Error Handling

None. E.g., early Fortran

Bits and Pieces

Errors

Error Handling

None. E.g., early Fortran

Exercise Read about IEEE *Not a Number* (NaN)

Bits and Pieces

Error Codes

Error codes. E.g., C

Bits and Pieces

Error Codes

Error codes. E.g., C

Functions return error codes, e.g., special values like 0 or -1 to indicate an error, and other values are the returned value

Bits and Pieces

Error Codes

Error codes. E.g., C

Functions return error codes, e.g., special values like 0 or -1 to indicate an error, and other values are the returned value

For example, the POSIX file `read` function returns the number of bytes read; or -1 in the case of an error

Bits and Pieces

Error Codes

Error codes. E.g., C

Functions return error codes, e.g., special values like 0 or -1 to indicate an error, and other values are the returned value

For example, the POSIX file `read` function returns the number of bytes read; or -1 in the case of an error

```
n = read(...);
if (n < 0) { ...error case... }
else { ...use data... }
```

Bits and Pieces

Error Codes

However, such codes are easily (and often) ignored, leading to buggy code

Bits and Pieces

Error Codes

However, such codes are easily (and often) ignored, leading to buggy code

```
n = read(...);
...use data...
```

Bits and Pieces

Error Codes

However, such codes are easily (and often) ignored, leading to buggy code

```
n = read(...);
...use data...
```

Or even:

```
read(...);
...use data...
```

where the programmer doesn’t even check that the `read` read the right number of bytes

Bits and Pieces

Error Codes

However, such codes are easily (and often) ignored, leading to buggy code

```
n = read(...);
...use data...
```

Or even:

```
read(...);
...use data...
```

where the programmer doesn't even check that the `read` read the right number of bytes

Error values are chosen by convention and not enforced by the language

Bits and Pieces

Error Codes

You might find conventions like:

- 0 is success, while non-0 indicates an error (and indicates what kind of error)

Bits and Pieces

Error Codes

You might find conventions like:

- 0 is success, while non-0 indicates an error (and indicates what kind of error)
- non-0 is success, while 0 indicates an error

Bits and Pieces

Error Codes

You might find conventions like:

- 0 is success, while non-0 indicates an error (and indicates what kind of error)
- non-0 is success, while 0 indicates an error
- non-negative is success, while negative indicates an error (and indicates what kind of error)

Bits and Pieces

Error Codes

You might find conventions like:

- 0 is success, while non-0 indicates an error (and indicates what kind of error)
- non-0 is success, while 0 indicates an error
- non-negative is success, while negative indicates an error (and indicates what kind of error)
- and so on for other return types

Bits and Pieces

Error Codes

You might find conventions like:

- 0 is success, while non-0 indicates an error (and indicates what kind of error)
- non-0 is success, while 0 indicates an error
- non-negative is success, while negative indicates an error (and indicates what kind of error)
- and so on for other return types

You have to read the documentation for the functions you are using to determine how errors are coded

Bits and Pieces

Error Codes

You might find conventions like:

- 0 is success, while non-0 indicates an error (and indicates what kind of error)
- non-0 is success, while 0 indicates an error
- non-negative is success, while negative indicates an error (and indicates what kind of error)
- and so on for other return types

You have to read the documentation for the functions you are using to determine how errors are coded

Widely use in real code, e.g., see the POSIX standard

Bits and Pieces

Error Codes

Sometimes it is hard to pick a special value to return to indicate an error, e.g., in the case all integer values are possible returned values

Bits and Pieces

Error Codes

Sometimes it is hard to pick a special value to return to indicate an error, e.g., in the case all integer values are possible returned values

And what to do if the function needs to return a value as well as an error code?

Bits and Pieces

Error Codes

Sometimes it is hard to pick a special value to return to indicate an error, e.g., in the case all integer values are possible returned values

And what to do if the function needs to return a value as well as an error code?

