Structures and Pointers

Arrays are fixed-size structures in C

Structures and Pointers

Arrays are fixed-size structures in C

Some languages allow variably sized arrays: there is a hidden cost to this, though, in speed of access to the elements of the array

Structures and Pointers

Arrays are fixed-size structures in C

Some languages allow variably sized arrays: there is a hidden cost to this, though, in speed of access to the elements of the array

Modern programs need dynamic structures, like lists

Structures and Pointers

Arrays are fixed-size structures in C

Some languages allow variably sized arrays: there is a hidden cost to this, though, in speed of access to the elements of the array

Modern programs need dynamic structures, like lists

Lists and other dynamic datastructures (such as trees) are made easy in C by the use of structures and pointers

Structures and Pointers

We can define

```
struct intlist {
  int val;
  struct intlist *next;
};
```

This structure contains an integer value and a pointer to the next item in the list

Structures and Pointers

Exercise. Reflect for a moment why

```
struct intlist {
  int val;
  struct intlist next;
};
```

does not make sense

Structures and Pointers

We can define a few values

```
struct intlist a, b, c;
a.val = 12; a.next = &b;
b.val = 34; b.next = &c;
c.val = 56; c.next = 0;
```

(N.B. this is *not* the right way to do this kind of thing)

Structures and Pointers

We can define a few values

```
struct intlist a, b, c;
a.val = 12; a.next = &b;
b.val = 34; b.next = &c;
c.val = 56; c.next = 0;
```

(N.B. this is *not* the right way to do this kind of thing)

So `a` is the head of the list; `b` is next; then `c`

Structures and Pointers

We can define a few values

```
struct intlist a, b, c;
a.val = 12; a.next = &b;
b.val = 34; b.next = &c;
c.val = 56; c.next = 0;
```

(N.B. this is *not* the right way to do this kind of thing)

So `a` is the head of the list; `b` is next; then `c`

We conventionally terminate the list with a 0 pointer as this turns out to be useful later (think about Boolean values)

Structures and Pointers

We can define a few values

```
struct intlist a, b, c;
a.val = 12; a.next = &b;
b.val = 34; b.next = &c;
c.val = 56; c.next = 0;
```

(N.B. this is *not* the right way to do this kind of thing)

So `a` is the head of the list; `b` is next; then `c`

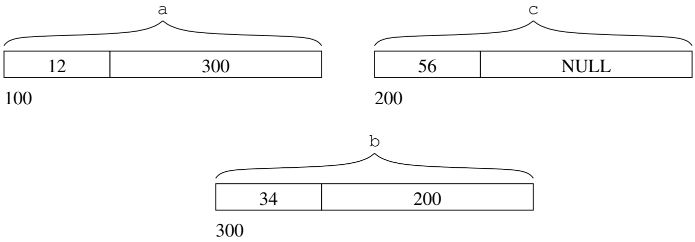
We conventionally terminate the list with a 0 pointer as this turns out to be useful later (think about Boolean values)

In fact, C defines a symbol `NULL` that is the same as zero, but visually indicates a null pointer, i.e., end of list:

```
c.next = NULL;
```

Structures and Pointers

In memory, each instance of the structure contains the value and a pointer



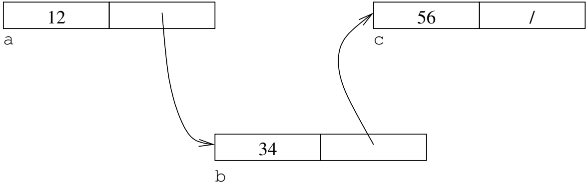
Each instance can be anywhere in memory the compiler wants to put them; they are not necessarily in the order they appear in the code or the order they are created

Structures and Pointers

Note for geeks: there will be alignment padding between the `int` and the pointer

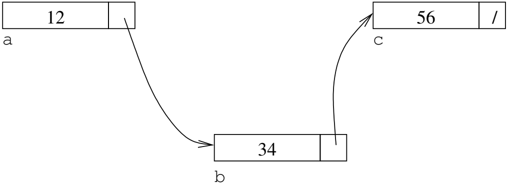
Structures and Pointers

Now, the address values are distracting and not realistic: the convention is to use *box and pointer* pictures. There are no particular values, instead arrows indicate the relationships between the boxes



Structures and Pointers

Even



if we get less representational and more relational

Structures and Pointers

Suppose we are given the head of the list, `a`

Structures and Pointers

Suppose we are given the head of the list, `a`

Getting the value in `a` is easy: just `a.val`

Structures and Pointers

Suppose we are given the head of the list, `a`

Getting the value in `a` is easy: just `a.val`

How to get the next value in the list?

