LUND INSTITUTE OF TECHNOLOGY Department of Computer Science

Solutions, C++ Programming examination

2024-05-31

1. a) As `void add(Vektor<T>, Vektor<T>, Vektor<T>)` has by-value parameters, the arguments will be copied but as `Vektor` does not define a copy constructor, only the *pointer value* is copied. When the function returns, the copies will be destroyed, and thus the array pointed to by `p` will be deallocated, leaving the `Vektor` objects in the calling function in an unusable state.

b) Yes. Change to by-reference parameters:

```
template <typename T>
void add(const Vektor<T>& c1, const Vektor<T>& c2, Vektor<T>& c3)
```

c) No, if a copy-constructor is defined (which it should be) the result will be `0 0 0 0 0 0`, as the assignments in `add` are made to the copy.

d) The empty braces (`{}`) force value initialization. Thus, without them the program will have undefined behaviour for element types that are not guaranteed to be value initialized (e.g. `Vektor<int>`).

2. We need a stateful predicate that accepts an `int` and returns `true` every *n*th call, and `false` otherwise. Just accepting `int` and `std::string` arguments solves the given problem, but to make it general you can make `operator()` a template member function that takes a const reference to any type and ignores it.

```
template <typename T>
class nth {
  public:
    nth(int n) : val{n} {}
    bool operator()(const T&);

  private:
    int val;
    int count{0};
};

template <typename T>
bool nth<T>::operator()(const T&) // We don't use the parameter
                                  // so we don't give it a name
{
    auto res = count == 0;
    if(++count >= val){
        count = 0;
    }
    return res;
}

template <typename InIt, typename OutIt>
OutIt
copy_nth(InIt first, InIt last, OutIt out, int n)
{
    using T = typename std::iterator_traits<InIt>::value_type;
    return std::copy_if(first, last, out, nth<T>(n));
}
```

3. A possible solution is

```
#include <algorithm>
#include <iterator>
#include <map>

struct token {
    token(char c='?') :ch(c) {}
    token(const std::string& code);
    operator char() const {return ch;}
    friend std::istream& operator>>(std::istream& is, token& t)
    {
        char c;
        if (!is.get(c)) {
            return is; // read failed, bail out
        };
        // remove tags
        while(c == '<'){
            std::string code;
            is.ignore(std::numeric_limits<std::streamsize>::max(), '>');
            if(!is.get(c)) return is;
        }
        // translate special char
        if (c == '&'){
            std::string code;
            std::getline(is, code, ';');
            t = token(code);
        } else {
            t = token(c);
        }
        return is;
    }
private:
    char ch;
    const static std::map<std::string, char> codes;
};

const std::map<std::string, char> token::codes{
    {"lt", '<'}, {"gt", '>'}, {"amp", '&'}};

token::token(const std::string& code) : token::token()
{
    auto it = codes.find(code);
    if( it != codes.end()){
        ch = it->second;
    }
}

void remove_html(std::istream& is, std::ostream& os)
{
    std::istream_iterator<token> iit(is);
    std::istream_iterator<token> ie{};
    std::ostream_iterator<char> oi(os);
    std::copy(iit, ie, oi);
}
```

4. The difference is that a local variable is an *lvalue,* whereas the `istringstream` created as a function argument is a temporary object – an *rvalue*.

The first parameter of `remove_html` has the type `std::istream&` and such a reference may only bind to an *lvalue*. Only `const` references and rvalue references (`&&`) may bind to temporary values.

5. Yes, `example1` leaks memory, as the object pointed to by the owning pointer is not destroyed before the pointer goes out of scope.
No, `example2` is correct. It does not leak memory, as the object owned by the `unique_ptr` is destroyed when the pointer goes out of scope.
Yes, `example3` leaks memory, as the object pointed to by the owning pointer is not destroyed before the pointer goes out of scope.
Yes, `example4` leaks memory, as the object pointed to by the owning pointer is not destroyed before the pointer goes out of scope.
Yes, `example5` leaks memory (has undefined behaviour, strictly speaking), as even though the `unique_ptr<Foo>` destroys the object it owns, `Bar::~Bar()` is not called as `Foo::~Foo()` is not `virtual`.
No, `example6` does not leak memory, as the object owned by the `unique_ptr` is destroyed when the pointer goes out of scope.

6. This is a so called *buffer overflow* and has undefined behaviour. The string `"abcdefgh"` is (including the terminating null) 9 characters long, but the char array it is written to is can only hold eight chars, and in this case the null (0) overwrites the value of `nbr`.