So another convention is to return a value in a pointer passed in as an argument and use the function return for purely the error code

Bits and Pieces

Error Codes

```
int foo(int arg, int *retval) {
  ...
  // ok case
  *retval = stuff; // return value
  return 0; // indicate ok
  ...
  // error case
  return 1; // indicate error
}
```

Bits and Pieces

Error Codes

```
int foo(int arg, int *retval) {
  ...
  // ok case
  *retval = stuff; // return value
  return 0; // indicate ok
  ...
  // error case
  return 1; // indicate error
}
```

This is called by something like

Bits and Pieces

Error Codes

```
int foo(int arg, int *retval) {
  ...
  // ok case
  *retval = stuff; // return value
  return 0; // indicate ok
  ...
  // error case
  return 1; // indicate error
}
```

This is called by something like

```
int value = 0;
...
err = foo(42, &value);
// now check to see if value is ok
if (err) { ... }
```

Bits and Pieces

Error Codes

This is more flexible than the straight error code return, but is (a tiny bit) more sophisticated to write

This is more flexible than the straight error code return, but is (a tiny bit) more sophisticated to write

There is barely more incentive to check the error, as it is still easy to ignore the error value

Bits and Pieces

Error Codes

This is more flexible than the straight error code return, but is (a tiny bit) more sophisticated to write

There is barely more incentive to check the error, as it is still easy to ignore the error value

And error values are still chosen by convention and not enforced by the language

Bits and Pieces

Error Codes

An alternative is to return an error code value in a pointer argument, and have the result as the normal returned value:

```
value = foo(42, &err);
```

Bits and Pieces

Error Codes

An alternative is to return an error code value in a pointer argument, and have the result as the normal returned value:

```
value = foo(42, &err);
```

This way at least reminds the programmer that an error is possible

Bits and Pieces

Error Codes

An alternative is to return an error code value in a pointer argument, and have the result as the normal returned value:

```
value = foo(42, &err);
```

This way at least reminds the programmer that an error is possible

Both alternatives are widely used in real code

Bits and Pieces

Error Codes

An alternative is to return an error code value in a pointer argument, and have the result as the normal returned value:
`value = foo(42, &err);`

This way at least reminds the programmer that an error is possible

Both alternatives are widely used in real code

And you can see it takes a bit of effort from the programmer to check for an error, so they often “forget” to do so

Bits and Pieces

Error Codes

An alternative is to return an error code value in a pointer argument, and have the result as the normal returned value:

```
value = foo(42, &err);
```

This way at least reminds the programmer that an error is possible

Both alternatives are widely used in real code

And you can see it takes a bit of effort from the programmer to check for an error, so they often “forget” to do so

Of course, they really should deal with errors!

Bits and Pieces

Global Error Codes

Some systems have a special global variable to indicate errors

Bits and Pieces

Global Error Codes

Some systems have a special global variable to indicate errors

For example, C uses the integer `errno`, and some functions set this to particular values to indicate an error: usually `0` indicate "no error"

Bits and Pieces

Global Error Codes

Some systems have a special global variable to indicate errors

For example, C uses the integer `errno`, and some functions set this to particular values to indicate an error: usually `0` indicate "no error"

This can be used for functions whose return values can't be used as error values

Bits and Pieces

Global Error Codes

Some systems have a special global variable to indicate errors

For example, C uses the integer `errno`, and some functions set this to particular values to indicate an error: usually `0` indicate "no error"

This can be used for functions whose return values can't be used as error values

```
value = foo(42);
if (errno != 0) { ... }
```

Bits and Pieces

Global Error Codes

Unfortunately, this removes any prompt to the programmer to check for errors: a variable that doesn't even appear in your code gets set?

Bits and Pieces

Global Error Codes

Unfortunately, this removes any prompt to the programmer to check for errors: a variable that doesn't even appear in your code gets set?

So very easy to ignore

Bits and Pieces

Global Error Codes

Unfortunately, this removes any prompt to the programmer to check for errors: a variable that doesn't even appear in your code gets set?

So very easy to ignore

`errno` is also widely used by POSIX

Bits and Pieces

Global Error Codes

Error values are chosen by convention and not enforced by the language

Bits and Pieces

Global Error Codes

Error values are chosen by convention and not enforced by the language

Also, this doesn't interact well with parallel code, where different threads may simultaneously want to set `errno` to different values

Bits and Pieces

Global Error Codes

Error values are chosen by convention and not enforced by the language

Also, this doesn't interact well with parallel code, where different threads may simultaneously want to set `errno` to different values

Exercise Read about the hacks C requires to mitigate the parallel `errno` problem

Bits and Pieces

Error Codes

Languages that can return multiple values (e.g., Go) can return a `result`, `error` pair for the same effect in a neater way

