Object-Oriented Programming

Algorithmic Decomposition

Michael L. Collard, Ph.D.

Department of Computer Science, The University of Akron

Software can be ...

- *complex*

  "has many parts"

- *complicated*

  "has a high level of difficulty"

- Need *design approaches* to deal with *complexity* and *complicated* code

Decomposition

Breaking a complex problem or system into a collection of smaller parts

- Start with the initial problem or system and apply decomposition recursively

- With good design, the smaller parts improve *modularity* which improves *maintainability* and *reusability*

- Design answers the question: *What parts do I need to solve this problem, and how do these parts relate to each other?*

- Possible tradeoffs: *efficiency*

Decomposition Advantages

When performed correctly, the smaller parts are easier to:

- Design
- Implement
- Comprehend
- Test
- Maintain
- Reuse

Algorithmic Decomposition

Breaking a complex algorithm into a collection of smaller algorithms

- Start with the initial algorithm and apply algorithm decomposition recursively

- With good design, the smaller algorithms improve *modularity* which improves *maintainability* and *reusability*

- Design answers the question: *What algorithms do I need to solve this problem, and how do these algorithms relate to each other?*

- Possible tradeoffs: *efficiency*

Rainfall: Algorithmic Decomposition

1. Input rainfall data

2. Calculate maximum rainfall

3. Calculate average rainfall

4. Output the rainfall report

Modeling Decomposed Structure

- ***Data-Flow Diagram (DFD)***

  Shows data exchanged between elements

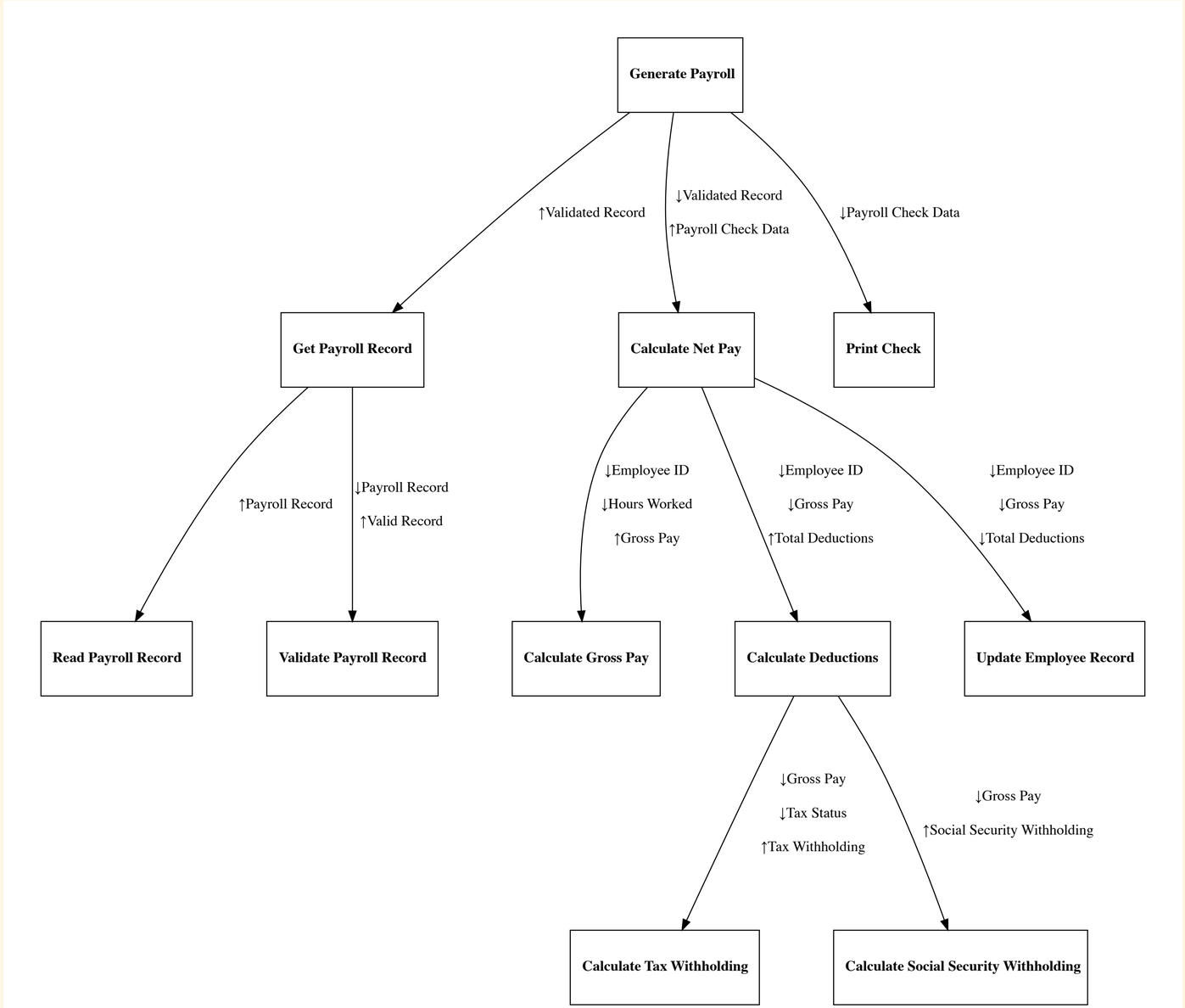
- ***Hierarchy Chart*** or ***Organization Chart***

