Exercise Class 11

Contents

1 (Extended)-Backus-Naur-Form ((E)BNF) – *Blackboard* **1**

2 Structs **3**

2.1 Data members - *Blackboard* 3

3 Function overloading - *Blackboard* **5**

3.1 Operator overloading - *Blackboard* 7

4 Tribool - *Presentation AND/OR Programming Environment* **10**

5 Const references **10**

1 (Extended)-Backus-Naur-Form ((E)BNF) – *Blackboard*

The BNF is a recursive way to describe the set of sequences we can obtain by concatenating elements of a so-called alphabet according to some predefined rules. The BNF itself is basically a set of substitution rules which can be used to generate every allowed sequence (called: word). Let's start with a short example. *If your class is in German: Satz is used for word (i.e. possible sequence), and not Wort.*

Alphabet: `{A, a, _}`
Rules: `A` can only appear directly after an underscore or as the very first symbol. And underscores cannot occur in pairs and cannot be placed as the very first or the very last symbol.

For example, the following sequence is valid: `Aaaaa_aa`, but *not* `AaaAa`. Our task is now to come up with a BNF expressing such sequences. A BNF describing all allowed sequences is (omitting a formal proof):

```
seq = term | term "_" seq
term = "A" | "A" lowerterm | lowerterm
lowerterm = "a" | "a" lowerterm
```

We can for example obtain the word `Aaa_A` from the BNF as follows:

```
seq
-> term "_" seq
-> "A" lowerterm "_" seq
-> "Aa" lowerterm "_" seq
-> "Aaa_" seq
-> "Aaa_" term
-> "Aaa_A"
```

Notice that this BNF has 3 rules, the first one has two alternatives (`term`, `term "_" seq`), the second one has three alternatives (`"A"`, `"A" lowerterm`, `lowerterm`), and the third one has again two alternatives (`"a"`, `"a" lowerterm`). This BNF has 3 non-terminal symbols (`seq`, `term`, `lowerterm`) and 3 terminal symbols (`A`, `a`, `_`).

The EBNF extends the BNF with additional syntax allowing us to write things more compactly. In the above BNF we immediately spot one thing that is a bit suboptimal: we needed a special non-terminal symbol `lowerterm` to express the concatenations of the symbol `"a"` which may not have an `"A"` in between. The EBNF introduces the syntax `{...}`, which means that at this location the content between the brackets can be repeated an arbitrary finite number of times (including 0 times). With this we can rewrite it as

```
seq = term | term "_" seq
term = "A" { "a" } | "a" { "a" }
```

The EBNF also introduces angular brackets `[...]` meaning that the content between the brackets can be inserted either 0 times or once. With this we can further simplify our rules:

```
seq = term [ "_" seq ]
term = "A" { "a" } | "a" { "a" }
```

Questions?

Exercise 1) Valid Words

Which of the following concatenations are valid `seq`s in the sense of the above (E)BNF?

```
(i)  A,  (ii) a,  (iii) _,  (iv) Aaa,  (v) aaA,  (vi) A_A,  (vii) Aa_Aa
```

Solution 1) *Blackboard*

Valid words are (i), (ii), (iv), (vi), (vii); all of them can be obtained by suitably substituting. The words (iii) and (v) are not valid. The following aims at the intuition, but is not a formal proof. First, (iii) is not valid since _ can only be brought in if another `seq` comes right behind (first rule in the EBNF), which starts by an `"A"` or an `"a"`. Also (v) is not valid, as the only way for an `"A"` to come into the `seq` is either if it is at the very front of the very first term (where there is nothing in front of the `"A"`), or if it is inserted in the optional part in the first rule (where an `"_"` is in front of it).

Questions?

2 Structs

2.1 Data members - *Blackboard*

In the lecture we have already seen a `struct` for representing rational numbers. Struct members are initialized element-wise or if C++11 functionality is available by using initializer lists.

```
struct rational {
  int n;
  int d; // INV: d != 0
};


int main () {

  rational r; // With C++11 initializer list: r = {1, 2}
  r.n = 1;
  r.d = 2;

  return 0;
}
```

Structs can have any kind of data member, including C arrays.

```
struct strange {
  int n;
  bool b;
  int a[3];
};


int main () {

  strange x; // With C++11 initializer list: x = {1, true,{ 1,2,3}};
  x.n = 1;
  x.b = true;
  x.a[0] = 1;
  x.a[1] = 2;
  x.a[2] = 3;
  strange y = x;         // C++ can copy C arrays this way

  return 0;
}
```

If you want you can make another example for structs where you model a more real-life situation. This could additionally help the students to understand the topic.

Questions?

Exercise 2) Struct for Point and Lines

There are slides available. Just as a reminder: the students might come up with other solutions to the questions which might also work.

Questions?

3 Function overloading - *Blackboard*

It is possible for two functions to have the same name, as long as the compiler has another way to differentiate between them. The possible criteria are number and type of the arguments of the function. The following is an example for a different number of arguments.

```
int f (int a) {
 ...
}

int f (int a, int b) {
 ...
}
```

And one for different types of arguments.

```
int f (int a) {
 ...
}

int f (float a) {
 ...
}
```

It is important to notice that the name of the variables cannot be used as criterion as parameter names are not used in caller.

```
int f (int a) {
 ...
}

int f (int b) {        // error
 ...
}
```

As isn't the return type.

```
int f (int a) {
 ...
}

double f (int a) {        // error
 ...
}

void caller() {
  f(1); // call without return type
        // what function should be called?
}
```

Tip: Ask the students why these are not viable criteria and explain the logic afterwards.