Bits and Pieces

Error Codes

Languages that can return multiple values (e.g., Go) can return a `result`, `error` pair for the same effect in a neater way

```
val, err = lotsafun(x);
```

Bits and Pieces

Error Codes

Languages that can return multiple values (e.g., Go) can return a `result`, `error` pair for the same effect in a neater way

```
val, err = lotsafun(x);
```

Though, again, it is still easy to ignore the error value

Bits and Pieces

Error Codes

Languages that can return multiple values (e.g., Go) can return a `result`, `error` pair for the same effect in a neater way

```
val, err = lotsafun(x);
```

Though, again, it is still easy to ignore the error value

```
// ignore error as my code is perfect
val, _ = lotsafun(x);
```

Exceptions. E.g., Java, Python. Changing flow of control by a jump in the error case using constructs like `try`, `catch`, `throws`, `finally`

Bits and Pieces

Exceptions

Exceptions. E.g., Java, Python. Changing flow of control by a jump in the error case using constructs like `try`, `catch`, `throws`, `finally`

Verbose and infectious: an error case deep in the code can bubble up (via `throws`) and need to be treated (or explicitly ignored: bad practice) in many other areas of code

Bits and Pieces

Exceptions

Exceptions. E.g., Java, Python. Changing flow of control by a jump in the error case using constructs like `try`, `catch`, `throws`, `finally`

Verbose and infectious: an error case deep in the code can bubble up (via `throws`) and need to be treated (or explicitly ignored: bad practice) in many other areas of code

Quite often you have to deal with an exception in code that is far removed from its cause, so the programmer has less of a clue about what caused the error

Bits and Pieces

Exceptions

Some languages require you to notate explicitly where exceptions might occur:

```
int foo(int n) throws BadException {
  ... something that can cause a BadException ...
}
```

or something that calls something that calls something that calls something . . . that can cause a `BadException`

Bits and Pieces

Exceptions

Some languages require you to notate explicitly where exceptions might occur:

```
int foo(int n) throws BadException {
  ... something that can cause a BadException ...
}
```

or something that calls something that calls something that calls something . . . that can cause a `BadException`

Here the good thing is that the compiler enforces checking for errors: whenever we use `foo` we have to write code to deal with the possibility of the `BadException`

Bits and Pieces

Exceptions

Some languages require you to notate explicitly where exceptions might occur:

```
int foo(int n) throws BadException {
  ... something that can cause a BadException ...
}
```

or something that calls something that calls something that calls something . . . that can cause a `BadException`

Here the good thing is that the compiler enforces checking for errors: whenever we use `foo` we have to write code to deal with the possibility of the `BadException`

No matter how deeply nested or far away is the code that does the `BadException`

Bits and Pieces

Exceptions

Used widely (Java), but often hated due to the overhead of the exception mechanism (code runs more slowly) and the non-local flow of control through various `catch` blocks

Bits and Pieces

Exceptions

Used widely (Java), but often hated due to the overhead of the exception mechanism (code runs more slowly) and the non-local flow of control through various `catch` blocks

And hated simply because it forces the programmer to deal with errors

Bits and Pieces

Exceptions

Used widely (Java), but often hated due to the overhead of the exception mechanism (code runs more slowly) and the non-local flow of control through various `catch` blocks

And hated simply because it forces the programmer to deal with errors

A common statement by programmers is “I know this code can’t go wrong, so I don’t have to deal with an error case here”

Bits and Pieces

Exceptions

Used widely (Java), but often hated due to the overhead of the exception mechanism (code runs more slowly) and the non-local flow of control through various `catch` blocks

And hated simply because it forces the programmer to deal with errors

A common statement by programmers is “I know this code can’t go wrong, so I don’t have to deal with an error case here”

Of course, this usually just demonstrates the programmer’s lack of understanding of their own code

Bits and Pieces

Exceptions

Equally, exceptions are liked by others, as they separate out error handling from the general flow of the code

Bits and Pieces

Exceptions

Equally, exceptions are liked by others, as they separate out error handling from the general flow of the code

The “everything ok case” can be written simply, while the “error case” is kept mostly separate in an error handler

Bits and Pieces

Exceptions

Equally, exceptions are liked by others, as they separate out error handling from the general flow of the code

The “everything ok case” can be written simply, while the “error case” is kept mostly separate in an error handler