  Shows levels of elements

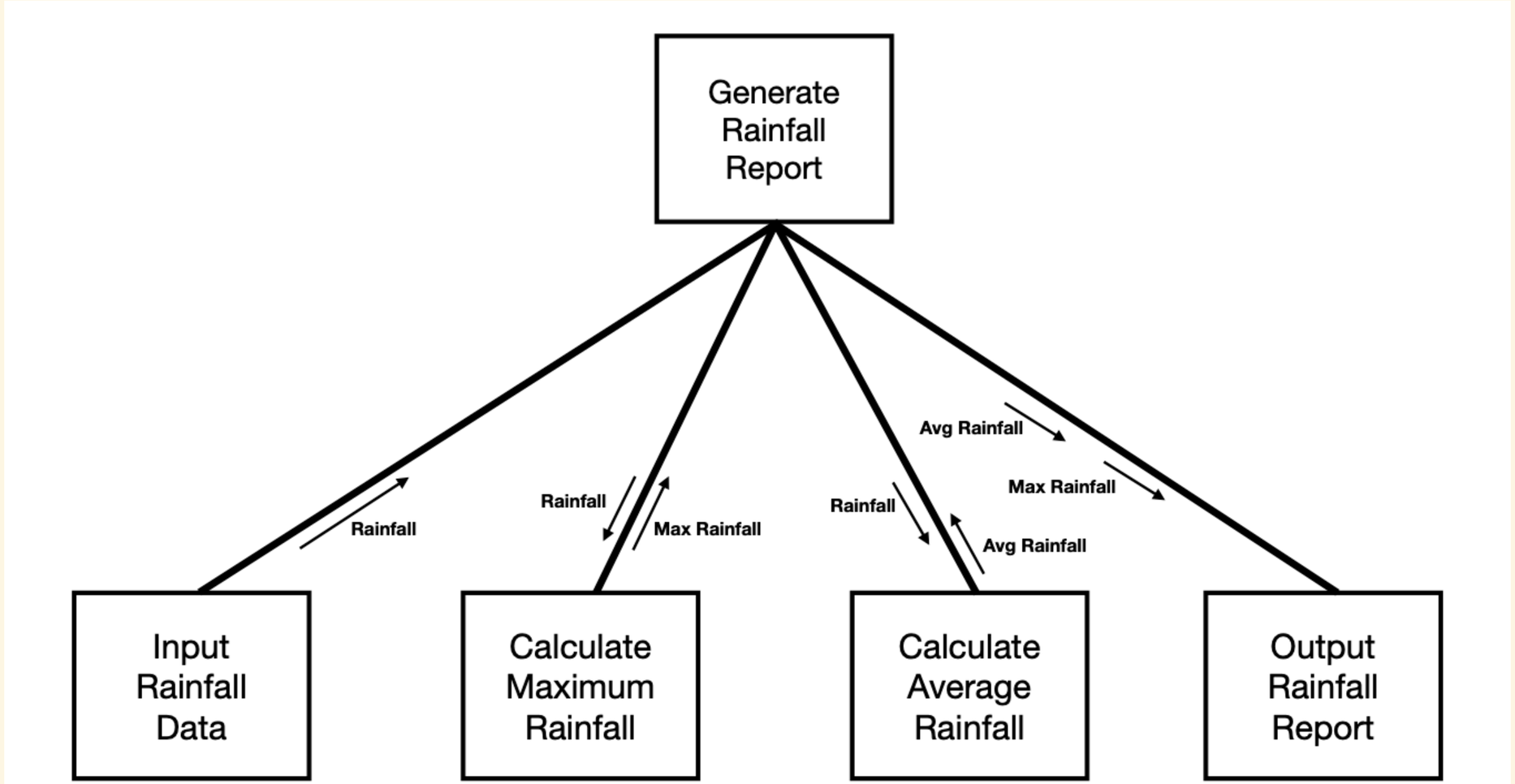
- ***Structure Chart***

  Shows levels of elements and the data exchanged between the elements

Structure Chart



Rainfall: Structure Chart



Function Decomposition

Breaking a function down into a collection of smaller functions

- Start with the main() function and apply function decomposition recursively

- With good design, the smaller functions improve *modularity* which improves *maintainability* and *reusability*