Overloading can change the behavior of the program. In the following example we start off with one function `double calculation` that we are going to call in the main function.

Since the function `calculation` needs an argument of the type `double`, the integer `number` is converted implicitly when the function is called.

```
double calculation (double value) { return value / 2.0; }

int main () {
  int number;
  std::cin >> number;
  double result = calculation(number);
  std::cout << result;
  return 0;
}
```

Here we added another function with the same name which has an argument of type `int`. Note, that we did neither change the `main` function nor the `double calculation (double value)` function, but now the new function is called because the argument type matches.

```
int calculation (int value) { return value / 2; }
double calculation (double value) { return value / 2.0; }

int main () {
  int number;
  std::cin >> number;
  double result = calculation(number);
  std::cout << result;
  return 0;
}
```

Questions?

3.1 Operator overloading - *Blackboard*

For the new types we create using structs we can define our own operators (like `operator +`, `operator<`, `operator++`). We do this by overloading the already existing corresponding operators for the fundamental types.

Explain why the return value and one of the arguments of the operators `+=`, `-=`, `*=`, `/=` *need to be references once again. This was already shown during the lecture, but it is worth repeating. (It's then as the operators for fundamental types do, thus a matter of convention. Other programmers are accustomed to getting an l-value back from these operators.)*

```
rational& operator+= (rational& a, const rational b) {
  a.n = a.n * b.d + a.d * b.n;
  a.d *= b.d;
  return a;
}
```

The operators `+`, `-`, `*`, `/` can be expressed in terms of `+=`, `-=`, `*=`, `/=`.

```
rational operator+ (rational a, const rational b) {
  return a += b;
}
```

We do this for to reduce redundancy: If we were to change the data members of a struct, e.g., from:

```
struct rational {
int n;
int d; // INV: d != 0
};
```

to something like:

```
struct rational {
bool sign;
unsigned int n;
unsigned int d; // INV: d != 0
};
```

we would have to change just the operators `+=`, `-=`, `*=`, `/=` to accommodate for the changed struct. This applies as well to the relational operators.

Another interesting pair of operators are the pre-increment and post-increment operators. Let's say we wanted to implement `++r` and `r++` for rational numbers, since they are also implemented for `double` and `float`. One of the two is probably named `operator++`. But what about the other one; is it called `++operator`? The standard forbids key words to start with a `+`, so that is not possible.

The compiler is able to differentiate between the two using operator overloading. They are both called `operator++`, but one of them (the post increment operator) has an additional dummy argument `int i` so the compiler is able to tell which one to use. It is called a dummy argument, because `i` is never used in the function body.

```
rational& operator++ (rational& r) {
  rational s = {1,1};
  return r += s;
}

rational operator++ (rational& r, int i) {
  rational s = {1,1};
  rational r_0 = r;
  r += s;
  return r_0;
}
```

Again we used `operator+=` to reduce redundancy (instead of adding the denominator to the numerator or something like that).

Questions?

The operator `operator<<` is also worth to be mentioned. For example:

```
std::ostream& operator<< (std::ostream& o, rational r) {
    o << r.n << "/" << r.d;
    return o;
}
```

Here `std::ostream` is an output stream just like `std::cout`. Thus the `operator<<` allows us to call: `std::cout << my_rat << ....`

Notice that in `o << r.n << "/ "<< r.d` we are just using the usual `<<` operator for `int`.

Questions?

4 Tribool - *Presentation AND/OR Programming Environment*

The main idea here is to give the students a larger example where they themselves have to reproduce the steps from a concept described in words to a fully-fledged struct. This is a bit like the `rational` *example from the lecture, but this time it's wrapped up as a step-by-step exercise. You could consider to briefly mention a couple of practical applications for Tribool; this might make the theoretical concept a bit easier to digest for them.*

There are multiple ways to approach this topic in your exercise class. One way is to show them the presentation and let them first implement the tasks themselves, and then discuss their results together using the sample solutions in the slides. Another way is to entirely solve this exercise together with the students in the Programming Environment. A third way is to show them the questions using the presentation, but implement the answers together with the students in the Programming Environment. And if neither of these approaches suits your needs you could maybe even come up with a fourth way to cover Tribool in your exercise class. :-)

Also notice that you don't have to cover the entire exercise if you don't have the time. A good alternative is to just cover parts a) and b).

Questions?

5 Const references

References too can be `const`, but what does it mean when they are? First of all, if the reference to a variable is `const`, the variable itself can still change (but not through this reference).

```
int a = 5;
int& b = a;
const int& c = a;

c++;  // runtime error, a cannot be changed through const reference c
b++;  // a is now 6, a can still be changed through other non-const references
a++;  // a is now 7, a can still change by itself
```

So the const in front of a reference just means that the variable cannot be changed through that very reference. It can be read as a promise not to change the variable from the angle of that particular reference. It can still be changed through any other non-const reference.

A good reason for calling a function by const reference is that you can be sure the function you are calling is not going to change the variable you passed by reference. As an example, let's say we have a function that is supposed to just output a double result we calculated before.

```
void print_result (const double& result) {
  std::cout << result;
}
```

Because of the const, we know for sure that this function can not change result basically guaranteeing us that, if there is a bug somewhere, it is not here.

Another fact is that, in contrast to non-const references, we can initialize const references with an r-value, a literal.

```
const int& a = 5;
```

When this happens we can imagine that the compiler creates an int variable, puts 5 into it and then creates a const reference pointing to that very int variable. This is very useful if we think about functions again. It allows us to write functions that accept l-values passed as const references and normal r-values. So a const reference as an argument enables a function to do one of the two, const call by reference and call by value.

Questions?