Though, the positive thing is that for the most part, the compiler makes it hard for the programmer to avoid dealing with the error cases somewhere — or be explicit in ignoring errors

Equally, exceptions are liked by others, as they separate out error handling from the general flow of the code

The “everything ok case” can be written simply, while the “error case” is kept mostly separate in an error handler

Though, the positive thing is that for the most part, the compiler makes it hard for the programmer to avoid dealing with the error cases somewhere — or be explicit in ignoring errors

Thus forcing the programmer to write better code

Bits and Pieces

Exceptions

Equally, exceptions are liked by others, as they separate out error handling from the general flow of the code

The “everything ok case” can be written simply, while the “error case” is kept mostly separate in an error handler

Though, the positive thing is that for the most part, the compiler makes it hard for the programmer to avoid dealing with the error cases somewhere — or be explicit in ignoring errors

Thus forcing the programmer to write better code

Exercise But read about *unchecked* exceptions, like `RuntimeException` in Java

Comparing with error codes: exceptions give a better description of the error than error code

Comparing with error codes: exceptions give a better description of the error than error code

Exceptions can contain information about the error; or they can have types even allowing an OO approach to error management

Comparing with error codes: exceptions give a better description of the error than error code

Exceptions can contain information about the error; or they can have types even allowing an OO approach to error management

Error codes are notoriously not standardised across libraries (and often not within a single library!)

Bits and Pieces

Exceptions

Exceptions have code potentially far from the problem point: hard to fix the problem and continue from the point of error

Exceptions have code potentially far from the problem point: hard to fix the problem and continue from the point of error

Error codes you can deal with the problem directly at the point of error and continue, e.g., try again

Bits and Pieces

Exceptions

Exceptions have code potentially far from the problem point: hard to fix the problem and continue from the point of error

Error codes you can deal with the problem directly at the point of error and continue, e.g., try again

With exceptions you write the error handling code once and reuse it. Error codes need code for each place they might occur

Bits and Pieces

Result Types

Result Types. E.g., Rust

Bits and Pieces

Result Types

Result Types. E.g., Rust

A function returns a *union type*, e.g., `Result<i32,String>` that contains the result (an integer `i32`) **or** an error message (the `String`)

```
fn doit(n: i32) -> Result<i32, String> { ... }
```

Bits and Pieces

Result Types

Result Types. E.g., Rust

A function returns a *union type*, e.g., `Result<i32,String>` that contains the result (an integer `i32`) **or** an error message (the `String`)

```
fn doit(n: i32) -> Result<i32, String> { ... }
```

The important thing here is that this is not an integer value that is returned in the non-error case: you can't get at the value without actively "unwrapping" the returned `Result`

Bits and Pieces

Result Types

Result Types. E.g., Rust

A function returns a *union type*, e.g., `Result<i32,String>` that contains the result (an integer `i32`) **or** an error message (the `String`)

```
fn doit(n: i32) -> Result<i32, String> { ... }
```

The important thing here is that this is not an integer value that is returned in the non-error case: you can't get at the value without actively "unwrapping" the returned `Result`

The programmer is forced to write code to get at the returned value, so they might as well deal with the error case while they are at it

Bits and Pieces

Result Types

This again forces the code to deal with the error case or explicitly ignore it

Bits and Pieces

Result Types

This again forces the code to deal with the error case or explicitly ignore it

Not as verbose as exceptions, generally coding is a bit cleaner

This again forces the code to deal with the error case or explicitly ignore it

Not as verbose as exceptions, generally coding is a bit cleaner

Exercise Python has a `None` value you can use to indicate errors, e.g., a function returns an integer or `None`. Compare with using union types

Advanced Exercise Sometimes you see people using the word *monads* in the context of these types to get category theorists interested. Read about this

Bits and Pieces

Error Handlers

Next: Common Lisp has *error handlers*: code that executes in the context of the error, i.e., without unwinding the stack

Bits and Pieces

Error Handlers

Next: Common Lisp has *error handlers*: code that executes in the context of the error, i.e., without unwinding the stack

Roughly analogous to interrupt handlers in operating systems

Bits and Pieces

Error Handlers

Next: Common Lisp has *error handlers*: code that executes in the context of the error, i.e., without unwinding the stack

Roughly analogous to interrupt handlers in operating systems

You can throw the error, like an exception, jumping to some other remote part of the code, but more interestingly you also can *restart* from the point of error, and thereby continue after “fixing” the error

Bits and Pieces

Error Handlers

Next: Common Lisp has *error handlers*: code that executes in the context of the error, i.e., without unwinding the stack

Roughly analogous to interrupt handlers in operating systems

You can throw the error, like an exception, jumping to some other remote part of the code, but more interestingly you also can *restart* from the point of error, and thereby continue after “fixing” the error

You can even insert new value to continue with: e.g., continue from a division by 0 error with 3.14 (not a good idea)

And, of course, there are languages that mix these approaches!