- Design answers the question: *What functions do I need to solve this problem, and how do these functions relate to each other?*

- Possible tradeoffs: *efficiency*

What Functions Care About and Algorithms Don't

- Input source and format

  Where did the array data come from?

- Output source and format

  Where is the data going?

- Exact syntax used

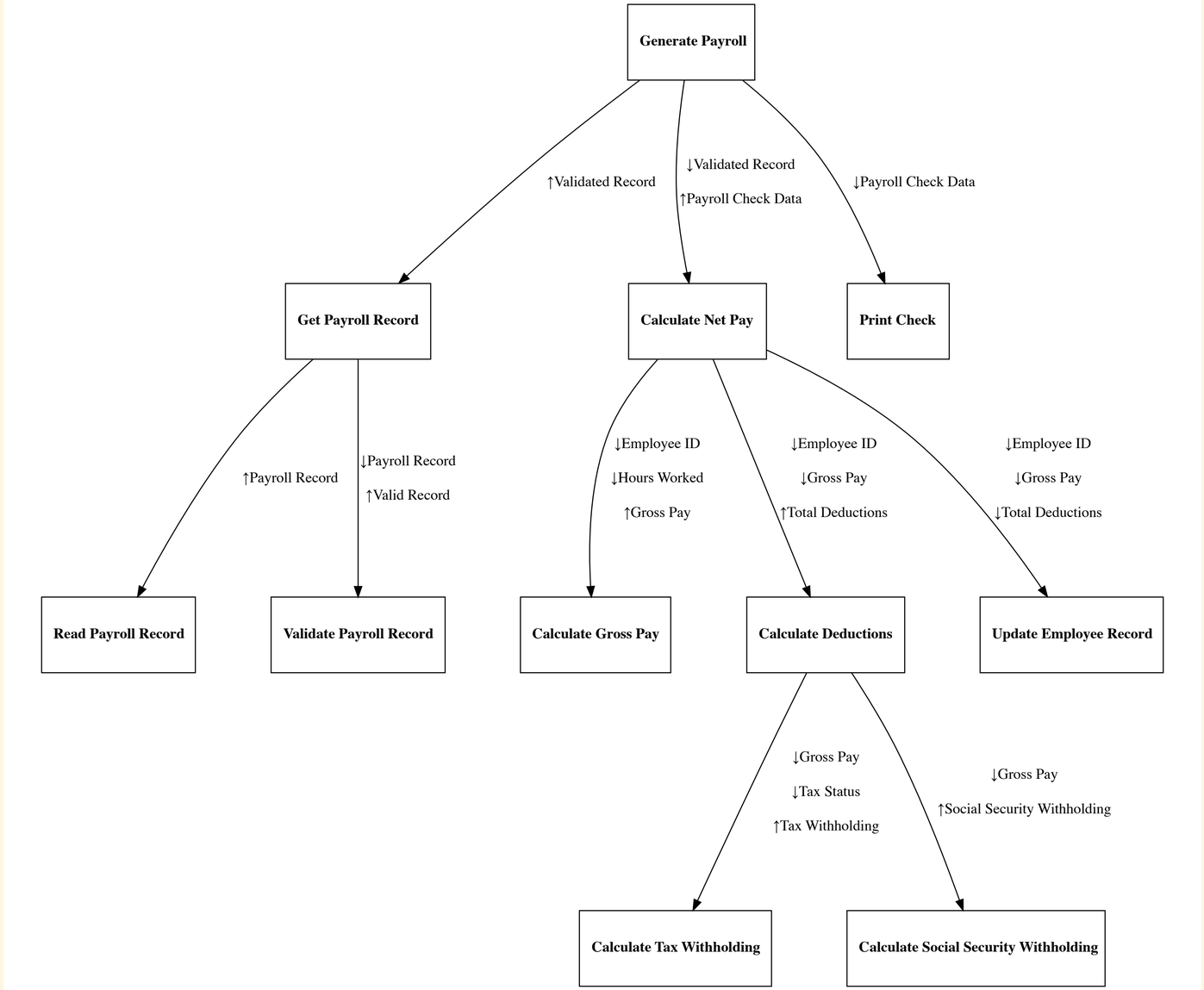
  What language features do we use?

