C Programming
Dynamic Allocation

Adam Sampson (based on material by Greg Michaelson)
School of Mathematical and Computer Sciences
Heriot-Watt University

Recap: arrays and pointers

`type name[size];`

- Allocates space for `size` elements of `type`

`type *name;`

- Allocates space for pointer to type
- Does **not** allocate space for elements

Dynamic memory allocation

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

- Allocates `size` bytes from the heap
- Returns address of the first byte, or `NULL` if out of memory (remember to always check for `NULL` when using this!)
- Memory is uninitialised – may contain junk
- Like `new` in Java/C++, but no constructors or GC
- Declared in `<stdlib.h>`

void *

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

- A `void *` is a pointer to something, but without specifying what!
- `void *` will convert to any other pointer type automatically – **no cast is needed**

```
int *thing = malloc(4);
```

void *

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

- You will see some code doing:
  `int *thing = (int *) malloc(4);`
- This is only needed **in C++** –
  it will work in C, but it's not necessary (and is poor style)

Using `malloc`

- To allocate an array with `malloc`:
  `malloc(`*`number * sizeof(type)`*`)`
  - Allocate space for *`number`* of elements of *`type`*

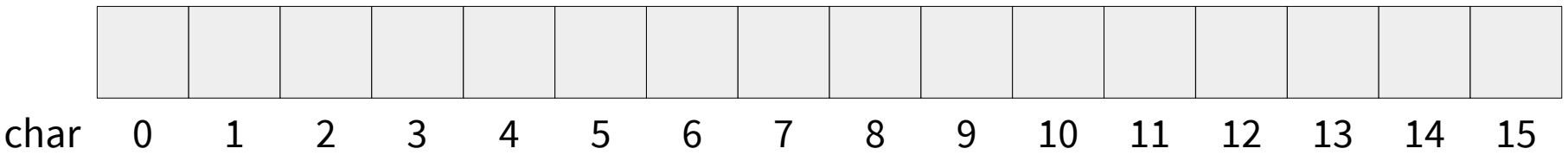
- The C idiom for allocating an object:

```
struct person *fred =
    malloc(sizeof *fred);
```

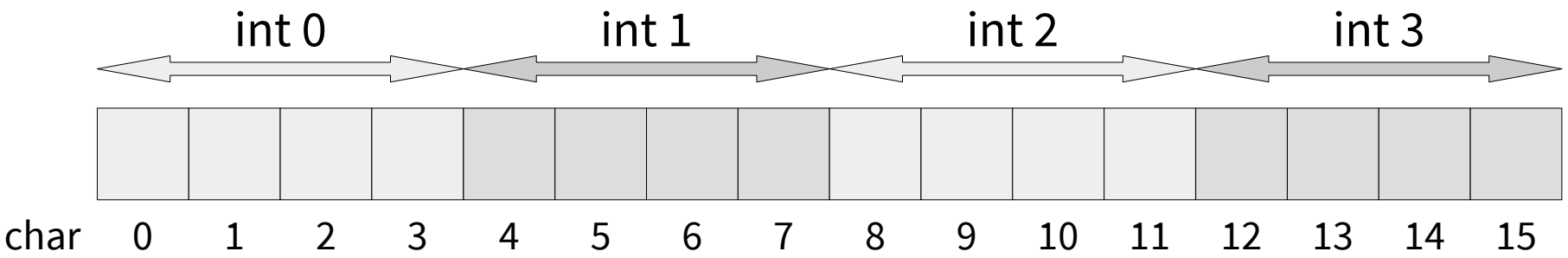
  - `sizeof`, without brackets, also works on expressions
  - So this is “allocate enough bytes for the type `fred` points to”

Bytes are bytes

- `malloc(16)` returns a pointer to 16 `chars`



- `malloc(4 * sizeof(int))` returns a pointer to 4 `ints`... but it's really doing the same as above



calloc

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

- Use `calloc` when allocating arrays
- It's like `malloc(nmemb * size)`, except...
  - It **checks for overflow** in the multiplication, which is a common source of (security) faults
  - It **initialises the memory** to all zeroes

free

```
free(void *ptr);
```

- Returns memory to heap
- `ptr` must have been allocated by `malloc`
  (or a function like `calloc` that uses `malloc` internally)

- **Does not recurse** into data structures:
  if you are freeing a structure containing a pointer, you must free the inner pointer by hand first

Allocating a string dynamically

```
char *name;
```

- Allocates space for pointer to characters...
- ... but you must allocate space for the characters too

```
name = malloc(size);
```

- Remember to count one extra space for the `'\0'`!

