block
- `void free (void`


```
int *a = (int*) malloc(                    locate
free(a);//free
```

free

- The `free` function will add a previous used area of memory to the free list
  - If it is adjacent to another free block these will be coalesced into a larger block
- `void free (void`


```
int *a = (int*) malloc(                              locate
free(a);//free
```

free

- The `free` function will add a previous used area of memory to the free list
  - If it is adjacent to another free block these will be coalesced into a larger block
- `void free (void`


```
int *a = (int*) malloc(                                  locate
free(a);//free
```

free

- The `free` function will add a previous used area of memory to the free list
  - If it is adjacent to another free block these will be coalesced into a larger block

- `void free (void`

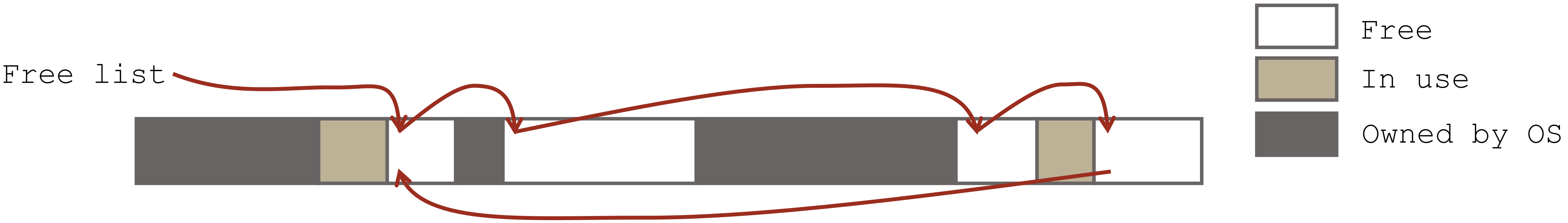

```
int *a = (int*) malloc(                                locate
free(a);//free
```


free

- The `free` function will add a previous used area of memory to the free list
  - If it is adjacent to another free block these will be coalesced into a larger block

- `void free (void`


```
int *a = (int*) malloc(                    locate
free(a);//free
```

Memory operations

- Set a block of memory to char value
  - `void *memset(void *str, int c, size_t n)`
    - Can be used to set any memory to a value (e.g. `0`)
    - Useful as allocated memory has undefined values


```
int *a;
int size = sizeof(int) * 10;
a = (int*) malloc(size);
memset(a, 0, size);
```


- Coping memory
  - `void *memcpy(void *dest, const void *src, size_t n)`
    - Copies `n` bytes of memory from `src` to `dst`

```
int *a;
int b[] = {1,2,3,4,5,6,7,8,9,10};
int size = sizeof(int) * 10;
a = (int*) malloc(size);
memcpy(a, b, size);
```

This Lecture

- ❑Pointers
- ❑Advanced use of pointers
- ❑Dynamically managed memory
- ❑Structures
- ❑Binary files

Structures

- A structure is a collection of one or more variables
  - Variables may be of different types
  - Groups variables as a single unit under a single name
- A structure is not the same as a class (at least in C)
  - No functions
  - No private members
  - No inheritance
- Structures are defined o
  - Values can be assigned with an initialisati ugh structure member operator '.'


```
struct vec{
    int x;
    int y;
};

struct vec v_1 = {123, 456};
struct vec v_2;
v_2.x = 123;
v_2.y = 456;
```

Features of structures

- As with everything, structures are passed by value

```
struct vec make_vec(int x, int y){
    struct vec v = {x, y};
    return v;
}
```


- Pointers to structures er operator
    - '->' accesses member
    - Alternatively dereference and use the erator '.'


```
struct vec v = {123, 456};
struct vec *p_vec = &v;//CORRECT
p_vec->x = 789;//CORRECT
p_vec.x = 789; //INCORRECT
```

- Declarations and definition can be combined

```
struct vec{
    int x;
    int y;
} v1 = {123, 456};
```

