Lösningsförslag, Tentamen i C++

2024-01-10

1. **a**. The constructor does not initiate the variable `a`, it *assigns* it in the function body. As the object must be completely constructed before the function body is executed, there is an implicit call of the default constructor `A::A()`, but that does not exist.

b. The problem can be fixed in `A` by giving it a default constructor. Two examples:

```
A(int x=0) {val = x;}
```

or

```
A() = default;
A(int x) {val = x;}
```

Here, assigning in the function body will work, but you should always write constructors with initiaisation(i.e., `A(int x=0) :val{x} {}`) when possible.

c. To solve the problem in `B`, the member `a` is initialised instead of assigned:

```
B(int x) :a{A(x)} {}
```

2. In `use2` a pointer to a stack allocated variable is passed, and the difference is that this is not an *owning pointer*, so `consume_number` then does `delete` on a stack allocated variable, which is wrong. In `use`, an owning pointer to a dynamically allocated `int` is passed, and the program works as expected.

The error is caused by mixing models for ownership – in `use2` it is attempted to transfer ownership of a variable with automatic storage duration, which is not allowed.

3. a) To check if they refer to the same object, compare the addresses: `if(&a == &b)...`

b) To check if they have the same value, use `if(a == b)...`

c) As there is no common superclass to all types in C++, a function template must be used. The following function template covers the simple case where both arguments are of the same type, and `operator==` is defined for the type,
The function must have reference parameters to enable checking for reference equality.

```
template <typename A>
void compareObjects(const A& a, const A& b) {
  if( &a == &b) {
    std::cout << "a and b is the same object\n");
  }

  if( a == b) {
    std::cout << "the values of a and b are equal\n");
  }
}
```

4. a)
```
#include <iterator>
#include <vector>

template <typename T>
std::vector<T> take_until_sum(std::vector<T> &v, const T& n)
{
    std::vector<T> r;
    T sum{};
    auto res = copy_while(begin(v), end(v),
                          std::back_inserter(r),
                          [&](const T& x){return (sum += x)<n;});
    v.erase(begin(v), res.first);
    return r;
}
```

Or, instead of the lambda, you can use a function object like:

```
struct sum_while {
    sum_while(int max) :max{max} {}
    int tot{0};
    int max;
    bool operator()(int x){
        if( (tot + x) < max) {
            tot += x;
            return true;
        } else return false;
    }
};
```

and use it as follows:

```
    auto res = copy_while(begin(v), end(v),
                          std::back_inserter(r), sum_while(n));
```

b) The algorithm can be implemented as

```
#include <utility>

template <typename FwdIt, typename OutputIt, typename Pred>
std::pair<FwdIt, OutputIt>
copy_while(FwdIt first, FwdIt last, OutputIt out, Pred p)
{
    while(first != last && p(*first)){
        *out = *first;
        ++first;
        ++out;
    }
    return std::make_pair(first, out);
}
```

5. The function that creates a predicate can be written with a lambda expression:

```
std::function<bool(char)> is_char_from(std::string s)
{
    return [s](char c){return s.find(c) != string::npos;};
}
```

Alternatively, a class describing the function object can be written:

```
struct Is_char_from {
    Is_char_from(const string& s) : chars(s) {}
    bool operator()(char c) { return chars.find(c) != string::npos; }

  private:
    string chars;
};
```

and an instance of that class returned:

```
std::function<bool(char)> is_char_from(std::string s)
{
    return Is_char_from{s};
}
```

Note that when we write a functor class, the return type has a (known) name, so in this case the function declaration could have been written:

```
Is_char_from is_char_from(std::string);
```

The problem asks for a function, but as the test case is written, it would also work with just a function object class:

```
struct is_char_from {
    is_char_from(const string& s) : chars(s) {}
    bool operator()(char c) { return chars.find(c) != string::npos; }

  private:
    string chars;
};
```

as `auto f = is_char_from("abc");` would be a valid constructor call, and the calls `f('a')`, etc., would call `operator()`.