Tentamen
C++-programmering

2015–08–28, 8.00–13.00

Hjälpmedel: En valfri C++-bok. OH-bilderna från föreläsningarna är *inte* tillåtna.

Du ska i dina lösningar visa att du behärskar C++ och att du kan använda C++ standardklasser. ”C-lösningar” ger inga poäng, även om de är korrekta.

Uppgifterna ger preliminärt $6 + 16 + 16 + 12 = 50$ poäng. För godkänt krävs 25 poäng (3/25, 4/33, 5/42).

---

1. I C++11 kan man lagra ”callable objects” (funktioner, lambdafunktioner, funktionsobjekt) i objekt av klassen `function` (definieras i header-filen `<functional>`). Man anger returtyp och parametertyper för funktionen som ska lagras. Exempel:

```
int incr1(int a) {
    int b = 1;
    return a + b;
}

int main() {
    function<int(int)> f1 = incr1; // auto also works
    cout << f1(10) << endl;        // prints 11
}
```

Om man vill ha en funktion som ökar argumentet med ett annat tal än 1 måste man skriva en ny funktion. Alternativt kan man skapa funktionen under exekvering:

```
function<int(int)> create_function(int b) {
    return [???](int a) { return a + b; }; // see below for what ??? should be
}

int main() {
    auto f2 = create_function(2);  // adds 2 to the argument
    cout << f2(10) << endl;        // prints 12
    auto f3 = create_function(5);  // adds 5 to the argument
    cout << f3(10) << endl;        // prints 15
}
```

Förklara vad som inträffar om `[???]` i definitionen av lambdafunktionen byts mot:

a) `[]`
b) `[=]`
c) `[&]`

2. I ett schemaläggningsprogram ska ett antal aktiviteter schemaläggas så att de inte kolliderar i tiden. Med den metod som beskrivs här får man oftast en hyfsad lösning men inte säkert en lösning med minsta totala tid för aktiviteterna.

   Problemet beskrivs enligt följande: ett antal rader innehåller ettor och nollor — etta betyder upptagen, nolla betyder ledig. Raderna ska skiftas åt höger så att det blir högst en etta i varje kolonn. Exempel med fem rader där ettor visas med 1, nollor med blank:

```
1
1 1 11 1
1
11 11  1 1
1        1  1
```

   Alla rader har från början en etta längst till vänster. Man börjar med att sortera raderna så att raden med flest ettor hamnar först, sedan den rad som innehåller näst flest ettor, osv. Detta ger:

```
11 11  1 1
1 1 11 1
1        1  1
1
1
```

   Efter sorteringen har den första raden fått sin rätta plats. Sedan skiftar man den andra raden åt höger tills ingen etta i den kolliderar med någon etta i den första raden. På samma sätt skiftar man den tredje raden tills ingen etta kolliderar med någon etta i *någon* av de tidigare raderna, osv. Slutresultatet blir:

```
11 11  1 1
      1 1 11 1
     1        1  1
  1
            1
```

   Implementera en klass `Scheduler` som genomför schemaläggningen. Programmet ska användas enligt följande exempel:

```
int main() {
    ifstream in("activities.txt"); // input file (lines with 1's and spaces)
    if (!in) {
        cerr << "Cannot open activity file" << endl;
        exit(1);
    }
    Scheduler sc(in); // read input from in
    sc.schedule();    // create schedule
    sc.print(cout);   // print results on cout (lines with 1's and spaces)
}
```

   Tips: `bitset` är en bra klass (på sidan 5 och framåt finns dokumentation av klassen). Du kan förutsätta att positionen för den etta som hamnar längst till höger blir högst 80. (Bitarna ryms alltså inte i ett `unsigned long long`-tal.)

3. En digital gråskalebild består av ett rutnät av pixlar där varje pixel har ett värde mellan 0 (helt svart) och 255 (helt vitt). Följande färdigskrivna klass beskriver sådana bilder:

```
class Image {
public:
    int width() const;            // number of columns
    int height() const;           // number of rows
    int get(int r, int c) const;  // get pixel value at row r, column c
    void set(int r, int c, int v); // set pixel value to v
    ...                           // constructors and other member functions
};
```

Figur 1(a) visar exempel på en gråskalebild (av röda blodkroppar). I figur 1(b) har bruset reducerats, i figur 1(c) har ”föremålen” skilts ut från bakgrunden genom att alla pixlar med värde över ett visst tröskelvärde har satts till 255, övriga pixlar till 0.

För att man ska kunna analysera föremålen som finns i en bild vill man ”känna igen” och numrera föremålen 1, 2, 3, ... Denna numrering kallas *märkning* (görs i uppgift 4) och innebär att pixlar som tillhör samma föremål ges samma värde. Märkningen utgår från en trösklad bild. Figur 1(d) visar hur blodkropparna märkts.