`scopy.c`

String copy

String copy

```
char *scopy(const char *s);
```

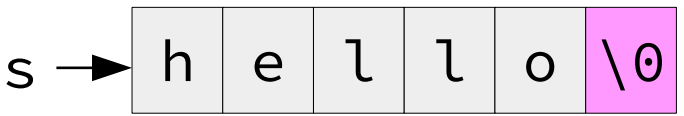
- Make a copy of string `s` – like `<string.h>`'s `strcpy`

- Find length of `s`
- Allocate new memory of size length **+ 1**
- Copy `s` element-by-element to new memory
- Return address of new memory

String copy

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

char *scopy(const char *s)
{
    int l = slength(s) + 1;
    char *c = malloc(l);
    for (int i = 0; i < l; i++) {
        c[i] = s[i];
    }
    return c;
}
```

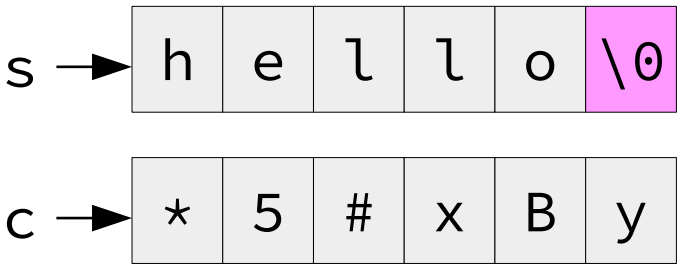
e.g. `scopy("hello");`



String copy

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

char *scopy(const char *s)
{
    int l = slength(s) + 1;
    char *c = malloc(l);
    for (int i = 0; i < l; i++) {
        c[i] = s[i];
    }
    return c;
}
```

e.g. `scopy("hello");`

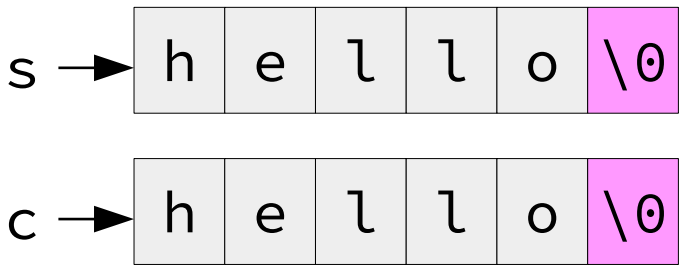


Newly-allocated memory may contain junk

String copy

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

char *scopy(const char *s)
{
    int l = slength(s) + 1;
    char *c = malloc(l);
    for (int i = 0; i < l; i++) {
        c[i] = s[i];
    }
    return c;
}
```

e.g. `scopy("hello");`



String copy

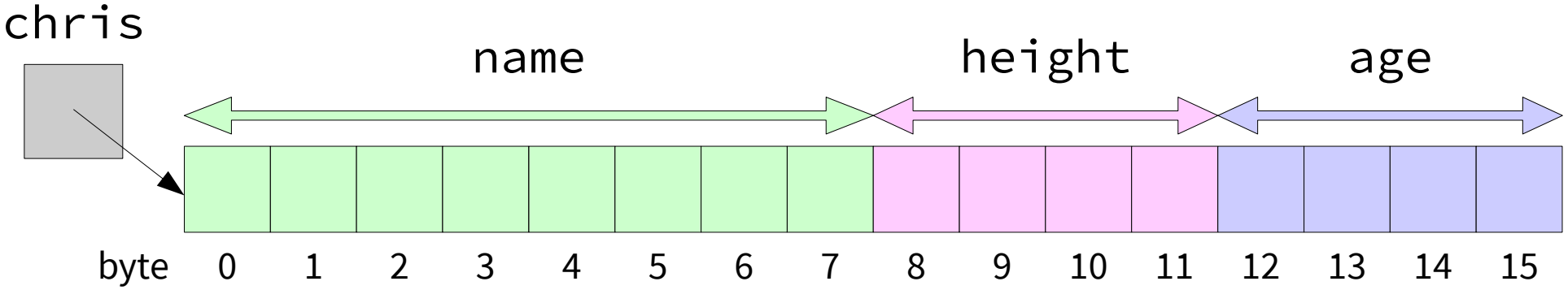
```
int main(int argc, char *argv[])
{
    char *s = scopy("typewriter");
    printf("%s\n", s);
    free(s);
    return 0;
}

$ ./scopy
typewriter
```

Structure assignment

- If you assign `struct foo` to `struct foo`…
  - Fields are copied automatically
  - Gives two identical versions in different pieces of memory
  - This is **copying**
- If you assign `struct foo *` to `struct foo *`…
  - Both pointers now refer to the same piece of memory
  - Gives two aliases for the same structure
  - This is **sharing**

Structure declaration