- Interface

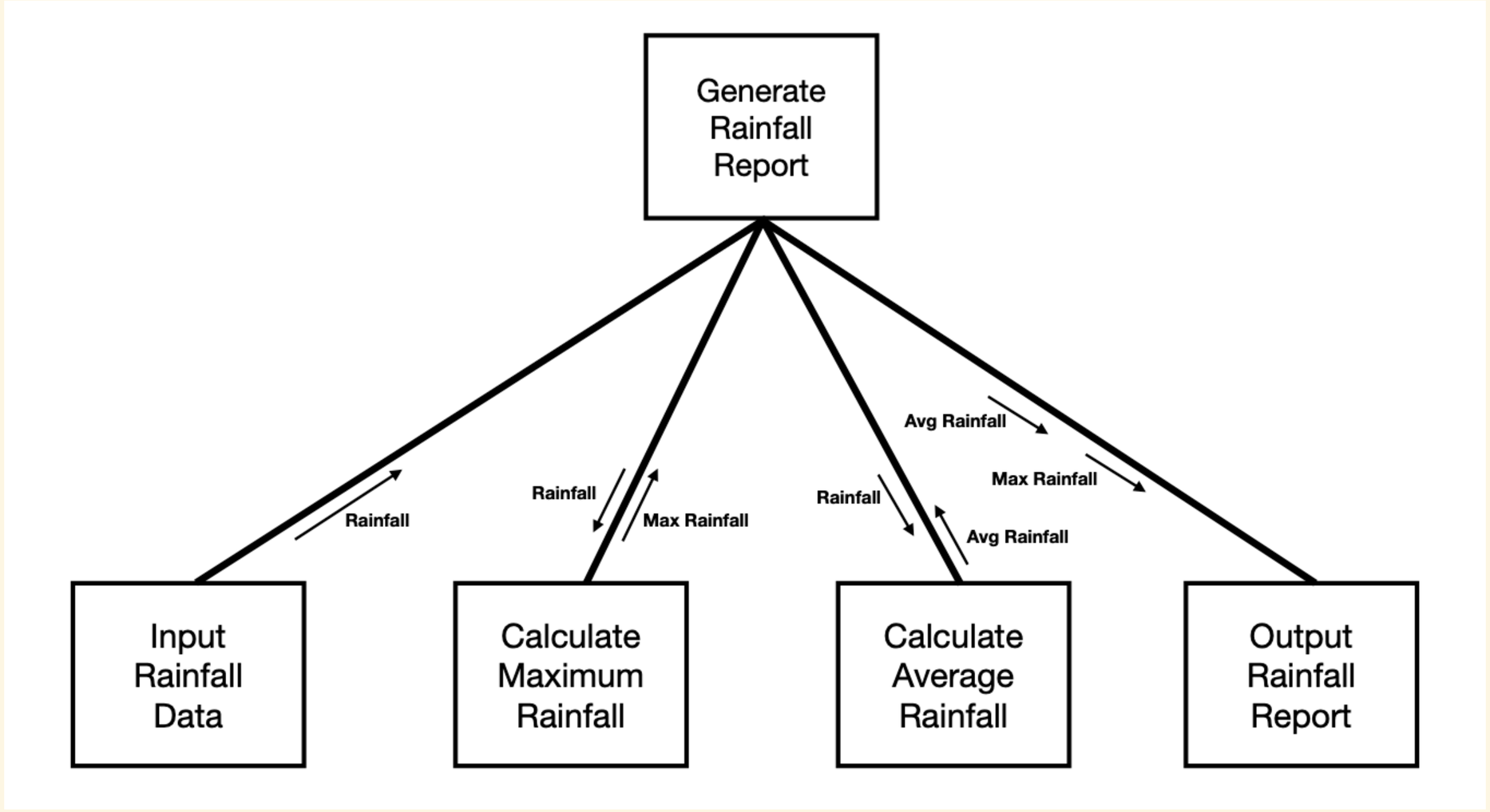
  What name, parameters, and return type do the functions have?

