# Pointers

Pointers are used for:

Managing memory

- `ptr = malloc(size)` `// allocate memory`
- `ptr = realloc(ptr, size)` `// re-allocate`
- `ptr = calloc(num, size)` `// clear and allocate`
- `free(ptr)` `// de-allocate`

Dynamic data

- dynamic arrays
- dynamic structures
- linked lists
- trees (next chapter)

# Arrays

Constant arrays – the compiler knows the size

```
char name[10];
```

Variable arrays – a variable is used to specify the size of the array on creation, but the size stays fixed after that, and the compiler keeps track of it – contrary to popular opinion, you don't need pointer notation

```
char name[n];
```

Dynamic arrays – the size can change after creation

```
char *name = malloc(n);
name = realloc(name, 2*n);
```

# Creating a constant array

Allocate memory inside a function:

```
void f() {
  char name[10];
   ...
```

The space is allocated when the function is called
The initial contents are unknown ("rubbish")
The space "disappears" when the function returns
    It gets re-used by other function calls

# Creating a variable array

Allocate memory inside a function:

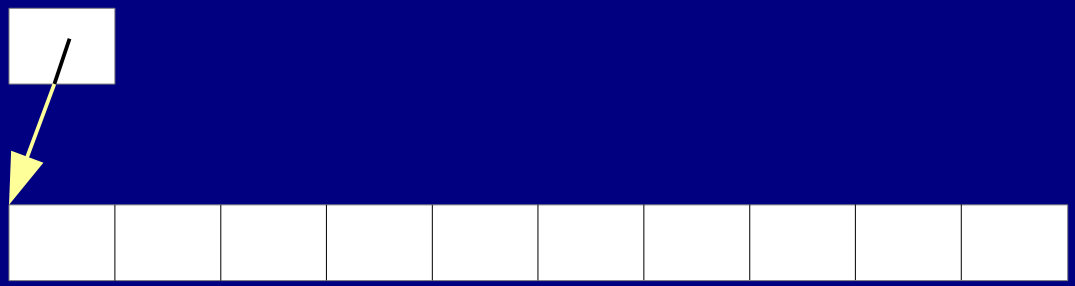
```
void f(int n) {
  char name[n];
   ...
```

If n is 10 at the time of the allocation, an array of length 10 is allocated
The compiler remembers length 10, even if n changes
The space is still deallocated when the function returns
This is a C99 feature, but it was also a gcc extension to C89, so no options are needed to gcc to make it work

# Creating a dynamic array

Allocate memory inside a function:

```
void f(int n) {
  char *name;
   name = malloc(n);
   ...
```



Switch to pointer notation

**name** is a pointer to the start of the actual array

The space is allocated from permanent memory (the heap)

The space remains allocated to the end of the program
or until you call free

A pointer is a number (byte-address) but is visualized as an arrow

# Notation

You can declare and initialize a pointer in one

```
char *name = malloc(n);
```

This means **`char *name`** followed by **`name =`** ... and not **`char *name`** followed by **`*name =`** ... so some programmers write

```
char* name = malloc(n); (or char * name...)
```

However, this isn't good when declaring multiple pointers

```
char* name1, name2;
```

declares **`name2`** as a **`char`**, not **`char *`**, so it is better to write

```
char *name1, *name2;
```

# Indexing

You can index a dynamic array, and pass it to functions, in the same way as any other kind of array

```
char *name = malloc(n);
name[0] = toupper(name[0]);
char *name2 = strdup(name)
```

When indexing, the compiler automatically follows the pointer, first
When calling a function, the compiler passes a dynamic array directly, for other arrays it converts to a pointer first, so the function is unaffected by the difference

# Array arguments

A function can have a constant array argument:

```
void capital(char s[10]) {
   s[0] = toupper(s[0]);
}
int main() {
   char name[10];
   fgets(name, 10, stdin);
   capital(name);
}
```

The compiler implicitly passes a pointer in the call, and implicitly treats s as a pointer in the function

A pointer to name is copied into the pointer s, and so the function affects the original array

# Array arguments

A function can have an unknown-sized array argument:

```
void capital(char s[]) {
   s[0] = toupper(s[0]);
}
int main() {
   char name[10];
   fgets(name, 10, stdin);
   capital(name);
}
```

