Structures

Jordi Cortadella
Department of Computer Science
Structures

• Also known as **tuples** or **records** in other languages

• All components of a vector have
  – the same type (e.g. int)
  – a uniform name: vector_name[index]

• Sometimes we want a structured variable to contain
  – Components of **different** types
  – Specific **meanings** – they should have different **names**
Example: Person

// A var_person variable contains heterogeneous information on a person

struct {
    string first_name, last_name;
    int age;
    bool can_vote;
    Email email; // assuming an Email type exists
} var_person;

first_name, last_name, age, can_vote and email are the fields of var_person
// An alternative way of defining a type
// with a struct in C++

struct Person {
    string first_name, last_name;
    int age;
    bool can_vote;
    Email email;
};

Person var_person;
Structures: another example

```c
struct Address {
    string street;
    int number;
};

struct Person {
    string first_name, last_name;
    int age;
    Address address;
    bool can_vote;
};
```
Example: Person

The dot operator . selects fields of a struct variable, the same way that the [ ] operator selects components of a vector variable.

```cpp
Person p;
cin >> p.first_name >> p.last_name;
cin >> p.address.street;
cin >> p.address.number;
cin >> p.age;
p.can_vote = (p.age >= 18);
```
Structures to return results

Structs can be used in the definition of functions that return more than one result:

```c
struct Result {
    int location;
    bool found;
};

Result search(int x, const vector<int>& A);
```
Example: rational numbers

- Two components, same type, different meaning:

  ```
  struct Rational {
    int num, den;
  };
  ```

- We could also use a `vector<int>` of size 2, but we should always remember:

  "was the numerator in v[0] or in v[1]?"

- It produces uglier code that leads to errors
Invariants in data structures

• Data structures frequently have some properties (*invariants*) that must be preserved by the algorithms that manipulate them.

• Examples:

```c
struct Rational {
    int num, den;
    // Inv: den > 0
};

struct Clock {
    int hours, minutes, seconds;
    /* Inv: 0 <= hours < 24,
        0 <= minutes < 60,
        0 <= seconds < 60 */
};
```
Example: rational numbers

// Returns a + b.
Rational sum(Rational a, Rational b) {

    Rational c;
    c.num = a.num*b.den + b.num*a.den;
    c.den = a.den*b.den;
    // c.den > 0 since a.den > 0 and b.den > 0

    reduce(c); // simplifies the fraction
    return c;
}
// Post: r is in reduced form.
void reduce(Rational& r) {
    int m = gcd(abs(r.num), r.den);
    r.num = r.num/m;
    r.den = r.den/m;
    // r.den > 0 is preserved
}
Exercise: using the definition

```c
struct Clock {
    int hours;    // Inv: 0 <= hours < 24
    int minutes;  // Inv: 0 <= minutes < 60
    int seconds;  // Inv: 0 <= seconds < 60
}
```

write a function that returns the result of incrementing a Clock by 1 second.

```c
// Returns c incremented by 1 second.
Clock increment (Clock c);
```

(Note: the invariants of Clock are assumed in the specification.)
Clock increment(Clock c) {
  c.seconds = c.seconds + 1;
  if (c.seconds == 60) {
    c.seconds = 0;
    ++c.minutes;
    if (c.minutes == 60) {
      c.minutes = 0;
      ++c.hours;
      if (c.hours == 24) c.hours = 0;
    }
  }
  // The invariants of Clock are preserved
  return c;
}
Summary

• Structures can hold multiple heterogeneous data under a single variable.

• They can provide simple abstractions of complex objects.

• The generalization of structures with their own methods (functions) leads to the Object-Oriented Programming (OOP) paradigm.