Exercise Investigate the various ways Python deals with errors

We now look at different ways languages use *scopes*

Bits and Pieces

Scoping

We now look at different ways languages use *scopes*

Recall: the *scope* of a variable is the region of code where that variable can be used to refer to a particular value

Bits and Pieces

Scoping

We now look at different ways languages use *scopes*

Recall: the *scope* of a variable is the region of code where that variable can be used to refer to a particular value

```
{
  int x = 1;              S
  ...                     S
  {
    double x = 2.0;
    ...
  }
  ...                     S
}
```

Bits and Pieces

Scoping

We now look at different ways languages use *scopes*

Recall: the *scope* of a variable is the region of code where that variable can be used to refer to a particular value

```
{
  int x = 1;                  S
  ...                         S
  {
    double x = 2.0;
    ...
  }
  ...                         S
}
```

The scope of the outer version of `x` in this code is the regions of code marked by `S`

Bits and Pieces

Scoping

Note that the value of the outer `x` continues to exist, even though it cannot be referred to within the inner block

```
{
  int x = 1;               E
  ...                      E
  {                        E
    double x = 2.0;        E
    ...                    E
  }                        E
  ...                      E
}
```

Note that the value of the outer `x` continues to exist, even though it cannot be referred to within the inner block

```
{
  int x = 1;              E
  ...                     E
  {                       E
    double x = 2.0;       E
    ...                   E
  }                       E
  ...                     E
}
```

The *extent* of the value is the region marked by `E`

Bits and Pieces

Scoping

So scope is where a value is accessible through the given variable

So scope is where a value is accessible through the given variable

Extent is where a value exists

Bits and Pieces

Scoping

So scope is where a value is accessible through the given variable

Extent is where a value exists

The extent can be difficult to determine, which is why we have the various GC, reference counting and other mechanisms

Consider the (poor) code in a random language

```
int a = 23;

void foo() {
  printf("foo a = %d\n", a);
}

void bar() {
  int a = 42;
  printf("bar a = %d\n", a);
  foo();
}
```

What do you expect to see if you call `bar()`?

Bits and Pieces

Scoping

Do you expect to see

```
bar a = 42
foo a = 23
```

or

```
bar a = 42
foo a = 42
```

?

```c
int a = 23;

void foo() {
  printf("foo a = %d\n", a);
}

void bar() {
  int a = 42;
  printf("bar a = %d\n", a);
  foo();
}
```

Bits and Pieces

Scoping

Do you expect to see

```
bar a = 42
foo a = 23
```

or

```
bar a = 42
foo a = 42
```

?

```
int a = 23;

void foo() {
  printf("foo a = %d\n", a);
}

void bar() {
  int a = 42;
  printf("bar a = %d\n", a);
  foo();
}
```

In some languages you get the first, others the second

It comes down to what you think the variable `a` in `foo` refers to:

```
void foo() {
  printf("foo a = %d\n", a);
}
```

Bits and Pieces

Scoping

It comes down to what you think the variable `a` in `foo` refers to:

```
void foo() {
  printf("foo a = %d\n", a);
}
```

In some languages the `a` refers to the global `a` with value 23

Bits and Pieces

Scoping

It comes down to what you think the variable `a` in `foo` refers to:

```
void foo() {
  printf("foo a = %d\n", a);
}
```

In some languages the `a` refers to the global `a` with value 23

In other languages the `a` refers to the `a` from `bar` with value 42

Bits and Pieces

Scoping

The first, probably the more common these days, is called *lexical scoping* (also: *static scoping*), as the variable refers to the scope of `a` you can see in the code **text**, the global `a`

The first, probably the more common these days, is called *lexical scoping* (also: *static scoping*), as the variable refers to the scope of `a` you can see in the code **text**, the global `a`

The second, which used to be more common than is it now, is called *dynamic scoping*, as it refers to the scope of `a` that was most recent in the **execution** of the code, namely the local one in `bar`

The first, probably the more common these days, is called *lexical scoping* (also: *static scoping*), as the variable refers to the scope of `a` you can see in the code **text**, the global `a`

The second, which used to be more common than is it now, is called *dynamic scoping*, as it refers to the scope of `a` that was most recent in the **execution** of the code, namely the local one in `bar`

Another static vs. dynamic!