The function doesn't need to know the length (no bounds checking)
This works for strings, for example, because the function uses **`strlen`**
The function now looks more generic/flexible/reusable because it can be re-used on arrays of different sizes

# Array arguments

A function can have a variable array argument:

```
void sum(int n, int list[n]) {
  int total = 0;
  for (int i=0; i<n; i++) total += list[i];
   return total;
}
int main() {
   int numbers[5] = {3, 9, 2, 7, 4};
   int result = sum(5, numbers);
   printf("Total: %d\n", result);
}
```

The argument n must come before the argument that uses n
The compiler remembers the length, even if n changes

# Array arguments

A function can have a dynamic array argument:

```
void sum(int n, int *list) {
  int total = 0;
  for (int i=0; i<n; i++) total += list[i];
   return total;
}
int main() {
   int numbers[5] = {3, 9, 2, 7, 4};
   int result = sum(5, numbers);
   printf("Total: %d\n", result);
}
```

This is actually what happens, no matter what kind of array argument
The kind of array in main doesn't have to match the kind of argument,
unless the function does dynamic operations on the array

# Increasing the size of an array

A dynamic array can be increased in size like this:

```
char *name = malloc(10);
...
name = realloc(name, 20);
```

The first argument to realloc must be a pointer to the start of a previously allocated block of memory

The realloc function does a lot of work for you

- it allocates a new block of space of size 20 bytes
- it copies the contents of the old space into the new space
- it deallocates the old space

# Freeing a dynamic array

A dynamic array can be deallocated like this:

```
char *name = malloc(10);
...
free(name);
```

The argument to free must be a pointer to the start of some space that was previously allocated with malloc or calloc

After freeing, you must not touch the space again, nor call free again the system can re-use the space for later allocations

# The NULL pointer

One particular pointer is special - the NULL pointer
The NULL constant is defined in stdio.h
It is not the same as the null character `'\0'` that terminates strings
It points to nothing (actually to memory that doesn't belong to you)
Any attempt to index it causes a crash
You can use it to initialize or terminate:

```
char *name = NULL;      // safe zero-length array
...
name = malloc(10);      // now initialize
...
free(name);
name = NULL;            // prevent accidental
                        // access to freed memory
```

# Other types

What about an array of ints? Use sizeof:

```
int *numbers = malloc(10 * sizeof(int));
```

Or use the calloc function:

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

The calloc function takes two arguments instead of one
As well as doing the same as malloc, it clears the allocated space
It clears all bits to zero, so

- chars become `'\0'`
- ints become 0
- doubles become 0.0
- ...

# Example: strdup

The standard strdup function makes a copy of a string, e.g. because the original string is constant, or is in a non-dynamic array that is going to be deallocated at the end of a function call, or is in a dynamic array that is too big for it and you want to allocate exactly the right space:

```
#include <stdlib.h>

char *strdup (char *s) {
  char *result = malloc(strlen(s) + 1);
  if (result == NULL) { ... }
  strcpy( result, s ) ;
  return result ;
}
```

The malloc/calloc functions return NULL if memory runs out - you could try printing and exiting (but there may not be enough memory!)

# Example: join

Suppose we want a function to join two strings to form a new string:

```
#include <stdlib.h>

char *join (char *s1, char *s2) {
  int n = strlen(s1) + strlen(s2);
  char *result = malloc(n + 1);
  strcpy(result, s1);
  strcat(result, s2);
  return result;
}
```

# Pointing into the middle

You can point into the middle of an array, and so pass a portion of an array to a function:

```
#include <stdlib.h>

// Add s2 on to the end of s1 (assumed big enough)
char *strcat (char *s1, char *s2) {
  int n = strlen(s1);
  char *remainder = &s1[n];
  strcpy(remainder, s2);
}
```

The pointer **`&s1[n]`** points to position n in the array s1
You can also write it as **`(s1+n)`** which means "pointer s1 plus n times the size of the things **`s1`** points to"

# Example: contacts

This example uses an array of structures to implement a contacts list

```
typedef struct {
  char name[20];
  char number[12];
} entry;

entry *newContacts( ) {
  entry *contacts = calloc(5000, sizeof(entry));
  strcpy(contacts[0].name, "Aardvark" ) ;
  strcpy(contacts[0].number, "0117-9093145" ) ;
  return contacts;
}
```

