CS 3

Introduction to Software Design

Design Patterns

What is a “Design Pattern”?

Definition

A **design pattern** is a standard solution to a common programming problem.

What is a “Design Pattern”?

Definition

A **design pattern** is a standard solution to a common programming problem.

Design Patterns can be. . .

- high-level programming idioms
- techniques for making code more flexible
- shorthand for describing program design
- vocabulary for communication & documentation

What is a "Design Pattern"?

Definition

A **design pattern** is a standard solution to a common programming problem.

Design Patterns can be...

- high-level programming idioms
- techniques for making code more flexible
- shorthand for describing program design
- vocabulary for communication & documentation

You should care about them because...

- You could come up with these solutions on your own, but you shouldn't have to!
- Programming languages do not build in solutions to every problem

Problem (Polymorphism)

We want to be able to write one function for multiple types, but we have to explicitly specify a single type for each argument and return value.

Polymorphism via `void *`

Problem (Polymorphism)

We want to be able to write one function for multiple types, but we have to explicitly specify a single type for each argument and return value.

Solution (`void *`)

A `void *` pointer can be cast to point to whatever type we like. If the functions take in and return `void *`'s, it can handle pointers of any type;

Polymorphism via `void *`

Problem (Polymorphism)

We want to be able to write one function for multiple types, but we have to explicitly specify a single type for each argument and return value.

Solution (`void *`)

A `void *` pointer can be cast to point to whatever type we like. If the functions take in and return `void *`'s, it can handle pointers of any type;

Examples

- `void *malloc()` and `void free(void *)`
- `void qsort(void *base, size_t nel, size_t width, int (*compar)(const void *, const void *))`

```
1 void sort_ints(int *arr, size_t n) {
2     qsort(arr, n, sizeof(int), compare_ints);
3 }
```

```
1 void sort_strings(char **arr, size_t n) {
2     qsort(arr, n, sizeof(char *), strcmp);
3 }
```

The Output Tree ("Abstract Syntax Tree")

We can represent any mathematical expression as a tree where the root is the operation and the children are the operands.

The Output Tree ("Abstract Syntax Tree")

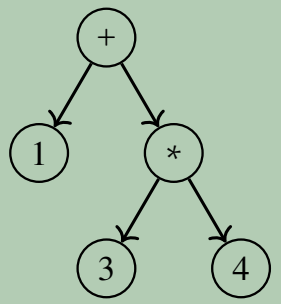
We can represent any mathematical expression as a tree where the root is the operation and the children are the operands.

For example, given $1 + 3 * 4$, we would have:

The Output Tree ("Abstract Syntax Tree")

We can represent any mathematical expression as a tree where the root is the operation and the children are the operands.

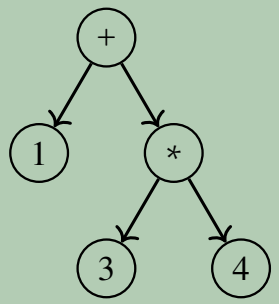
For example, given $1 + 3 * 4$, we would have:



The Output Tree ("Abstract Syntax Tree")

We can represent any mathematical expression as a tree where the root is the operation and the children are the operands.

For example, given $1+3*4$, we would have:




```
1  class Expression { }
2  class BinaryExpression { }
3  class AdditionExpression extends BinaryExpression {
4     Expression left, right;
5  }
6  class MultiplicationExpression extends BinaryExpression {
7     Expression left, right;
8  }
9  class NumberExpression extends Expression {
10    int value;
11 }
```

Problem (Inheritance)

We want to be able to model a relationship across types (e.g., a `BinaryExpression` **is a subtype of** `Expression`).

Problem (Inheritance)

We want to be able to model a relationship across types (e.g., a `BinaryExpression` **is a subtype of** `Expression`).

Solution (Casting Structs)

If two structs have the same ***beginning layout****, the larger one can be cast to the smaller one. (What?)*

Problem (Inheritance)

We want to be able to model a relationship across types (e.g., a `BinaryExpression` **is a subtype of** `Expression`).

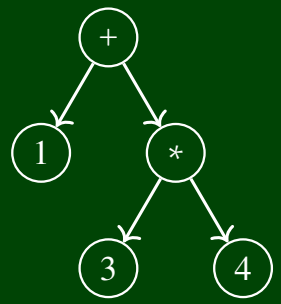
Solution (Casting Structs)

*If two structs have the same **beginning layout**, the larger one can be cast to the smaller one. (What?)*