- Options and exceptional cases

Structure Chart



Rainfall: Structure Chart



Information Hiding

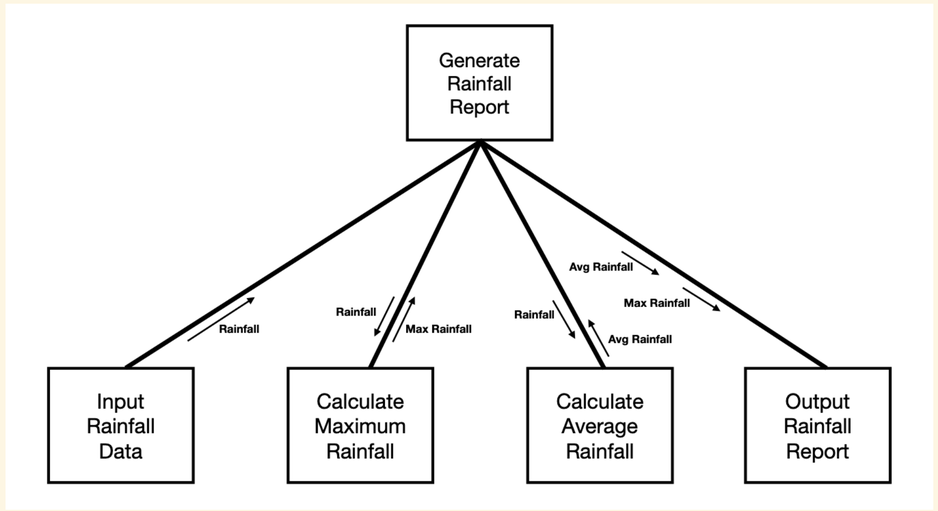
Hide the internal details of a part, algorithm, or function, and only expose the minimal interface

- *details* can be specific values, complex operations, complex series of operations, knowledge, etc.