# Array lists

The contact list is an example of using a dynamic array to store a list
This kind of list is often called an array list
It has some disadvantages

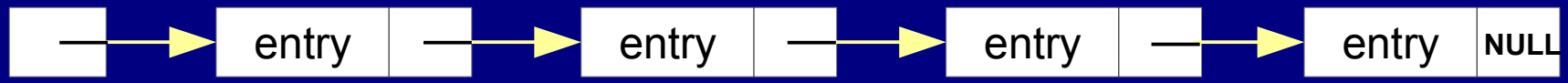
- the array is almost always bigger than we need
- we have to keep track of how many entries are in it
- if it runs out of space, we have to reallocate
- it is expensive to insert an entry in the middle, or keep ordered

The right strategy for sizing array lists is

- create them small
  - to keep space down if a program has a lot of nearly empty ones
- double the size when they run out of space
  - to keep copying time down if a program uses big ones

# Linked lists

An alternative to an array list is a linked list, using pointers




Add a pointer to each entry, which points to the next entry
It is particularly easy to add another entry at the front
The structure type for an entry is

```
typedef struct entry {
  int number;
  struct entry *next;
} entry;
```

The type of **next** has to be **struct entry** because the type **entry** doesn't exist yet, so there is a tag name after **typedef struct**

# A crude linked list

Here's a linked list of numbers without dynamic memory - this isn't normal, because we wouldn't know how many items there would be

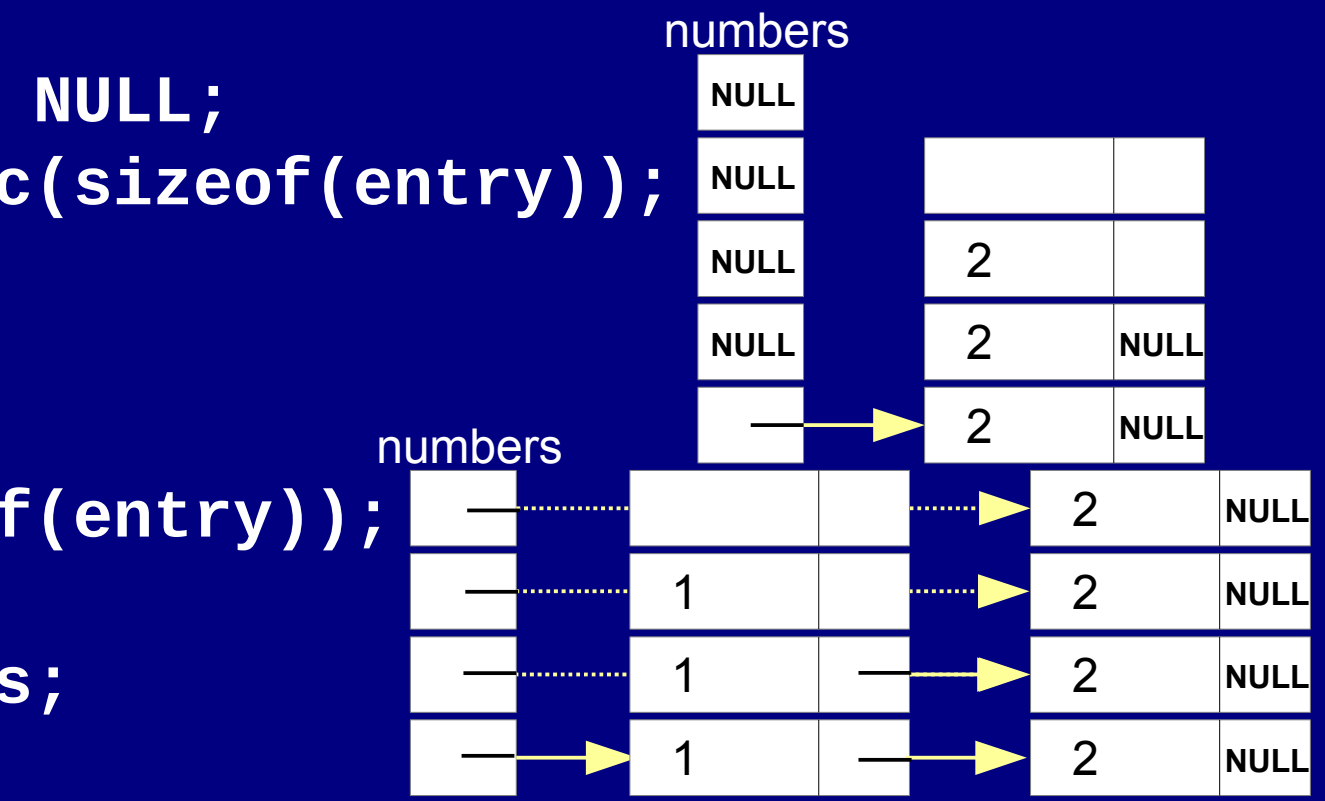

```
typedef struct entry {
  int number;
  struct entry *next;
} entry;

int main() {
  entry e3 = { 17, NULL };
  entry e2 = { 42, &e3 };
  entry e1 = { 23, &e2 };
  entry *numbers = &e1;
}
```