Structures and Pointers

Suppose we are given the head of the list, `a`

Getting the value in `a` is easy: just `a.val`

How to get the next value in the list?

`a.next` is a pointer to `b`, so we need `*(a.next)` to follow the pointer to get at the object `b`; then `(*(a.next)).val` for the value in `b`

Structures and Pointers

Suppose we are given the head of the list, `a`

Getting the value in `a` is easy: just `a.val`

How to get the next value in the list?

`a.next` is a pointer to `b`, so we need `*(a.next)` to follow the pointer to get at the object `b`; then `(*(a.next)).val` for the value in `b`

This is ugly, but is such a common usage C provides the arrow -> operator, to prettify code. So `expr->val` is the same as `(*expr).val`

Structures and Pointers

Now `a.next->val` same as `(*(a.next)).val`, but easier to read

Structures and Pointers

Now `a.next->val` same as `(*(a.next)).val`, but easier to read

Further, `a.next->next->val` is the value in `c`

Structures and Pointers

Now `a.next->val` same as `(*(a.next)).val`, but easier to read

Further, `a.next->next->val` is the value in `c`

If we were perverse, we could write
`(&a)->next->next->val`

Structures and Pointers

Warning! Java uses just `obj.val` everywhere, while C uses `obj.val` and `pobj->val` for the different cases of objects and pointers to objects

Structures and Pointers

Warning! Java uses just `obj.val` everywhere, while C uses `obj.val` and `pobj->val` for the different cases of objects and pointers to objects

The two languages differ here

Structures and Pointers

Warning! Java uses just `obj.val` everywhere, while C uses `obj.val` and `pobj->val` for the different cases of objects and pointers to objects

The two languages differ here

You will get this wrong!

Structures and Pointers

Warning! Java uses just `obj.val` everywhere, while C uses `obj.val` and `pobj->val` for the different cases of objects and pointers to objects

The two languages differ here

You will get this wrong!

Fortunately, the compiler will pick up the problem and give you loads of error messages

Structures and Pointers

Use dot `.` to get at a slot in an object

Use arrow `->` to get at a slot in a pointer to an object (follow the arrow!)

Structures and Pointers

```
void printlist(struct intlist *l)
{
  struct intlist *ptr;

  for (ptr = l; ptr != NULL; ptr = ptr->next) {
    printf("%d\n", ptr->val);
  }
}
...
struct intlist l;
l.val = ...
...
printlist(&l);
```

Structures and Pointers

- We pass a pointer to the structure into `printlist`

Structures and Pointers

- We pass a pointer to the structure into `printlist`
- The pointer variable `ptr` will iterate down the items in the list

Structures and Pointers

- We pass a pointer to the structure into `printlist`
- The pointer variable `ptr` will iterate down the items in the list
- `for` loops are not just restricted to integer iteration

Structures and Pointers

- We pass a pointer to the structure into `printlist`
- The pointer variable `ptr` will iterate down the items in the list
- `for` loops are not just restricted to integer iteration
- The test for termination of loop is "`ptr != NULL`" as `ptr` is `NULL` at the end of the list

Structures and Pointers

- We pass a pointer to the structure into `printlist`
- The pointer variable `ptr` will iterate down the items in the list
- `for` loops are not just restricted to integer iteration
- The test for termination of loop is "`ptr != NULL`" as `ptr` is `NULL` at the end of the list
- The `ptr` is updated at each iteration to point to the next item in the list

Structures and Pointers

Slightly more idiomatic is to do this:

```
...
  for (ptr = l; ptr; ptr = ptr->next) {
    printf("%d\n", ptr->val);
  }
...
```