Bits and Pieces

Scoping

We can think of dynamic binding as using the latest binding in the current execution stack

Bits and Pieces

Scoping

We can think of dynamic binding as using the latest binding in the current execution stack

So, when we enter a new function we can have new dynamic bindings

Bits and Pieces

Scoping

We can think of dynamic binding as using the latest binding in the current execution stack

So, when we enter a new function we can have new dynamic bindings

When referring to a variable, we look up the execution stack to find the most recent binding for that variable

Bits and Pieces

Scoping

We can think of dynamic binding as using the latest binding in the current execution stack

So, when we enter a new function we can have new dynamic bindings

When referring to a variable, we look up the execution stack to find the most recent binding for that variable

And when we exit a function, these dynamic bindings are removed

Bits and Pieces

Scoping

With dynamic scoping, the call to `foo` is happening within the dynamic scope of the binding `a = 23`

With dynamic scoping, the call to `foo` is happening within the dynamic scope of the binding `a = 23`

When `bar` exits, that binding of `a` finishes, so a subsequent call to `foo` would refer to the global `a` or whatever the latest binding for `a` was

Bits and Pieces

Scoping

With dynamic scoping, the call to `foo` is happening within the dynamic scope of the binding `a = 23`

When `bar` exits, that binding of `a` finishes, so a subsequent call to `foo` would refer to the global `a` or whatever the latest binding for `a` was

```
bar(); foo();
```

might print

```
bar a = 42
foo a = 42
foo a = 23
```
outside of the scope (and extent) of `bar`'s `a`

Bits and Pieces

Scoping

Lisp and Perl support both lexical and dynamic scopes

Bits and Pieces

Scoping

Lisp and Perl support both lexical and dynamic scopes

Some Lisps have special forms like `(dynamic x)` vs. the simple lexical `x` to be visually distinct, helping the programmer

Bits and Pieces

Scoping

Lisp and Perl support both lexical and dynamic scopes

Some Lisps have special forms like `(dynamic x)` vs. the simple lexical `x` to be visually distinct, helping the programmer

Other Lisps (and Perl) don’t, so the programmer has to figure out what is happening for themselves: is a variable reference lexical or dynamic?

Bits and Pieces

Scoping

Lexical scope is about how the text of the program is set out

Bits and Pieces

Scoping

Lexical scope is about how the text of the program is set out

With lexical it is generally easier to understand what a variable means as you can read the code

Bits and Pieces

Scoping

Lexical scope is about how the text of the program is set out

With lexical it is generally easier to understand what a variable means as you can read the code

And most modern programmers are used to this way of doing things, though this wasn't always the case

Dynamic scope is about how the code of the program is executed

Bits and Pieces

Scoping

Dynamic scope is about how the code of the program is executed

For example, with dynamic, if you have a lot of state in variables that many functions all need to access you don't need to pass them as arguments to the function, but just use dynamic bindings for them

Bits and Pieces

Scoping

Dynamic scope is about how the code of the program is executed

For example, with dynamic, if you have a lot of state in variables that many functions all need to access you don't need to pass them as arguments to the function, but just use dynamic bindings for them

It give a kind of "local global" variable; global to this part of the execution

Bits and Pieces

Scoping

Dynamic scope is about how the code of the program is executed

For example, with dynamic, if you have a lot of state in variables that many functions all need to access you don't need to pass them as arguments to the function, but just use dynamic bindings for them

It give a kind of "local global" variable; global to this part of the execution

But code can be much harder to read for those unfamiliar with the concept

Bits and Pieces

Scoping

```
fun manipulate_window(win: Window) {
  int w = win.width; // dynamic bind sizes
  int h = win.height;
  area();
}

fun area() {
  printf("area = %d\n", w * h); // use sizes
}
```

Global or non-local state is considered dangerous these days so dynamic scope is a lot less common

Bits and Pieces

Scoping

Global or non-local state is considered dangerous these days so dynamic scope is a lot less common

Particularly in parallel code: it's unclear what behaviour we would want from a dynamic scope across threads

Global or non-local state is considered dangerous these days so dynamic scope is a lot less common

Particularly in parallel code: it’s unclear what behaviour we would want from a dynamic scope across threads

But in the right hands dynamic scoping can be very useful

Exercise With lexical binding the variable you use is not important to the execution; given code with local variable `x` you can rewrite it to use `y` (barring name clashes). What about renaming when using dynamic scoping?