# Linked lists: more dynamic example

The notation **p->x** means **(*p).x** i.e. "field x of structure p points to"

```
typedef struct entry {
  int number;
  struct entry *next; } entry;
```

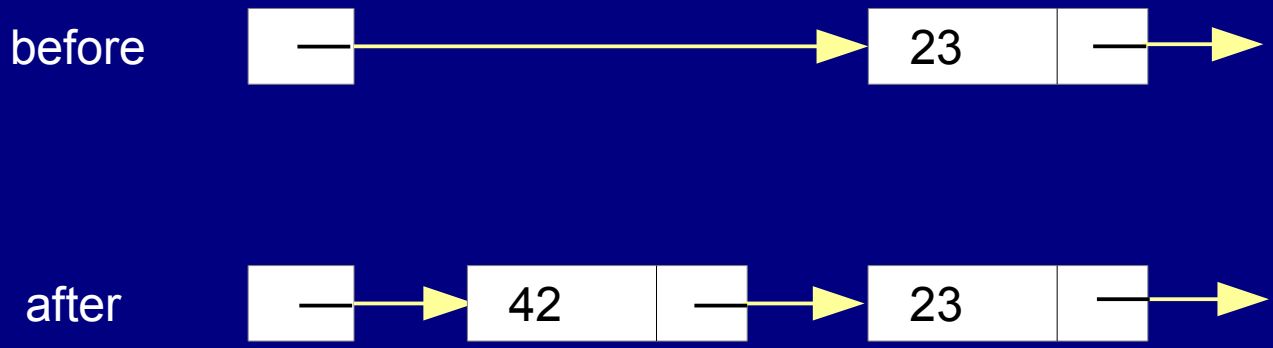
```
1 entry *numbers = NULL;
2 entry *e = malloc(sizeof(entry));
3 e->number = 2;
4 e->next = NULL;
5 numbers = e;
6 e = malloc(sizeof(entry));
7 e->number = 1;
8 e->next = numbers;
9 numbers = e;
```



# Complexity

The logic for adding a new entry is a bit complex and error-prone
The way round this is to wrap the code in a function so you can test it thoroughly and never have to write it again

```
// insert a number at the front of a list
entry *insert(entry *ns, int n) {
  entry *e = malloc(sizeof(entry));
  e->number = n;
  e->next = ns;
  return e;
}
```



# A problem

There is a problem with the insert function
You can't write

```
insert(numbers, 42);
```

Instead you have to write this

```
numbers = insert(numbers, 42);
```

because the function changes the list pointer, and it is easy to forget
It would be possible to re-write the insert function so it is called like this

```
insert(&numbers, 42);
```

but that's not very natural either

# A solution

Perhaps the best solution is to have a list type as well as an entry type

```
typedef struct entry {
  int number;
  struct entry *next;
} entry;
typedef struct list {
  struct entry *first;
} list;
```

It turns out later that this approach solves other problems too

# Rewriting insert

Here's the new version of insert

```
typedef struct entry {
  int number;
  struct entry *next;
} entry;
typedef struct list {
  struct entry *first;
} list;

void insert(list *ns, int n) {
  entry *e = malloc(sizeof(entry));
  e->number = n;
  e->next = ns->first;
  ns->first = e;
}
```

