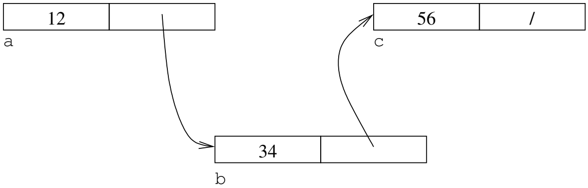
Structures and Pointers

Now the address values are distracting, not realistic and will vary depending on the compiler, runtime, and other factors.

Structures and Pointers

Now the address values are distracting, not realistic and will vary depending on the compiler, runtime, and other factors.

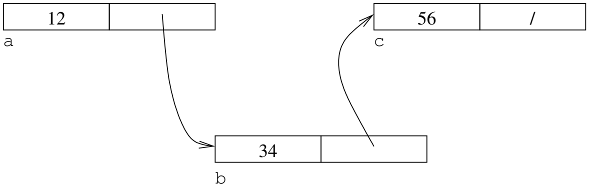
So the convention is to use *box and pointer* pictures. There are no particular values for the addresses, instead arrows indicate the relationships between the boxes



Structures and Pointers

Now the address values are distracting, not realistic and will vary depending on the compiler, runtime, and other factors.

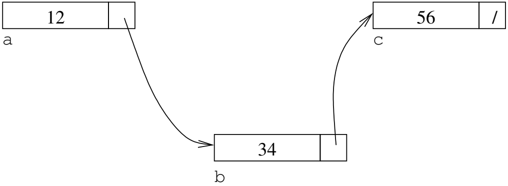
So the convention is to use *box and pointer* pictures. There are no particular values for the addresses, instead arrows indicate the relationships between the boxes



The actual locations of the structures in memory are not relevant here: but the *relationships* between the structures are

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 struct `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 struct `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->name` is the same as `(*expr).name`

Structures and Pointers

expr->name is the same as (*expr).name

Structures and Pointers

expr->name is the same as (*expr).name

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

Structures and Pointers

expr->name is the same as (*expr).name

Thus `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

expr->name is the same as (*expr).name

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

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

The first accessor is a dot, as `a` is a struct; the others are arrows as they follow pointers to structs

Structures and Pointers

expr->name is the same as (*expr).name

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

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

The first accessor is a dot, as `a` is a struct; the others are arrows as they follow pointers to structs

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

Structures and Pointers

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

Structures and Pointers

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

This is because in C quite different things are happening in the access of `val` in the two cases

Structures and Pointers

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

This is because in C quite different things are happening in the access of `val` in the two cases

The languages differ here

Structures and Pointers

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

This is because in C quite different things are happening in the access of `val` in the two cases

The languages differ here

You will get this wrong!

Structures and Pointers

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

This is because in C quite different things are happening in the access of `val` in the two cases

The 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

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

This is because in C quite different things are happening in the access of `val` in the two cases

The languages differ here

You will get this wrong!

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

Read those error messages!

Structures and Pointers

Use dot `.` to get at a slot in a struct

Use arrow `->` to get at a slot in a pointer to a struct (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:
  `for (do something; test something; do something)`

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:
  `for (do something; test something; do something)`
- 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:
  `for (do something; test something; do something)`
- 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

Exercise. Think through the following:

```
void printlistrec(struct intlist *l)
{
  if (l) {
    printf("%d\n", l->val);
    printlistrec(l->next);
  }
}
...
struct intlist l;
l.val = ...
...
printlistrec(&l);
```

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

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

Structures and Pointers

When we have

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

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

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

Structures and Pointers

When we have

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

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

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

And the *value* of the result is copied out from the function to its destination

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`

Namely a structure comprising two integers

Structures and Pointers

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

Namely a structure comprising two integers

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`

Namely a structure comprising two integers

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 (slow)

Structures and Pointers

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

```
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 things

Structures and Pointers

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

```
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 things

If I want to tell someone where you live, it is much easier to copy your address than to copy your house!

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, when and if `obj.val` in Java corresponds to `obj->val` or to `obj.val` in C

Structures and Pointers

Exercise.

```
void add1(int *arr, int len)
{
  int i;
  for (i = 0; i < len; i++) {
    arr[i]++;
  }
}
...
int vals[] = { 1, 2, 3 };
add1(vals, 3);
printf("%d %d %d\n", val[0], val[1], val[2]);
```

produces `2 3 4`. But C is a call by value language, so surely `add1` can't affect the array `vals`? What is happening here?

