A Design Recipe

EECS 230

Winter 2017
Good software design

- Correct
- Efficient
- Simple
Code isn’t just for computers

In practice, other people need to read it:
Code isn’t just for computers

In practice, other people need to read it:

- Your boss
Code isn’t just for computers

In practice, other people need to read it:

- Your boss
- Your colleagues
Code isn’t just for computers

In practice, other people need to read it:

- Your boss
- Your colleagues
- Your successors
Code isn’t just for computers

In practice, other people need to read it:

- Your boss
- Your colleagues
- Your successors
- You in the future
A recipe

1. Problem analysis
2. Header (purpose and signature)
3. Examples
4. Strategy
5. Coding
6. (Testing)
Example

Goal: Write a function that sums a vector of doubles.
Step 1: Problem analysis
Step 1: Problem analysis

We need a function that takes a `vector<double>` and returns a double.
// Sums a vector of doubles
double sum(vector<double> doubles)
Step 3: Examples

// Sums a vector of doubles

// Examples:
// - sum({}) == 0
// - sum({1, 2, 3, 4}) = 10

double sum(vector<double> doubles)
Step 4: Strategy

// Sums a vector of doubles

// Examples:
// - sum({}) == 0
// - sum({1, 2, 3, 4}) = 10

// Strategy: structural iteration
double sum(vector<double> doubles)
{
    ...
    for (double d : doubles)
        ... d ...
    ...
}
Step 5: Coding

// Sums a vector of doubles

// Examples:
// - sum({}) == 0
// - sum({1, 2, 3, 4}) = 10

// Strategy: structural iteration
double sum(vector<double> doubles)
{
    double result = 0;
    for (double d : doubles)
        result += d;
    return result;
}
Strategies

structural iteration  iterate over an existing vector
Strategies

- **structural iteration**: iterate over an existing vector
- **generative iteration**: iterate producing results while some condition holds
Strategies

structural iteration iterate over an existing vector

 generative iteration iterate producing results while some condition holds

domain knowledge translate non-programming knowledge into code
Strategies

structural iteration  iterate over an existing vector

generative iteration  iterate producing results while some condition holds

domain knowledge  translate non-programming knowledge into code

function composition  combine other functions to get the desired result
Strategy: structural iteration

```cpp
result fun(vector<T> v, ...) {
    ...
    ...
    for (T a : v) {
        ...
        ...
    }
    ...
}
```
vector<T> fun(...) {
    vector<T> result;
    while (...) {
        ... result.push_back(...) ...
    }
    return result;
}
Separation of concerns

- Input
- Computation
- Output
Separation of concerns

Input \rightarrow Computation \rightarrow Output

data \quad data
Data must be structured

Bits without structure are meaningless

Two most basic data structures:

- struct
- vector
What they are

- a struct creates a new type of compound of box made of smaller boxes
- a vector is a sequence of any number of boxes of the same type
Struct basics: declaration

To declare a new struct type:

```cpp
struct Posn
{
    double x;
    double y;
};
```
Struct basics: declaration

To declare a new struct type:

```cpp
struct Posn
{
    double x;
    double y;
};

struct Account
{
    long id;
    std::string owner;
    long balance;
};
```
Struct basics: construction

To declare and initialize a struct variable, list the values of the member variables:

```
Posn p{3, 4};
```
Struct basics: construction

To declare and initialize a struct variable, list the values of the member variables:

```
Posn p{3, 4};
```

You can also create a struct without declaring a variable:

```
Posn get_posn()
{
    double x = get_x_coordinate();
    double y = get_y_coordinate();
    return Posn{x, y};
}
```
A member variable of a struct is accessed by following the struct with a period and the name of the member variable:

```cpp
Posn p = get_posn();
std::cout << '(' << p.x << ', ' << p.y << ')';
```

If you don't initialize a struct, its fields are uninitialized:

```cpp
Posn p;
z = p.x + p.y;  // Error!
```

However, you can assign them:

```cpp
p.x = 3;
p.y = 4;
```
Struct basics: using

A member variable of a struct is accessed by following the struct with a period and the name of the member variable:

```
Posn p = get_posn();
std::cout << '(' << p.x << ',' << p.y << ')';
```

If you don’t initialize a struct, its fields are uninitialized:

```
Posn p;
z = p.x + p.y;  // Error!
```
Struct basics: using

A member variable of a struct is accessed by following the struct with a period and the name of the member variable:

```cpp
Posn p = get_posn();
std::cout << '(' << p.x << ', ' << p.y << ')';
```

If you don’t initialize a struct, its fields are uninitialized:

```cpp
Posn p;
z = p.x + p.y;    // Error!
```

However, you can assign them:

```cpp
p.x = 3;
p.y = 4;
```
Vector basics: creating

You can declare a vector with elements similar to how you declare a struct:

```cpp
#include <vector>

std::vector<int> v{2, 3, 4, 5};
```

However, it's more common to build using `push_back`:

```cpp
std::vector<int> v;
v.push_back(2);
v.push_back(1);
v.push_back(3);
```

`v` now contains 2, 1, 3.
Vector basics: creating

You can declare a vector with elements similar to how you declare a struct:

```cpp
#include <vector>

std::vector<int> v{2, 3, 4, 5};
```

However, it’s more common to build using `push_back`:

```cpp
std::vector<int> v;
    v.push_back(2);
    v.push_back(1);
    v.push_back(3);

v now contains 2, 1, 3.
```
Vector basics: size

The size member function returns the number of elements:

```cpp
for (size_t i = 0; i < v.size(); ++i) {
    std::cout << v[i] << std::endl;
}
```
Vector basics: size

The size member function returns the number of elements:

```cpp
for (size_t i = 0; i < v.size(); ++i)
    std::cout << v[i] << '\n';
```

Note! The number of elements is one more than the last index.
The **empty** member function returns whether a vector is empty:

```cpp
if (grades.empty())
    std::cout << "No grades were entered.";
```
Vector basics: access

Reverse a vector:

```cpp
for (size_t i = 0; i < v.size() / 2; ++i) {
    size_t j = v.size() - i - 1;
    int temp = v[i];
    v[i] = v[j];
    v[j] = temp;
}
```
Vector basics: iteration

Can you spot the bug?

double sum = 0.0;

for (size_t i = 0; i <= v.size(); ++i)
    sum += v[i];
Vector basics: iteration

Can’t overrun the bounds when using for-each syntax:

```
double sum = 0.0;
for (double vi : v)
    sum += vi;
```
To CLion!