Structure assignment

- Structures can be assigned
  - Arithmetic operators not possible (e.g. `vec_2 += vec_1`)

```
struct vec vec_1 = {12, 34};
struct vec vec_2 = {56, 78};
vec_2 = vec_1;
```


- **BUT** No deep copies o
  - E.g. if a person struc pointer members (forename and surname)


```
struct person paul, imposter;
paul.forename = (char *) malloc(5);
paul.surname = (char *) malloc(9);
strcpy(paul.forename, "Paul");
strcpy(paul.surname, "Richmond");
imposter = paul;     // shallow copy
strcpy(imposter.forename, "John");
printf("Forename=%s, Surname=%s\n", paul.forename, paul.surname);
```

What is the Output?

Structure assignment

- Structures can be assigned
  - Arithmetic operators not possible (e.g. `vec_2 += vec_1`)

```
struct vec vec_1 = {12, 34};
struct vec vec_2 = {56, 78};
vec_2 = vec_1;
```


- **BUT** No deep copies o

  - E.g. if a person struc pointer members (`forename` and `surname`)


```
struct person paul, imposter;
paul.forename = (char *) malloc(5);
paul.surname = (char *) malloc(9);
strcpy(paul.forename, "Paul");
strcpy(paul.surname, "Richmond");
imposter = paul;     // shallow copy
strcpy(imposter.forename, "John");
printf("Forename=%s, Surname=%s\n", paul.forename, paul.surname);
```

```
Forename=John, Surname=Richmond
```

Structure allocations

- Structures passed as arguments have member variables values **copied**
  - If member is a pointer then pointer value copied not the thing that points to it (shown on last slide)
  - Passing large structures by value can be quite inefficient

- Structures can be alloc pointer
  - `sizeof` will return the combined size re members
  - Better to pass big structures as pointer

```
struct vec *p_vec;
p_vec = (struct vec *) malloc(sizeof(struct vec));
//...
free(p_vec);
```

Type definitions

- The keyword `typedef` can be used to create 'alias' for data types
  - Once defined a `typedef` can be used as a standard type

```
//declarations
typedef long long int int64;
typedef int int32;
typedef short int16;
typedef float vec3f [3];

//definitions
int32 a = 123;
vec3f vector = {1.0f, -1.0f, 0.0f};
```


- `typedef` is useful in simplifying the syntax of `struct` definitions

```
struct vec{
    int x;
    int y;
};
typedef struct vec vec;
vec p1 = {123, 456};
```

This Lecture

- ❑Pointers
- ❑Advanced use of pointers
- ❑Dynamically managed memory
- ❑Structures
- ❑Binary files

Binary File Writing

- `size_t fwrite(const void *ptr, size_t size, size_t nmemb, FILE *stream)`
  - `size_t`: size of single object
  - `nmemb`: number of objects
  - Returns the number of objects written (if not equal to `nmemb` then error)


```
void write_points(FILE* f, p
  fwrite(points, sizeof(poin                          f(point), f);
}

void main(){
  point points[] = { 1, 2, 3, 4 };
  FILE *f = NULL;
  f = fopen("points.bin", "wb"); //write and binary flags
  write_points(f, points);
  fclose(f);
}
```

Binary file reading

- size_t fread(void *ptr, size_t size, size_t nmemb, FILE *stream)

```
void read_points(FILE *f, point *points, unsigned int num_points){
    fread(points, sizeof(point), num_points, f);
}

void main(){
    point points[2];
    FILE *f = NULL;
    f = fopen("points.bin", "rb"); //read an                s
    read_points(f, points, 2);
    fclose(f);
}
```

Summary

- Pointers and arrays are closely related
- Using `&` and `*` we can manipulate memory
- Pointers can point to variable definitions or functions
- To dynamically alloca `malloc`
- Any memory allocate
- Structures and typedefs allow us to lpful storage units
- Files can be written to with raw binary data