```
struct person {
    const char *name;
    float height;
    int age;
};
struct person *chris = malloc(sizeof(struct person));
                or:  = malloc(sizeof *chris);
```



Structure pointer field access

`exp->field`

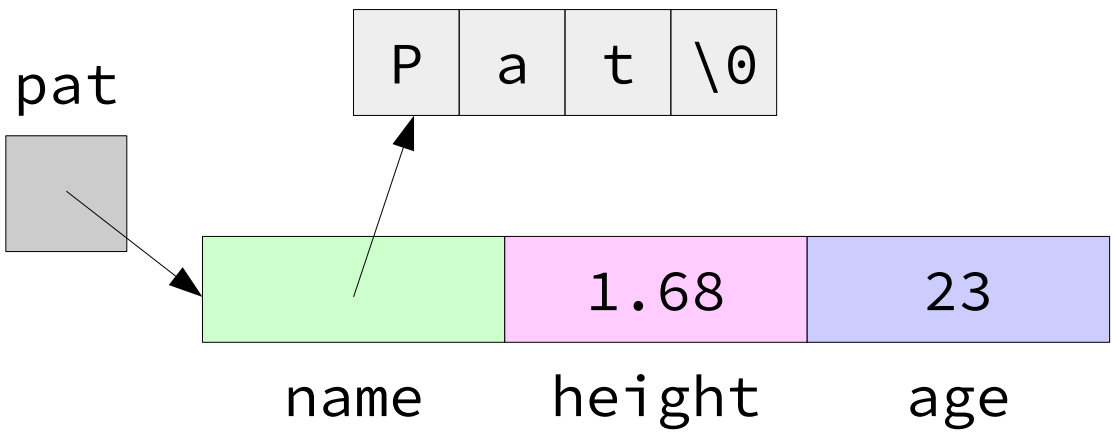
- If `exp` is a pointer to a struct, this refers directly to the field `field` within the struct
- Syntactic sugar for: `(*exp).field`

`person_dynamic_memory.c`

Structure pointer field access

Structure pointer field access