With a simpler termination condition

Structures and Pointers

Recall that 0 is treated as false in C and any non-zero value is true

Structures and Pointers

Recall that 0 is treated as false in C and any non-zero value is true

The loop will continue while there is a non-zero, i.e., non-NULL pointer next

Structures and Pointers

Recall that 0 is treated as false in C and any non-zero value is true

The loop will continue while there is a non-zero, i.e., non-NULL pointer next

This kind of trick is common in C and you will have to get used to seeing it

Structures and Pointers

We know that structures are like other types in C and can be passed to functions and returned as a result

```
struct rational {
  int num, den;
};
void printrat(struct rational a)
{
  printf("%d/%d\n", a.num, a.den);
}
...
printrat(r);
```

This works, but is more heavyweight than you probably want

Structures and Pointers

When we have

```
void printint(int n) {
  ...
}
...
printint(m);
```

the value of `m` is copied into the function and assigned to `n`

Structures and Pointers

When we have

```
void printint(int n) {
  ...
}
...
printint(m);
```

the value of `m` is copied into the function and assigned to `n`

Technically: is a *call by value* language. When calling a function values are copied into the parameters of the function

Structures and Pointers

In just the same way a `rational` will be *copied* into `printrat`

Structures and Pointers

In just the same way a `rational` will be *copied* into `printrat`

Not too bad here, but structures are generally much larger than this example

Structures and Pointers

In just the same way a `rational` will be *copied* into `printrat`

Not too bad here, but structures are generally much larger than this example

Copying large structures back and forth between functions will be very expensive

Structures and Pointers

So we typically pass the address of a structure to a function rather than (a copy of) the object

```
void printrat(struct rational *a)
{
  printf("%d/%d\n", a->num, a->den);
}
...
printrat(&r);
```

This is much more efficient, particularly as machine hardware is tuned to handle pointer-sized objects

Structures and Pointers

Exercise. Implement an `inttree` structure that contains an integer value and a left and right subtree

Exercise. Write code that prints out an `inttree`

Exercise. Explain why and when `obj.val` in Java corresponds to `obj->val` and when it corresponds to `obj.val` in C

Malloc and Free

The code

```
struct intlist a, b, c;
a.next = &b;
b.next = &c;
c.next = 0;
```

is a bit clunky, and certainly not suitable for dynamically growing lists where you don't know how many elements it's going to have in advance

Malloc and Free

The code

```
struct intlist a, b, c;
a.next = &b;
b.next = &c;
c.next = 0;
```

is a bit clunky, and certainly not suitable for dynamically growing lists where you don't know how many elements it's going to have in advance

Similarly, we might need an array of a size that we don't know in advance

Malloc and Free

The code

```
struct intlist a, b, c;
a.next = &b;
b.next = &c;
c.next = 0;
```

is a bit clunky, and certainly not suitable for dynamically growing lists where you don't know how many elements it's going to have in advance

Similarly, we might need an array of a size that we don't know in advance

Thus we need some kind of dynamic allocation of structures and arrays

Malloc and Free

Thinking in terms of memory an array is simply a chunk of bytes

Malloc and Free

Thinking in terms of memory an array is simply a chunk of bytes

As is a structure

Malloc and Free

Thinking in terms of memory an array is simply a chunk of bytes

As is a structure

Once we have a pointer to the structure or the address of the start of the array we are happy and can use that structure or array using the normal `[]` or `->`

Malloc and Free

Thinking in terms of memory an array is simply a chunk of bytes

As is a structure

Once we have a pointer to the structure or the address of the start of the array we are happy and can use that structure or array using the normal `[]` or `->`

We need something like

```
int *a = allocate_some_bytes(...);
a[7] = 42;
struct rational *r = allocate_some_bytes(...);
r->num = 7;
```

Malloc and Free

Exercise. This would not be correct:

```
int a[] = allocate_some_bytes(...);
```

Why?

Exercise. This would not be correct:

```
struct rational r = allocate_some_bytes(...);
```

Why?

