Lab one: Elementary C++

Joakim Carselind, 880720-0277, joacar@kth.se
Pascal Chatterjee, 881216-0938, pascalc@kth.se

1.1

Vad betyder \$* i en makefile?
If suffix is a known typ, i.e. .out, then in a rule of type %.out, $* will take ONLY the part before .out. Take the rule

```
%.out:    %.cpp
    g++ -g -Wall $*.cpp -o $*.out
$ make omg.out -> $*  = omg
```

http://www.gnu.org/software/make/manual/make.html#Automatic-Variables
Vad gör -Wall och -g ?
Compile with all warnings enabled (even those disabled by default) and debugging information made available

1.2 a)

```
// int powerof(int x, int y) {
//     int res = 1;
//     for (int i = 0; i < y; i++); {
//             res *= x;
//     }
//     return res;
// }
//
// int main() {
//     int x = 10;
//     int y = 3;
//
//     int res = powerof(x, y);
//
//     std::cout << x << " upphöjt till " << y << " är " << res << std::endl;
//
//     float z = 0.29;
//     int w = (int) (z * x * x);
//     if (z * x * x == 29)
//             std::cout << z << "*" << x * x << " är 29" << std::endl;
//     else
//             std::cout << z << "*" << x * x << " är inte 29" << std::endl;
// }
```

Varför blir värdet på variabeln w inte det man tror (0.29*100)?
Rounding error in w due to bad floating point precision

Hur många varv körs for-loopen i funktionen powerof?
None since it ends with a semi-colon

1.2 b)

```
// int must_follow_a(char * start, int length, char a, char b) {
//     int nr = 0;
//     for (int i = 0; i < length; i++, ++start) {
//           if (*start == a && *(start + 1) == b) // maintainers note: DANGER!
//           nr += 1;
//     }
//     return nr;
// }
```

Dina tre testfall

1. char vek[] = {'a', 'b', 'a', 'b', 'b'};
   int result = must_follow_a(vek, 3, 'a', 'b');
   TS_ASSERT_EQUALS(result, 1);
2. char vek[] = {'b', 'b', 'a', 'b', 'b'};
   int result = must_follow_a(vek, 3, 'a', 'b');
   TS_ASSERT_EQUALS(result, 0);
3. char vek[] = {'a', 'b'};
   int result = must_follow_a(vek, 1, 'a', 'b');
   TS_ASSERT_EQUALS(result, 0);

Varför är det så viktigt att testa randvillkoren?
Edge cases sometimes need some extra logic not necessary for “ordinary” cases.

1.3

Bifoga källkoden till din version av A.cpp

```
 #include <iostream>

class A {
public:
    A()
        {std::cout << "The default contructor" << std::endl; }
    A(const A & ref)
        {std::cout << "The copy contructor" << std::endl; }
    ~A()
        {std::cout << "The destructor" << std::endl; }
    A(char * s)
        {std::cout << "Some other constructor " << s << std::endl;}
    A & operator=(const A & s)
        {std::cout << "The assignment operator" << std::endl;
         return *this;}
};
```

```
void no_ref(A a) {}
void with_ref(const A & a) {}

int main()
{
    std::cout << "Some other constructor" << std::endl;
    A a("my name is a");
    std::cout << "1. A b = a <-- copy constructor" << std::endl;
    A b = a;         // vad är skillnaden
    std::cout << "2. A c(a) <--> A c = a <-- copy constructor" << std::endl;
    A c(a);          // mellan dessa
    std::cout << "3. A d; d = a <-- assignment operator" << std::endl;
    A d;             // tre tekniker?
    d = a;

    std::cout << "noref(a) <-- send by value, so temp object created and
destroyed" << std::endl;
    no_ref(a);       // Bildas temporära objekt?
    std::cout << "with_ref(a) <-- send by ref so no temp object" <<
std::endl;
    with_ref(a);     // Bildas temporära objekt?

    std::cout << "Array creation <-- default constructor * 5" << std::endl;
    A *aa = new A[5];
    std::cout << "delete aa <-- not array delete, so only first A deleted,
MEMORY LEAK" << std::endl;
    delete aa;       // Vad kommer att hända?
    std::cout << "end main" << std::endl;
    return 0;
}
```

Vad skriver ditt program ut, var förberedd att förklara varför.

```
 Some other constructor
Some other constructor my name is a
1. A b = a <-- copy constructor
The copy contructor
2. A c(a) <--> A c = a <-- copy constructor
The copy contructor
3. A d; d = a <-- assignment operator
The default contructor
The assignment operator
noref(a) <-- send by value, so temp object created and destroyed
The copy contructor
The destructor
with_ref(a) <-- send by ref so no temp object
Array creation <-- default constructor * 5
The default contructor
The default contructor
The default contructor
The default contructor
The default contructor
delete aa <-- not array delete, so only first A deleted, MEMORY LEAK
The destructor
A.out(94958) malloc: *** error for object 0x1012008c8: pointer being freed
was not allocated
*** set a breakpoint in malloc_error_break to debug
Abort trap: 6
```

När frigörs objekten?
When delete is called.