Exercise With lexical binding the variable you use is not important to the execution; given code with local variable `x` you can rewrite it to use `y` (barring name clashes). What about renaming when using dynamic scoping?

Advanced Exercise Look at *thread local* values, a concept related to dynamic scope, that is in common use

Exercise With lexical binding the variable you use is not important to the execution; given code with local variable `x` you can rewrite it to use `y` (barring name clashes). What about renaming when using dynamic scoping?

Advanced Exercise Look at *thread local* values, a concept related to dynamic scope, that is in common use

Exercise Work through and understand the Perl code on the next slide

Bits and Pieces

```
$a = 23;

sub foo() { print "foo = ", $a, "\n"; }

sub barL() {
    my($a) = 42;
    print "barL = ", $a, "\n";
    &foo();
}

sub barD() {
    local($a) = 42;
    print "barD = ", $a, "\n";
    &foo();
}

print "Lex\n"; &barL(); print "Dyn\n"; &barD();
```

Exercise And this LaTeX:

```
\def\msg{there}
\def\one{hello \msg}
\def\two{\def\msg{world}\one}
\def\main{\one\two}
\main
```

And this Bash:

```
MSG=there
function one { echo hello $MSG; }
function two { MSG=world one; }
function main { one; two; }
main
```

Exercise Other kinds of scope: investigate the difference in JavaScript between declaring variables with `var` and `let`

Managed/Unmanaged

Managed/Unmanaged

Often associated with bytecode languages with VMs is the idea of a *managed* language

Managed/Unmanaged

Often associated with bytecode languages with VMs is the idea of a *managed* language

This produces code (often byte-compiled like Java and C#, but not exclusively, e.g., JavaScript and Go) that only runs under a run-time abstract machine, and not natively

Managed/Unmanaged

Often associated with bytecode languages with VMs is the idea of a *managed* language

This produces code (often byte-compiled like Java and C#, but not exclusively, e.g., JavaScript and Go) that only runs under a run-time abstract machine, and not natively

The emphasis here is that the run-time then manages memory, usually including a GC, and does security checking, e.g., on memory accesses and other resources, such as network connections

The idea that this is a “safe” language, running in a secure *sandbox*, preventing all kinds of nasty things from happening: memory overruns, execution of virus code, connecting to rogue Web sites, and so on

Bits and Pieces

Managed and Unmanaged

The idea that this is a “safe” language, running in a secure *sandbox*, preventing all kinds of nasty things from happening: memory overruns, execution of virus code, connecting to rogue Web sites, and so on

The VM runtime enforces policy on what the program is allowed to do

Bits and Pieces

Managed and Unmanaged

The idea extends to *managed data*, where (some or all of) the data is managed

Bits and Pieces

Managed and Unmanaged

The idea extends to *managed data*, where (some or all of) the data is managed

For example (a deprecated extension to) C++ (a native compiled language) allows objects to be managed or unmanaged

Bits and Pieces

Managed and Unmanaged

The idea extends to *managed data*, where (some or all of) the data is managed

For example (a deprecated extension to) C++ (a native compiled language) allows objects to be managed or unmanaged

Inaccurately and misleadingly, but to a decent approximation

$$\text{managed} = \text{bytecode}$$
$$\text{unmanaged} = \text{native compiled}$$

and the word “managed” is mostly used to make “unmanaged” sound bad by comparison

Bits and Pieces

Managed and Unmanaged

The intended perception is that

Managed: safety

Unmanaged: speed

Bits and Pieces

Managed and Unmanaged

The intended perception is that

Managed: safety

Unmanaged: speed

Though this is a false dichotomy: you can get both

Bits and Pieces

Managed and Unmanaged

Note: the concept of “managed code” was invented by Microsoft for .Net, but is now used more widely for languages that execute in a runtime that provides support, management of resources (not just memory), and safety checking of the execution