Malloc and Free

Here is some (poor) code

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

int main(void)
{
  int *a;

  // allocate space for 10 integers
  a = malloc(40);

  a[7] = 42;

  return 0;
}
```

Malloc and Free

- We include `stdlib.h` to declare the type of the `malloc` function

Malloc and Free

- We include `stdlib.h` to declare the type of the `malloc` function
- The function `malloc` allocates a number of bytes from memory and returns a pointer to the start of the area allocated

Malloc and Free

- We include `stdlib.h` to declare the type of the `malloc` function
- The function `malloc` allocates a number of bytes from memory and returns a pointer to the start of the area allocated
- Where that area is in memory is up to the system and may well vary between runs of your program

Malloc and Free

- We include `stdlib.h` to declare the type of the `malloc` function
- The function `malloc` allocates a number of bytes from memory and returns a pointer to the start of the area allocated
- Where that area is in memory is up to the system and may well vary between runs of your program
- The bytes allocated will not be initialised to any particular value

Malloc and Free

- We include `stdlib.h` to declare the type of the `malloc` function
- The function `malloc` allocates a number of bytes from memory and returns a pointer to the start of the area allocated
- Where that area is in memory is up to the system and may well vary between runs of your program
- The bytes allocated will not be initialised to any particular value
- The argument `40` can of course be any computed value

Malloc and Free

This is poor code as we are assuming we know the size of an integer, 4 bytes in this case

Malloc and Free

This is poor code as we are assuming we know the size of an integer, 4 bytes in this case

Much better is to let the compiler tell us how big its integers are

Malloc and Free

This is poor code as we are assuming we know the size of an integer, 4 bytes in this case

Much better is to let the compiler tell us how big its integers are

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

Malloc and Free

This is poor code as we are assuming we know the size of an integer, 4 bytes in this case

Much better is to let the compiler tell us how big its integers are

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

The `sizeof` operator returns the size of a type in bytes

Malloc and Free

This is poor code as we are assuming we know the size of an integer, 4 bytes in this case

Much better is to let the compiler tell us how big its integers are

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

The `sizeof` operator returns the size of a type in bytes

So this will allocate enough bytes for 10 `int`s, however big they may be

Malloc and Free

Reason 2 for being poor code: we do not check the value returned from `malloc`

Malloc and Free

Reason 2 for being poor code: we do not check the value returned from `malloc`

Even though computers have masses of memory these days, we have huge amounts of data and it is simple to request more bytes than the machine can allocate

Malloc and Free

Reason 2 for being poor code: we do not check the value returned from `malloc`

Even though computers have masses of memory these days, we have huge amounts of data and it is simple to request more bytes than the machine can allocate

So `malloc` might fail. In this case it will return 0: a `NULL` pointer

Malloc and Free

Reason 2 for being poor code: we do not check the value returned from `malloc`

Even though computers have masses of memory these days, we have huge amounts of data and it is simple to request more bytes than the machine can allocate

So `malloc` might fail. In this case it will return 0: a `NULL` pointer

Well-written code always checks to see if `malloc` succeeded

Malloc and Free

```
a = malloc(n*sizeof(int));
if (a == NULL) { // failed ...
```

Malloc and Free

```
a = malloc(n*sizeof(int));
if (a == NULL) { // failed ...
```

Exercise. See how much memory you can allocate on your machine

Malloc and Free

`malloc` is particularly good when it comes to dynamic structures like lists and trees

Malloc and Free

```
struct intlist {
  int val;
  struct intlist *next;
};
struct intlist *make(int v)
{
  struct intlist *newl;
  // should check result...
  newl = malloc(sizeof(struct intlist));
  newl->val = v;
  newl->next = NULL;
  return newl;
}
...
struct intlist *l;
l = make(0);
l->next = make(1);
l->next->next = make(2);
```

Malloc and Free

We can now dynamically create a list of any length we want

Malloc and Free

We can now dynamically create a list of any length we want

Exercise. Think why (or when) the following might be better code

Malloc and Free