# Using the insert function

Here's a program that uses the new version of insert

```
int main() {
  list *ns = malloc(sizeof(list);
  ns->first = NULL; // { }
  insert(ns, 17);   // { 17 }
  insert(ns, 42);   // { 42, 17 }
  insert(ns, 23);   // { 23, 42, 17 }
}
```

There is still some mess on the first two lines, setting up the empty list
It is normal to write a function to create an empty list
And other functions for all the operations needed in the program
The result is a small, specialist list library, in fact an ADT (Abstract Data Type) where ***all*** the details are tucked away in functions (then you can change your mind about the details more easily)

# Searching for a number: while

Here's a function that checks whether a number is in a list

```
bool contains(list *ns, int n) {
  entry *e = ns->first;
  while (e != NULL) {
    if (e->number == n) return true;
    e = e->next;
  }
  return false;
}
```

The variable e points to each entry in turn
Using **`bool/false/true`** instead of **`int/0/1`** is a C99 feature
You need to include **`stdbool.h`**

# Searching for a number: for

This version uses a for-loop instead of a while-loop
It is easier not to forget the **`e = e->next`** with this style

```
bool contains(list *ns, int n) {
   entry *e;
   for (e = ns->first; e != NULL; e = e->next) {
      if (e->number == n) return true;
   }
   return false;
}
```

# Searching for a contact

Here's a contacts list example

```
typedef struct entry {
   char name[20], number[12]; struct entry *next;
} entry;
typedef struct list { struct entry *first; } list;

char *lookup(list *ns, char *name) {
   entry *e;
   while (e = ns->first; e != NULL; e = e->next) {
      if (strcmp(e->name, name) == 0) {
         return &e->number;
      }
   }
   return NULL;
}
```

# Adding to the end of a list

So far, only adding to the front of a list has been easy and quick
Adding to the end is also easy/quick if we keep track of the last entry
Add a last field to the list type

```
typedef struct entry {
  int number;
  struct entry *next;
} entry;
typedef struct list {
  struct entry *first, *last;
} list;
```

# newList function

Here's a function for creating an empty list

```
typedef struct entry {
  int number;
  struct entry *next;
} entry;
typedef struct list {
  struct entry *first, *last;
} list;

list *newList() {
   list *ns = malloc(sizeof(list));
   ns->first = ns->last = NULL;
   return ns;
}
```

# Insert function

Here's a function for adding an entry at the beginning

```
void insert(list *ns, int n) {
   entry *e = malloc(sizeof(entry));
   e->number = n;
   e->next = ns->first;
   ns->first = e;
   if (ns->last == NULL) ns->last = e;
}
```

# Append function

Here's a function for adding an entry at the end

```
void append(list *ns, int n) {
   entry *e = malloc(sizeof(entry));
   e->number = n;
   e->next = NULL;
   if (ns->first == NULL) {
      ns->first = e;
      ns->last = e;
   }
   else ns->last->next = e;
}
```

# Destroy function

Here's a function for deallocating a list

```
void destroy(list *ns) {
   entry *e, *remember;
   while (e != NULL) {
      remember = e->next;
      free(e);
      e = remember;
   }
   free(ns);
}
```

Note the need to remember the next field *before* freeing the entry

# Array lists versus linked lists

An array list (using a dynamic array)
and a linked list (using dynamic structures)
both do the same job

If all details are hidden inside functions
and all list operations are done by calling the functions
then the calling code doesn't need to know which has been used
*except* for efficiency

Space efficiency is similar:

- with an array list, there is "wasted" space in the unused part, which typically nearly doubles the amount of space needed
- with a linked list, there is "wasted" space in the next pointers, and in about two hidden pointers or integers per allocated structure

# Array lists versus linked lists

For a list of length $n$, the costs are:

| Operation | Linked | Array |
|---|---|---|
| Create $n$ elements | $n$ steps | $n$ steps |
| Access first element | 1 step | 1 step |
| Access element $i$ | $i$ steps | 1 step |
| Insert new first element | 1 step | $n$ steps |
| Insert new element at $i$ | $i$ steps | $n$-$i$ steps |
| Delete first element | 1 step | $n$ steps |
| Delete element at $i$ | $i$ steps | $n$-$i$ steps |