När skapas temporära objekt?
When a function is called by value.

Vad är skillnaden mellan dessa tre tekniker: A b = a;, A c(a), A d; d = a
- **A b = a** ← this is the copy constructor
- **A c(a)** ← this is the same as A c = a, so the copy constructor
- **A d; d = a** ← this uses the default constructor followed by the assignment operator

Bildas temporära objekt med anropet *no_ref(a)*?
Yes
Bildas temporära objekt med anropet *with_ref(a)*?
No

Vad kommer att hända när *delete aa* exekveras?
The first element of *aa* will be deleted from the heap, i.e. the space previously occupied by *aa* as freed. **Note:** Not all five elememts will be deleted, only first pointed at. This is because the proper array delete, `delete [] aa`, was not called.

```
// struct Data {
//    int x, y, z;
// };
//
// Data ** foo(Data ** v, int x) {
//    for (int i = 0; i < x; i++)
//          //if (v[i] != 0)
//          v[i] = new Data;
//    return v;
// }
//
// int main () {
//    const int size = 5;
//    Data ** v = new Data * [size];
//    Data ** p = foo(v, size);
//    delete [] p;
// }
```

Hur ser valgrinds felmeddelande ut?
joacar$ valgrind --tool=memcheck --leak-check=yes ./Data.out
==8341== Memcheck, a memory error detector

```
==8341== Copyright (C) 2002-2009, and GNU GPL'd, by Julian Seward et al.
==8341== Using Valgrind-3.6.0.SVN-Debian and LibVEX; rerun with -h for copyright info
==8341== Command: ./Data.out
==8341==
==8341== Conditional jump or move depends on uninitialised value(s)
==8341==       at 0x8048621: foo(Data**, int) (Data.cpp:11)
==8341==       by 0x804868A: main (Data.cpp:21)
==8341==
==8341==
==8341== HEAP SUMMARY:
==8341==       in use at exit: 20 bytes in 1 blocks
==8341==   total heap usage: 1 allocs, 0 frees, 20 bytes allocated
==8341==
==8341== 20 bytes in 1 blocks are definitely lost in loss record 1 of 1
==8341==       at 0x402532E: operator new[](unsigned int) (vg_replace_malloc.c:299)
==8341==       by 0x8048672: main (Data.cpp:20)
==8341==
==8341== LEAK SUMMARY:
==8341==       definitely lost: 20 bytes in 1 blocks
==8341==       indirectly lost: 0 bytes in 0 blocks
==8341==       possibly lost: 0 bytes in 0 blocks
==8341==       still reachable: 0 bytes in 0 blocks
==8341==       suppressed: 0 bytes in 0 blocks
==8341==
==8341== For counts of detected and suppressed errors, rerun with: -v
==8341== Use --track-origins=yes to see where uninitialised values come from
==8341== ERROR SUMMARY: 6 errors from 2 contexts (suppressed: 17 from 6)
```

Blir det någon skillnad i hur mycket minne som läcker när man kommenterar if-satsen?
Yes. Without the if-clause: definitely lost: 20 bytes in 1 blocks, indirectly lost: 60 bytes in 5 blocks and with the if-clause: definitely lost: 20 bytes in 1 blocks

Borde det ha blivit någon skillnad?
Since v is not set, the default pointer (0) is used everywhere and hence the if-clause is never evaluated to true. Without the if-clause a new
Since the Data object at v[i] is not initialised, it is a null-pointer, so v[i] == 0?

Varför läcker programmet fortfarande minne?
Print out: definitely lost: 60 bytes in 5 blocks. The memory leak remains since only the pointers to the Data objects are freed, not the Data objects themselves. Each Data object contains three integers which equals to 12 bytes and it is five in total hence 60 bytes are not freed.

With delete [] * p
definitely lost: 20 bytes in 1 blocks

indirectly lost: 48 bytes in 4 blocks

1.4

Generellt är det ofta en god idé att låta konstruktorer som tar ett argument deklareras som explicit. Varför? Ange ett exempel där det annars kan bli dumt.

```
size_t s = 5;
Vector v = s; ← this is valid if constructor not explicit
```

operatorn[] måste vara en konstant medlemsfunktion i vissa fall. När och varför?
Consider the case int i = a[5] (a[5] != 7) and then i = 7. If a[5] = 7 = i - it is an unwanted behavior. Declaring the method a[5] as ***const*** make that value non-alterable.

Hur kopierar man vektorn?
Create a new vector and go through the addresses and give them the value of the old vectors pointer. Vector v_new = new v_new[v_old.size()] -> *v_n++ = *v_o++.
memcpy(new_array, old_array, old_array.size)