```
struct intlist *make(int v, struct intlist *prev)
{
  struct intlist *newl;
  // should check result...
  newl = malloc(sizeof(struct intlist));
  newl->val = v;
  newl->next = prev;
  return newl;
}
...
struct intlist *l;
l = make(0, NULL);
l = make(1, l);
l = make(2, l);
```

Malloc and Free

Exercise. Implement similar code for binary trees

Malloc and Free

Every time we call `malloc` it allocates more bytes

Malloc and Free

Every time we call `malloc` it allocates more bytes

If we only need that space for a short while, this is terribly wasteful

Malloc and Free

Every time we call `malloc` it allocates more bytes

If we only need that space for a short while, this is terribly wasteful

And the system might run out of memory

Malloc and Free

Every time we call `malloc` it allocates more bytes

If we only need that space for a short while, this is terribly wasteful

And the system might run out of memory

So we ought to release space back to the system when we are done with it

Malloc and Free

Every time we call `malloc` it allocates more bytes

If we only need that space for a short while, this is terribly wasteful

And the system might run out of memory

So we ought to release space back to the system when we are done with it

That memory is then free to be used in other ways, maybe even given back to us in a later `malloc`

Malloc and Free

```
// allocate space for n integers
a = malloc(n*sizeof(int));
...
// done with a
free(a);
// don't use a from here on!
```

Malloc and Free

The function `free` tells the system that the given chunk of memory is no longer needed by the program and is free to be reallocated to something else

Malloc and Free

The function `free` tells the system that the given chunk of memory is no longer needed by the program and is free to be reallocated to something else

- The pointer handed to `free` must be one given by `malloc`

Malloc and Free

The function `free` tells the system that the given chunk of memory is no longer needed by the program and is free to be reallocated to something else

- The pointer handed to `free` must be one given by `malloc`
- Don't call `free` more than once on a given pointer: confusion will ensue

Malloc and Free

The function `free` tells the system that the given chunk of memory is no longer needed by the program and is free to be reallocated to something else

- The pointer handed to `free` must be one given by `malloc`
- Don't call `free` more than once on a given pointer: confusion will ensue
- `a = malloc(...); free(a); a = malloc(...);` using `a` after reassigning is OK

Malloc and Free

- `free(a);` does not alter the value of `a`: it still points to that area of memory but is no longer "owned" by `a`. You should not use `a` until you have `malloc`ed it again. Some people recommend always going `free(a); a = NULL;` explicitly making sure `a` no longer points to that area of memory

Malloc and Free

Consider the bad code

```
a = malloc(n*sizeof(int));
a[7] = 42;
a = malloc(m*sizeof(int));
```

Malloc and Free

Consider the bad code

```
a = malloc(n*sizeof(int));
a[7] = 42;
a = malloc(m*sizeof(int));
```

Each `malloc` allocates a new chunk of memory

Malloc and Free

Consider the bad code

```
a = malloc(n*sizeof(int));
a[7] = 42;
a = malloc(m*sizeof(int));
```

Each `malloc` allocates a new chunk of memory

The first chunk is still allocated, but is no longer accessible by the program

Malloc and Free

Consider the bad code

```
a = malloc(n*sizeof(int));
a[7] = 42;
a = malloc(m*sizeof(int));
```

Each `malloc` allocates a new chunk of memory

The first chunk is still allocated, but is no longer accessible by the program

We have overwritten the address of the memory: it could have been anywhere, we don't know anymore

Malloc and Free

Consider the bad code

