Introduction to Programming (in C++)

Data structures

Jordi Cortadella, Ricard Gavaldà, Fernando Orejas

Dept. of Computer Science, UPC

Outline

- Structures
- Data structure design

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

Example: Person

```
// A better option: use a type definition

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


Person var_person;
```

Example: 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;
```

Actually, this way of declaring struct types is more common today in C++ (for reasons outside the scope of this course), even if it is inconsistent with the syntax of other type definitions (`typedef <type_definition> type_name;`)

Structures: another example

```
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

```
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:

```
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 must 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:

```
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

```
// Pre: -
// 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;
}
```

Example: rational numbers

```
// Pre:
// Post: r is in reduced form.
void reduce(Rational& r) {

    int m = gcd(abs(r.num), r.den);
    // abs returns the absolute value

    r.num = r.num/m;
    r.den = r.den/m;
    // r.den > 0 is preserved
}
```

Structures

- Exercise: using the definition

```
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.

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

(Note: the invariants of Clock are assumed in the specification.)

Structures

```
Clock increment(Clock c) {
    c.seconds = c.seconds + 1;
    if (c.seconds == 60) {
        c.seconds = 0;
        c.minutes = c.minutes + 1;
        if (c.minutes == 60) {
            c.minutes = 0;
            c.hours = c.hours + 1;
            if (c.hours == 24) c.hours = 0;
        }
    }
    // The invariants of Clock are preserved
    return c;
}
```

DATA STRUCTURE DESIGN

Data structure design

- Up to now, designing a program (or a procedure or a function) has meant designing an algorithm. The structure of the data on which the algorithm operates was part of the problem statement.

  However, when we create a program, we often need to design data structures to store data and intermediate results.

- The design of appropriate data structures is often critical:
  - to be able to solve the problem
  - to provide a more efficient solution

Data structure design

- A very influential book by *Niklaus Wirth* on learning how to program is called precisely:

  Algorithms + Data Structures = Programs

- We will study some important data structures in the next course. However, even for the programs we are trying to solve in this course, we sometimes need to know the basics of data structure design.

- Let us see some examples.

Most frequent letter

- **Problem:** design a function that reads a text and reports the most frequent letter in the text and its frequency (as a percentage). The letter case is ignored. That is:

```
struct Res {
    char letter; // letter is in ‘a’..‘z’
    double freq; // 0 <= freq <= 100
};

// Pre: the input contains a text
// Returns the most frequent letter in the text
// and its frequency, as a percentage,
// ignoring the letter case
Res most_freq_letter();
```

Most frequent letter

- The obvious algorithm is to sequentially read the characters of the text and keep a record of how many times we have seen each letter.

- Once we have read all the text, we compute the letter with the highest frequency, and report it with the frequency divided by the text length $*$ 100.

- To do this process efficiently, we need fast access to the number of occurrences of each letter seen so far.

Most frequent letter

- **Solution:** keep a vector of length N, where N is the number of distinct letters. The i-th component contains the number of occurrences of the i-th letter so far.
- **Observation:** the problem specification did <u>not</u> mention any vectors. We introduce one to solve the problem efficiently.

```
const int N = int('z') - int('a') + 1;
vector<int> occs(N, 0);
int n_letters;

// Inv: n_letters is the number of letters read
//      so far, occs[i] is the number of occurrences
//      of letter ‘a’ + i in the text read so far
```

Most frequent letter

```
Res most_freq_letter() {
    const int N = int('z') - int('a') + 1;
    vector<int> occs(N, 0);
    int n_letters = 0;
    char c;
    // n_letters contains the number of letters in the text, and occs[i]
    // contains the number of occurrences of letter ‘a’ + i in the text
    while (cin >> c) {
        if (c >= 'A' and c <= 'Z') c = c - 'A' + 'a';
        if (c >= 'a' and c <= 'z') {
            ++n_letters; ++occs[int(c) - int('a')];
        }
    }
    int imax = 0;
    // imax = the index of the highest value in occs[0..i-1]
    for (int i = 1; i < N; ++i) {
        if (occs[i] > occs[imax]) imax = i;
    }
    Res r;
    r.letter = 'a' + imax;
    if (n_letters > 0) r.freq = double(occs[imax])*100/n_letters;
    else r.freq = 0; // 0% if no letters in the text
    return r;
}
```

Most frequent word

Problem: design a function that reads a non-empty sequence of words and reports the word with highest number of occurrences.

```
struct Result {
  string word;
  double freq; // 0 < freq <= 100
};

// Pre: the input is a non-empty sequence of words
// Returns the most frequent word in the input
// and its frequency, as a percentage
Result most_freq_word();
```

Most frequent word

- The algorithm is similar to the previous one, but with one complication:
  - In the previous problem, we could create a vector with one entry for each letter, and an easy computation told us where the component for each letter was (letter c was in component int(‘c’) - int(’a’)).
  - Now, we cannot create a vector with as many components as possible words.
- Strategy: each component will contain one word with its frequency. We do not have an immediate way of knowing where each word w is stored.