```
pat->name = “Pat”;
pat->height = 1.68;
pat->age = 23;
```

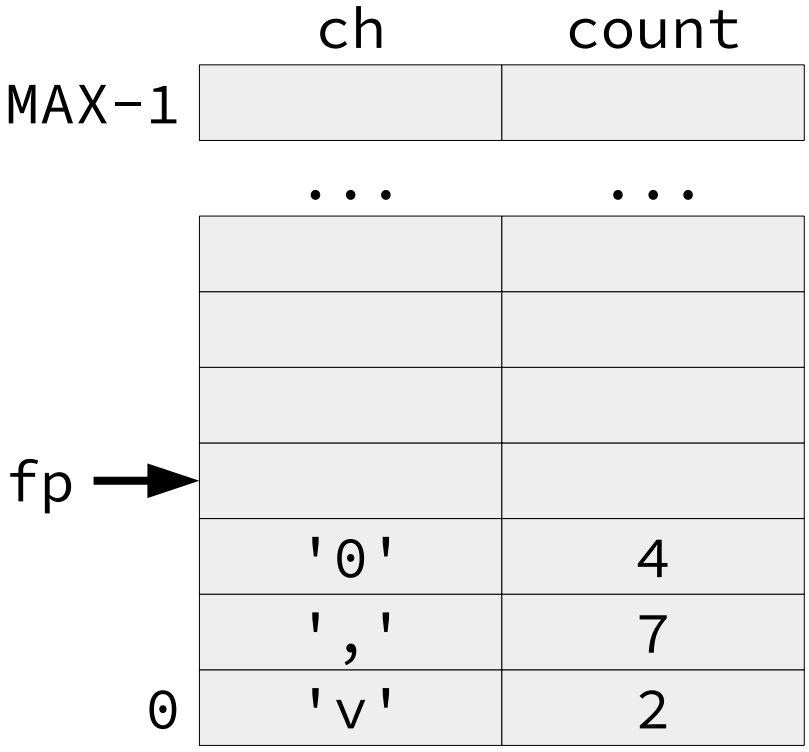


`freq1.c`

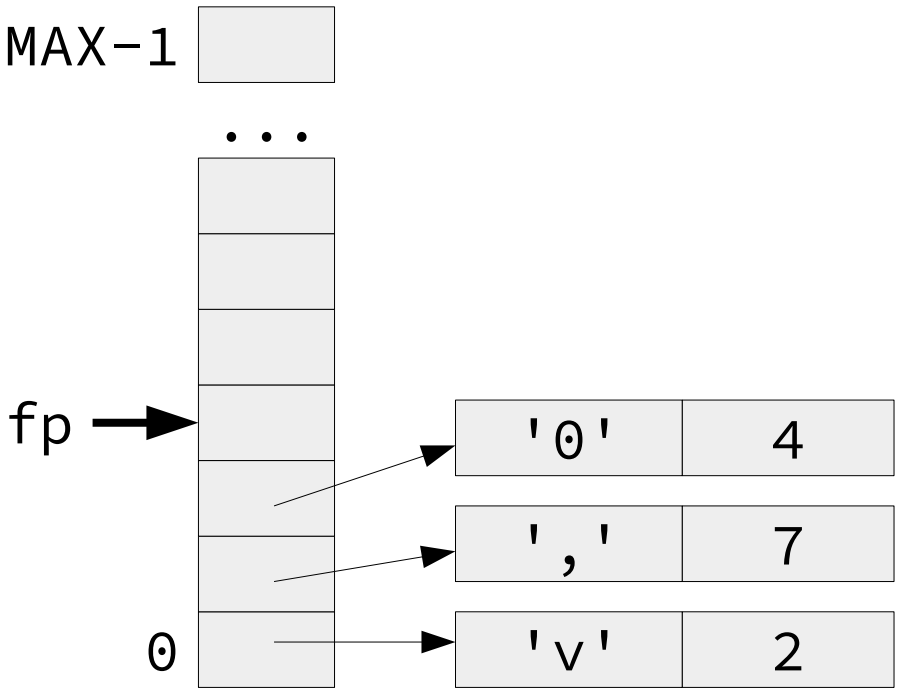
Frequency count with struct pointers

Frequency count

Old `f`: array of structs

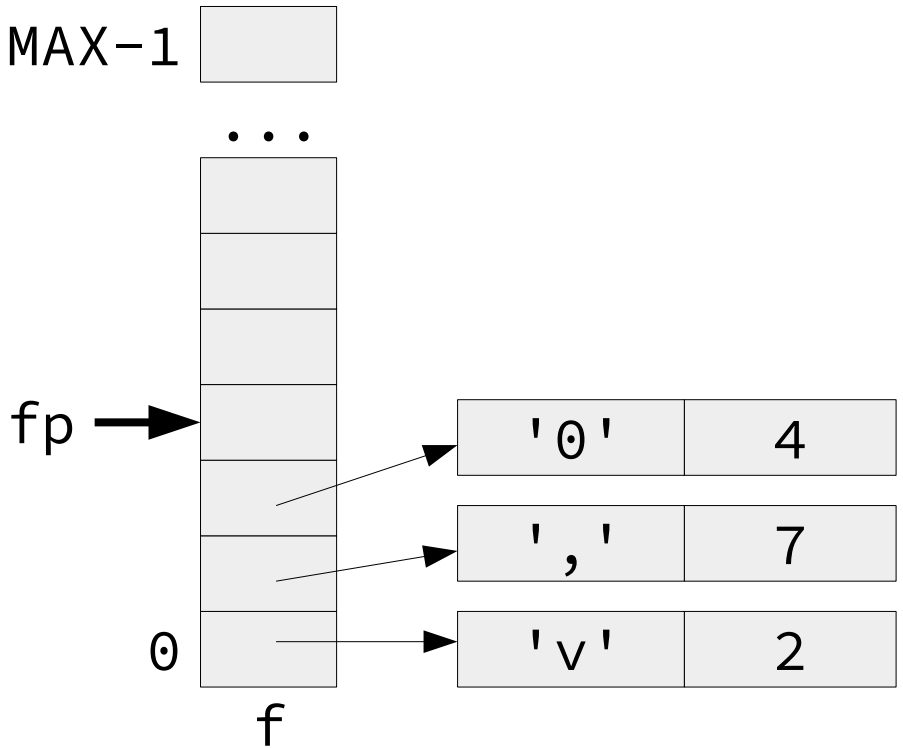


New `f`: array of **pointers to** structs



Frequency count

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

#define MAX 256

struct freq {
    int ch;
    int count;
};

struct freq *f[MAX];
int fp;
```



Frequency count

```
void incFreq(int ch)
{
    for (int i = 0; i < fp; i++) {
        if (f[i]->ch == ch) {
            f[i]->count++;
            return;
        }
    }
```

Frequency count

```
        if (fp == MAX) {
            printf("more than %d different
                    characters\n", MAX);
            exit(1);
        }
        f[fp] = malloc(sizeof(struct freq));
        f[fp]->ch = ch;
        f[fp]->count = 1;
        fp++;
}
```

Frequency count

```
void showFreq(void)
{
    for (int i = 0; i < fp; i++) {
        printf("%c : %d\n",
             f[i]->ch, f[i]->count);
    }
}
```

Frequency count

```
int main(int argc, char *argv[])
{
/* ... the same as before, up to ... */
    fclose(fin);
    showFreq();
    for (int i = 0; i < fp; i++) {
        free(f[i]);
    }
    return 0;
}
```

Remember to `free` anything you allocated with `malloc`!