Example

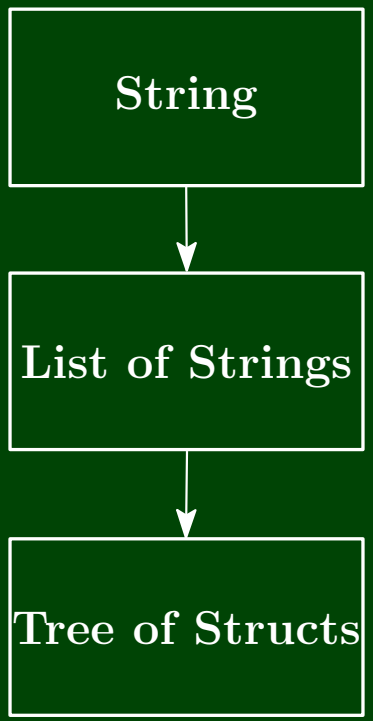
```
1  struct Int {
2     int i;
3  };
4
5  struct IntAndDouble {
6     int i;
7     double d;
8  };
9
10 struct DoubleAndInt {
11    double d;
12    int i;
13 };
```

Modelling Subclassing via Structs and Enums




```
 1 enum ExpressionType {
 2    NUMBER_EXPRESSION,
 3    ADDITION_EXPRESSION,
 4    MULTIPLICATION_EXPRESSION
 5 };
 6 struct Expression {
 7    ExpressionType type;
 8 };
 9 struct BinaryExpression {
10    ExpressionType type;
11    Expression *left, *right;
12 };
13 struct NumberExpression {
14    ExpressionType type;
15    int value;
16 };
```

How should we decompose the following program flow?



Problem (Module Decomposition)

We want to be able to separate chunks of code into independent units. This is one way of reducing complexity.

Problem (Module Decomposition)

We want to be able to separate chunks of code into independent units. This is one way of reducing complexity.

Solution (Header Files)

A header file is really just a listing of types and functions defined by the corresponding C file. We can use it as a specification for what the implementation should do.

Problem (Module Decomposition)

We want to be able to separate chunks of code into independent units. This is one way of reducing complexity.

Solution (Header Files)

A header file is really just a listing of types and functions defined by the corresponding C file. We can use it as a specification for what the implementation should do.

Example

You've seen many of these, but here's the AST example from the previous slide.

Problem (Encapsulation)

Users should not know or be able to edit our internal representation.

Problem (Encapsulation)

Users should not know or be able to edit our internal representation.

Solution (Incomplete Type Definitions)

- *Define the `typedef` in the header file, but put the actual definition inside a C file.*
- *Copy all data before returning it to the client.*

Problem (Encapsulation)

Users should not know or be able to edit our internal representation.

Solution (Incomplete Type Definitions)

- *Define the `typedef` in the header file, but put the actual definition inside a C file.*
- *Copy all data before returning it to the client.*

Example

You've seen this plenty of times. (Most notably, you've done this with `body_t`.)

Problem (Returning Multiple Values)

We would like to edit or return multiple variables in a function, but we can only have one return value.

Returning Multiple Values via Reference Arguments

Problem (Returning Multiple Values)

We would like to edit or return multiple variables in a function, but we can only have one return value.

Solution (Reference Arguments)

*Use indirection! Give the function **pointers** to the data to edit instead of the data directly. Then, it can edit those arguments if it needs to.*

Returning Multiple Values via Reference Arguments

Problem (Returning Multiple Values)

We would like to edit or return multiple variables in a function, but we can only have one return value.

Solution (Reference Arguments)

*Use indirection! Give the function **pointers** to the data to edit instead of the data directly. Then, it can edit those arguments if it needs to.*

Example

- `void swap(int *a, int *b)`
- `void divrem(int *quotient, int *remainder)`
- `void eat(char **buf, char *token)`

Problem (Error State)

The program reaches an unknown or invalid state. We need to do something to alert the user or client!

Problem (Error State)

The program reaches an unknown or invalid state. We need to do something to alert the user or client!

Solution (Three Solutions in Three Circumstances)

- `assert` *Non-Failure*
- *Print error and* `exit`
- *Return "error" value*

Problem (Error State)

The program reaches an unknown or invalid state. We need to do something to alert the user or client!

Solution (Three Solutions in Three Circumstances)

- `assert` *Non-Failure*
- *Print error and* `exit`
- *Return "error" value*

Example

Let's look at the AST example again.

Problem (Resource Management)

The program must manage resources (e.g., memory) and release them exactly once.

Problem (Resource Management)

The program must manage resources (e.g., memory) and release them exactly once.

Solution (Ownership)

- *Exactly one function should be responsible for allocation of a resource*
- *Exactly one function should be responsible for deallocation of a resource*

Problem (Resource Management)

The program must manage resources (e.g., memory) and release them exactly once.

Solution (Ownership)

- *Exactly one function should be responsible for allocation of a resource*
- *Exactly one function should be responsible for deallocation of a resource*

Example

Let's look at the AST example again.