Most frequent word

```
// structure to store information about
// the occurrences of a word
struct word_occ {
  string word;  // string representing the word
  int occs;     // number of occurrences
};


// type to define a vector of word occurrences
typedef vector<word_occ> list_words;
```

Most frequent word

```
Result most_freq_word() {
  list_words L;
  int total = 0;   // The number of words at the input
  string w;
  while (cin >> w) {
    ++total;
    store(L, w); // If w is not in L, it adds w to L;
                 // otherwise, its number of occurrences increases
  }

  int imax = argmax(L); // returns the index of the highest
                        // component (most frequent word)
  Result r;
  r.word = L[imax].word;
  if (total > 0) r.freq = double(L[imax].occs)*100/total;
  else r.freq = 0; // 0% in case of no words in the text
  return r;
}
```

Most frequent word

```
void store(list_words& L, string w) {
    word_occ w_occ;
    w_occ.word = w;
    w_occ.occs = 1;
    L.push_back(w_occ); // add sentinel (assume a new word)
    int i = 0;
    while (L[i].word != w) ++i;
    if (i != L.size() - 1) { // the word already existed
        ++L[i].occs;         // increase number of occurrences
        L.pop_back();        // remove the sentinel
    }
}

int argmax(const list_words& L) {
    int imax = 0;
    //Inv: imax = index of highest value in S.words[0..i-1].occs
    for (int i = 1; i < L.size(); ++i) {
        if (L[i].occs > L[imax].occs) imax = i;
    }
    return imax;
}
```

Pangrams

- A pangram is a sentence containing all the letters in the alphabet.

  An English pangram:

  The quick brown dog jumps over the lazy fox

  A Catalan pangram:

  Jove xef, porti whisky amb quinze glaçons d'hidrogen, coi!

- **Problem:** design a function that reads a sentence and says whether it is a pangram. That is,

```
// Pre: the input contains a sentence.
// Returns true if the input sentence is a pangram
// and false otherwise.
bool is_pangram();
```

Pangrams

- The algorithm is similar to previous the problem:
  - Use a vector with one position per letter as a data structure.
  - Read the sentence and keep track of the number of occurrences of each letter.
  - Then check that each letter appeared at least once.

Pangrams

```
bool is_pangram() {
    const int N = int('z') - int('a') + 1;
    vector<bool> appear(N, false);
    char c;

    // Inv: appear[i] indicates whether the letter
    //      ‘a’ + i has already appeared in the text.
    while (cin >> c) {
        if (c >= 'A' and c <= 'Z') c = c – 'A' + 'a';
        if (c >= 'a' and c <= 'z') appear[int(c) - int('a')] = true;
    }

    // Check that all letters appear
    for (int i = 0; i < N; ++i) {
        if (not appear[i]) return false;
    }
    return true;
}
```

Pangrams

- **Exercise:** design a variation of the previous algorithm without the second loop. Stop the first loop when all the letters have already appeared in the sentence.

Brackets

- A number of characters go in pairs, one used to “open” a part of a text and the other to “close” it. Some examples are:

( )	(parenthesis),
[ ]	(square brackets)
{ }	(curly brackets)
⟨ ⟩	(angle brackets)
¿ ?	(question marks - Spanish)
¡ !	(exclamation marks - Spanish)
“ ”	(double quotes)
‘ ’	(single quotes)

Brackets

The correct use of brackets can be defined by three rules:

1. Every opening bracket is followed in the text by a matching closing bracket of the same type – though not necessarily immediately.

2. Vice versa, every closing bracket is preceded in the text by a matching opening bracket of the same type.

3. The text between an opening bracket and its matching closing bracket must include the closing bracket of every opening bracket it contains, and the opening bracket of every closing bracket it contains

   (It’s ok if you need to read this more than once)

Brackets

- **Exercise:** design a function that reads a nonempty sequence of bracket characters of different kinds, and tells us whether the sequence respects the bracketing rules

`([][{}]¿¡¡!!?)[]`	answer should be true
`(([][{][}]¿¡¡!!?)[]`	answer should be false
`(([][{}]¿¡¡`	answer should be false
`([]){})`	answer should be false

Brackets

That is, we want:

```
// Pre: the input contains a nonempty sequence
//      of bracket chars
// Returns true if the sequence is correctly
// bracketed, and false otherwise.
bool brackets();
```

Suppose we use the following functions:

```
  bool is_open_br(char c); // Is c an opening bracket?
  bool is_clos_br(char c); // Is c a closing bracket?
  char match(char c);      // Returns the match of c
```

Brackets

- Strategy: keep a vector of **unclosed** open brackets.
- When we see an opening bracket in the input, we store it in **unclosed** (its matching closing bracket should arrive later).
- When we see a closing bracket in the input, either its matching opening bracket must be the last element in **unclosed** (and we can remove both), or we know the sequence is incorrect.
- At the end of the sequence **unclosed** should be empty.

Brackets

```
// Pre: the input contains a nonempty sequence of bracket chars
// Returns true if the sequence is correctly bracketed,
// and false otherwise.
bool brackets() {
  vector<char> unclosed;
  char c;
  while (cin >> c) {
    if (is_open_br(c)) unclosed.push_back(c);
    else if (unclosed.size() == 0) return false;
    else if (match(c) != unclosed[unclosed.size()-1]) return false;
    else unclosed.pop_back();
  }

  // Check that no bracket has been left open
  return unclosed.size() == 0;
}
```