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 = (int*)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

  (How do we know we should use `stdlib.h`? We read the documentation for `malloc`)

Malloc and Free

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

  (How do we know we should use `stdlib.h`? We read the documentation for `malloc`)
- 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

  (How do we know we should use `stdlib.h`? We read the documentation for `malloc`)
- 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

  (How do we know we should use `stdlib.h`? We read the documentation for `malloc`)
- 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

  (How do we know we should use `stdlib.h`? We read the documentation for `malloc`)
- 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

- `malloc` returns a `void*` pointer

Malloc and Free

- `malloc` returns a `void*` pointer
- This makes it more useful: a pointer to memory with no particular type

Malloc and Free

- `malloc` returns a `void*` pointer
- This makes it more useful: a pointer to memory with no particular type
- So we have a type cast “`(int*)`” to change it to a `int*` pointer

Malloc and Free

```
a = (int*)malloc(40);
```

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

Malloc and Free

```
a = (int*)malloc(40);
```

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

```
a = (int*)malloc(40);
```

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

Malloc and Free

```
a = (int*)malloc(40);
```

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

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

Malloc and Free

```
a = (int*)malloc(40);
```

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 = (int*)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 a `NULL` pointer (0)

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 a `NULL` pointer (0)

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

Malloc and Free

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

Malloc and Free

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

Exercise. See how much memory you can allocate on your machine. Compare this with the actual amount of memory in your machine

Malloc and Free

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

Exercise. See how much memory you can allocate on your machine. Compare this with the actual amount of memory in your machine

Exercise. See what happens with

```
int *a = malloc(5*sizeof(int));
```

i.e., no type cast

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 = (struct intlist *)malloc(sizeof(struct intlist));
  newl->val = v;
  newl->next = NULL; // good practice to initialise
  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

If we need another node in the list, just call `make` (i.e., just use `malloc`) to get an allocation of memory for it

Malloc and Free

Exercise. Lists can be grown from their start, as well as their end:

```
struct intlist *l, *new;
l = make(0);
new = make(1);
new->next = l;
l = new;
new = make(2);
new->next = l;
l = new;
```

Explain why (and when) this might be better than the previous way

Malloc and Free

Exercise. Implement 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 carry on allocating regardless, eventually the system will run out of free memory to allocate

Malloc and Free

Every time we call `malloc` it allocates more bytes

If we carry on allocating regardless, eventually the system will run out of free memory to allocate

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 carry on allocating regardless, eventually the system will run out of free memory to allocate

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 = (int*)malloc(n*sizeof(int));
...
// done with a
free(a);  // a is automatically coerced to void*
// don’t use a or the memory it refers to 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 function has type `void free(void *ptr);`

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 function has type `void free(void *ptr);`
- 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 function has type `void free(void *ptr);`
- 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 function has type `void free(void *ptr);`
- 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 = (type*)malloc(...); ...  free(a); ...  ;`
  `a = (type*)malloc(...); ...  free(a); ...`
  using `a` after another `malloc` is OK

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 function has type `void free(void *ptr);`
- 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 = (type*)malloc(...); ...  free(a); ...  ;`
`a = (type*)malloc(...); ...  free(a); ...`
using `a` after another `malloc` is OK
- `malloc` and `free` should always come in pairs

Malloc and Free

- `free(a);` does not alter the value of `a`: it still points to the same area of memory but the memory 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

- `free(a);` does not alter the value of `a`: it still points to the same area of memory but the memory 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
- `free(a);` likely does not clear or otherwise modify the values in the block of memory (speed, again)

Malloc and Free

- `free(a);` does not alter the value of `a`: it still points to the same area of memory but the memory 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
- `free(a);` likely does not clear or otherwise modify the values in the block of memory (speed, again)
- `free` does not “delete memory” or “remove memory”. It’s still there: just no longer allocated to our program

Malloc and Free

The point being that `malloc` **reserves a chunk of bytes for us to use in our program**

Malloc and Free

The point being that `malloc` **reserves a chunk of bytes for us to use in our program**

Then `free` **indicates the end of that reservation**

Malloc and Free

The point being that `malloc` **reserves a chunk of bytes for us to use in our program**

Then `free` **indicates the end of that reservation**

The system may then do anything it like with that chunk of bytes

Malloc and Free

The point being that `malloc` **reserves a chunk of bytes for us to use in our program**

Then `free` **indicates the end of that reservation**

The system may then do anything it like with that chunk of bytes

The system may do nothing at all

Malloc and Free

The point being that `malloc` **reserves a chunk of bytes for us to use in our program**

Then `free` **indicates the end of that reservation**

The system may then do anything it like with that chunk of bytes

The system may do nothing at all

If you give up your reserved seat on a train, you should not be surprised to find someone else sitting there!

Malloc and Free

```
a = (int*)malloc(4*sizeof(int));
...
free(a);
...
a[0] = 1;
printf("value = %d\n", a[2]);
```

Bad!

Malloc and Free

```
a = (int*)malloc(4*sizeof(int));
...
free(a);
...
a[0] = 1;
printf("value = %d\n", a[2]);
```

Bad!

The assignment accesses the same chunk of memory: it's still there, but potentially has been allocated to some other purpose. The program may produce correct results, or not, or crash etc.

Malloc and Free

```
a = (int*)malloc(4*sizeof(int));
...
free(a);
...
a[0] = 1;
printf("value = %d\n", a[2]);
```

Bad!

The assignment accesses the same chunk of memory: it’s still there, but potentially has been allocated to some other purpose. The program may produce correct results, or not, or crash etc.

Use `valgrind` or a similar tool to check for this