- *interface* includes what we pass and the direction, i.e., IN, OUT, and IN/OUT

- This is what gives decomposition its advantages

Relation to *functions*



- Functions created by algorithmic decomposition work only on the data in the parameters given (IN, OUT, and IN/OUT)

- These functions are operations, actions, etc., on some data

Interface and Functions

- *Name*

- *Parameters*

- *Return Type*

Direction and C++ Parameter Types

Direction	Example	Type
IN	**`void f(int data);`**	**value**
	`void f(const Data& data);`	**const reference**
	`void f(const Data* data);`	const pointer
OUT	**`void f(int& data);`**	**reference**
	`void f(Data& data);`	**reference**
	`void f(Data* data);`	pointer
	`void f(Data** data);`	pointer to pointer
	`int f();`	**return**
	`Data f();`	**return**
	`const Data& f();`	return
IN/OUT	**`void f(int& data);`**	**reference**
	`void f(Data& data);`	**reference**
	`void f(Data* data);`	pointer
	`void f(Data** data);`	pointer to pointer

C++ Parameter Types and Direction

Type	Example	Direction
value	`void f(int data);`	IN
const reference	`void f(const Data& data);`	IN
reference	`void f(int& data);`	OUT
	`void f(Data& data);`	OUT
	`void f(int& data);`	IN/OUT
	`void f(Data& data);`	IN/OUT
const pointer	`void f(const Data* data);`	IN
pointer	`void f(Data* data);`	OUT
	`void f(Data* data);`	IN/OUT
pointer to pointer	`void f(Data** data);`	OUT
	`void f(Data** data);`	IN/OUT
return	`int f();`	OUT
return	`Data f();`	OUT
return	`const Data& f();`	OUT

functions and State

```cpp
#include <string>
#include <vector>

/*
    Split a CSV line into a container of strings

    @param[in] str The CSV string to split
    @return The container of parts of the string
*/
std::vector<std::string> splitCSV(const std::string& line) {

    const char SPLIT_TOKEN = ',';

    // extract all fields
    std::vector<std::string> fields;
    std::size_t position = 0;
    while (position < line.size()) {

        // find the postion of the split token
        const auto splitPosition = line.find(SPLIT_TOKEN, position);

        // when the split token does not exist, then the rest of the
        // string is the last element
        if (splitPosition == std::string::npos) {
            fields.push_back(line.substr(position));
            break;
        }

        // the current field is from the last position to this one
        fields.push_back(line.substr(position, splitPosition - position));

        // move after the current position
        position = splitPosition + 1;
    }

    return fields;
}
```

- In general, free functions do not save *state* between calls; they do not have any internal "memory" between calls
- Parameters and local variables are created when the function is called and destroyed when the function call returns
- Mostly exhibit consistent behavior, i.e., if passed same data, produces the same result
- There are ways around this but use object-oriented approaches instead

Approaches to Decomposition

- *Top-Down Design*

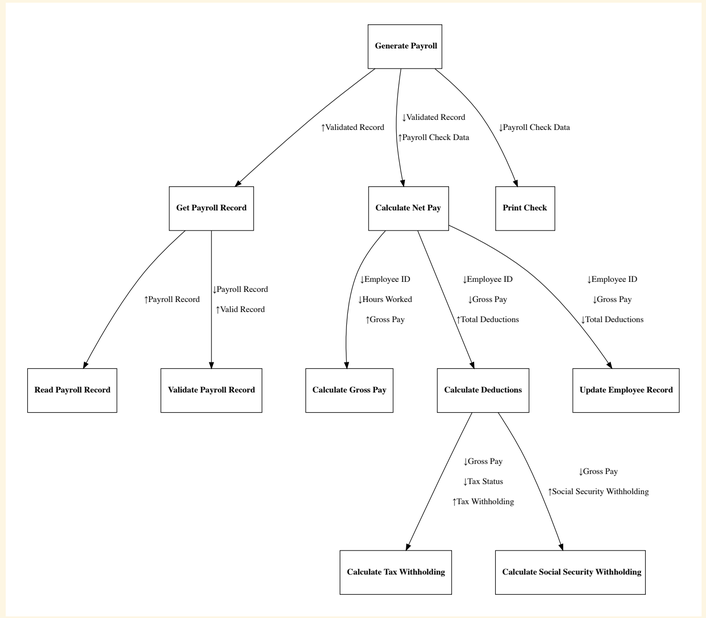
  Start with the root of the tree and work down decomposing at each level

- *Bottom-Up Design*

  Start with the leaves of the tree and combine them level-by-level

- Experts use a combination of the two and often switch during the design process