```
a = malloc(n*sizeof(int));
a[7] = 42;
a = malloc(m*sizeof(int));
```

Each `malloc` allocates a new chunk of memory

The first chunk is still allocated, but is no longer accessible by the program

We have overwritten the address of the memory: it could have been anywhere, we don't know anymore

That area of memory is now *garbage*. It takes up space but the program can't get at it

Malloc and Free

If we do this too much, then memory will fill up with inaccessible garbage, and we will probably run out

Malloc and Free

If we do this too much, then memory will fill up with inaccessible garbage, and we will probably run out

Of course, the correct thing is to call `free` on `a` before we overwrite it

Malloc and Free

If we do this too much, then memory will fill up with inaccessible garbage, and we will probably run out

Of course, the correct thing is to call `free` on `a` before we overwrite it

Or save the value of `a` in another variable `b = a;` first. Or save it in an array or a structure. And so on

Malloc and Free

If we do this too much, then memory will fill up with inaccessible garbage, and we will probably run out

Of course, the correct thing is to call `free` on `a` before we overwrite it

Or save the value of `a` in another variable `b = a;` first. Or save it in an array or a structure. And so on

The important thing is to ensure a pointer to every allocated chunk is somehow accessible (directly or indirectly) by the program and can be accessed or freed if necessary

Malloc and Free

Programs that create inaccessible areas of memory this (and there are many) are said to have a *memory leak*

Malloc and Free

Programs that create inaccessible areas of memory this (and there are many) are said to have a *memory leak*

Memory leaks often go unnoticed as programmers often test their programs on small examples: small enough that the amount of garbage is still small and `malloc` always succeeds

Malloc and Free

Programs that create inaccessible areas of memory this (and there are many) are said to have a *memory leak*

Memory leaks often go unnoticed as programmers often test their programs on small examples: small enough that the amount of garbage is still small and `malloc` always succeeds

They only discover the error when their code goes into production on big examples and then starts failing

Malloc and Free

Aside. Current operating systems clean up after you when your program exits, returning all malloced memory. Some early operating systems didn't, meaning poorly written programs could jam up the entire computer, eventually requiring a reboot

Malloc and Free

Aside. Current operating systems clean up after you when your program exits, returning all malloced memory. Some early operating systems didn't, meaning poorly written programs could jam up the entire computer, eventually requiring a reboot

Tools like `valgrind` will tell you how much memory you have malloced and not freed

Malloc and Free

`malloc` and `free` are a major source of bugs in C programs

Malloc and Free

`malloc` and `free` are a major source of bugs in C programs

- Using memory you have not `malloc`ed

Malloc and Free

`malloc` and `free` are a major source of bugs in C programs

- Using memory you have not `malloc`ed
- `free`ing memory more than once

Malloc and Free

`malloc` and `free` are a major source of bugs in C programs

- Using memory you have not `malloc`ed
- `free`ing memory more than once
- Using memory already `free`d

Malloc and Free

`malloc` and `free` are a major source of bugs in C programs

- Using memory you have not `malloc`ed
- `free`ing memory more than once
- Using memory already `free`d
- Accessing beyond the ends of the allocated space

Malloc and Free

`malloc` and `free` are a major source of bugs in C programs

- Using memory you have not `malloc`ed
- `free`ing memory more than once
- Using memory already `free`d
- Accessing beyond the ends of the allocated space
- Overwriting pointers, creating garbage

Malloc and Free

`malloc` and `free` are a major source of bugs in C programs

- Using memory you have not `malloc`ed
- `free`ing memory more than once
- Using memory already `free`d
- Accessing beyond the ends of the allocated space
- Overwriting pointers, creating garbage
- And so on

Malloc and Free

`malloc` and `free` are a major source of bugs in C programs

- Using memory you have not `malloc`ed
- `free`ing memory more than once
- Using memory already `free`d
- Accessing beyond the ends of the allocated space
- Overwriting pointers, creating garbage
- And so on

Again, `valgrind` is useful for tracing these kinds of memory errors

Malloc and Free

On the other hand, `malloc` and `free` are extremely useful in the right hands

Malloc and Free

On the other hand, `malloc` and `free` are extremely useful in the right hands

- the programmer has precise control on the allocation of memory

Malloc and Free

On the other hand, `malloc` and `free` are extremely useful in the right hands

- the programmer has precise control on the allocation of memory
- they concentrate the programmer's attention towards the efficient use of memory

Malloc and Free

On the other hand, `malloc` and `free` are extremely useful in the right hands

- the programmer has precise control on the allocation of memory
- they concentrate the programmer's attention towards the efficient use of memory
- they are reasonably fast (if whoever implemented them was a good programmer)

Malloc and Free

On the other hand, `malloc` and `free` are extremely useful in the right hands

- the programmer has precise control on the allocation of memory
- they concentrate the programmer's attention towards the efficient use of memory
- they are reasonably fast (if whoever implemented them was a good programmer)
- the programmer can tune their use to the problem in hand

Malloc and Free

Exercise. What is the bug here?

```
int a[10];
...
free(a);
```

Exercise. `malloc` and `free` are fast, but not free: they take some time (and some space) to manage memory. Find out how much of an overhead they incur on your computer

Exercise. Compare this with Java's memory management

Exercise. Look up `alloca` and dynamic stack allocation

Malloc and Free

Exercise. Deliberately write bad code that does these kinds of things. Run it and see what goes wrong. Use `valgrind` on your code

Exercise. Deliberately write good code that avoids these kinds of things