Top-Down Design



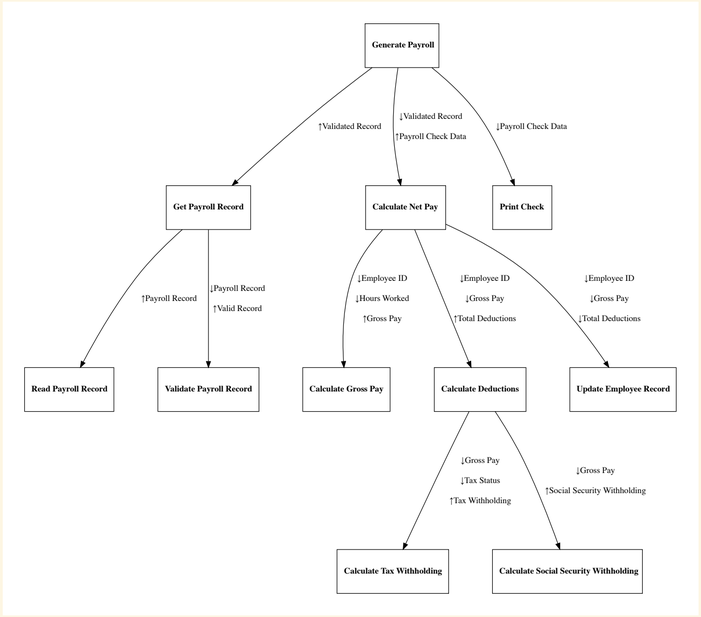
1) { *Generate Payroll* }

2) { *Get Payroll Record*, *Calculate Net Pay*, *Print Check* }

3) { { *Read Payroll Record*, *Validate Payroll Record* }, { *Calculate Gross Pay*, *Calculate Deductions*, *Update Employee Record* } }

4) { *Calculate Tax Withholding*, *Calculate Social Security Withholding* }

Bottom-Up Design



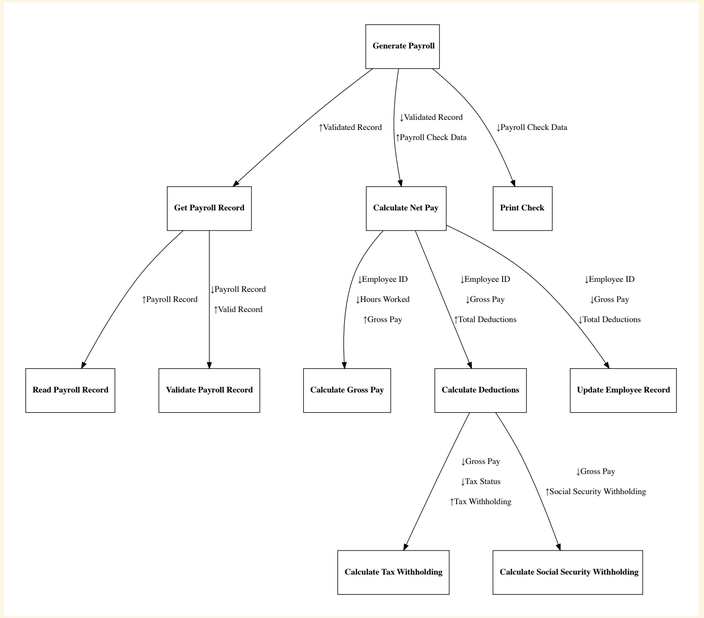
1) { *Calculate Tax Withholding, Calculate Social Security Withholding* }

2) { { *Read Payroll Record, Validate Payroll Record* }, { *Calculate Gross Pay, Calculate Deductions, Update Employee Record* } }

3) { *Get Payroll Record, Calculate Net Pay, Print Check* }

4) { *Generate Payroll* }

Algorithmic Decomposition and OOP/OOD



- A lower-level approach is needed for OOP/OOD
- Algorithmic decomposition only scales so far
- The primary challenge of modern applications is not algorithmic complexity but the sheer number of things to keep track of
- Core algorithms for a particular problem often do not change over time, but surrounding code does, e.g., input formats, output formats