För att hålla reda på att olika pixlar tillhör samma föremål används *ekvivalensklasser* (ordet har inget med klasser i objektorienterad programmering att göra). Om talen 1 och 3 hör ihop, liksom 7 och 8, och 5 och 9, finns det tre ekvivalensklasser:

[1 3] [7 8] [5 9]

Om också 2 och 7 hör ihop utökas en av klasserna:

[1 3] [2 7 8] [5 9]

Om också 1 och 5 hör ihop slås två klasser samman:

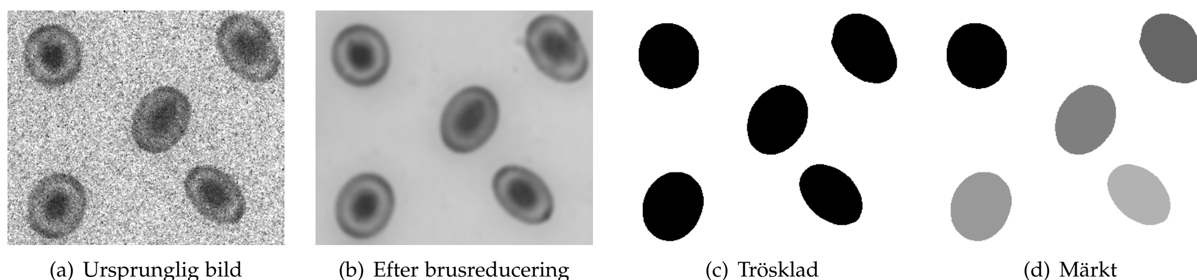
[1 3 5 9] [2 7 8]

Ett tal kan tillhöra högst en ekvivalensklass. Klassen `EquivalenceClasses` beskriver ett antal ekvivalensklasser:

```
class EquivalenceClasses {
public:
    void join(int a, int b); // a and b shall belong to the same equivalence class
                             // (three cases, see below)
    int least(int n) const;  // the smallest number in n's equivalence class
                             // (or n itself, if n doesn't belong to a class)
};
```

I `join`: 1) om båda talen ingår i olika klasser så slås klasserna samman, 2) om ett av talen inte ingår i en klass läggs det in i det andra talets klass, 3) annars skapas en ny klass.

Implementera klassen `EquivalenceClasses`. Varje ekvivalensklass är en mängd av heltal i intervallet $[1,255]$. Det är praktiskt att representera en sådan mängd med ett `bitset` (dokumentation finns på sidan 5 och framåt).



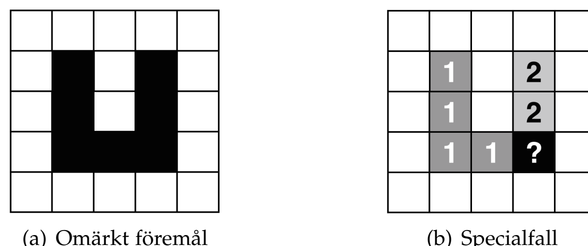
(a) Ursprunglig bild (b) Efter brusreducering (c) Trösklad (d) Märkt

Figur 1: Bilder av röda blodkroppar

4. Vi återvänder till märkningen av gråskalebilderna. Märkningen bygger på följande idé: gå igenom bildens pixlar. För varje pixel som tillhör ett föremål undersöks två andra pixlar: den ovanför och den till vänster. Om någon av dessa pixlar (ovanför/vänster) är märkt, så låter man den ”färga av sig” till den aktuella pixeln. Annars anses pixeln tillhöra ett nytt föremål och märks med ett eget värde.

   Ett specialfall inträffar då pixlarna (ovanför/vänster) båda är märkta men med olika värden. Ett exempel visas i figur 2(b), då pixeln med frågetecknet ska märkas. Man väljer då ett av de två värdena till den aktuella pixeln och gör en notering om att de två värdena (1 och 2 i figuren) är ekvivalenta. De tillhör ju samma föremål.

   Efteråt gås bilden igenom en andra gång. Då får alla pixlar i varje föremål samma värde, utifrån de ekvivalenser som upptäckts. Föremålet i figur 2 kan då märkas med värdet 1 för alla pixlar.



(a) Omärkt föremål (b) Specialfall

Figur 2: Märkning

Noggrannare beskrivning av algoritmen:

1. Låt n vara ett heltal, inledningsvis 1.
2. Gå igenom bildens alla pixlar, rad för rad från vänster till höger. Om den aktuella pixeln tillhör ett föremål (dess värde är mindre än 255) beräknas dess nya värde a som följer:
   - Låt p_1 vara pixelvärdet närmast till vänster om pixeln.
   - Låt p_2 vara pixelvärdet närmast ovanför pixeln.
   - Om p_1 tillhör ett föremål, men inte p_2, låt $a = p_1$.
   - Om p_2 tillhör ett föremål, men inte p_1, låt $a = p_2$.
   - Om både p_1 och p_2 tillhör föremål, låt $a = p_1$. Om p_1 och p_2 är olika (och båda tillhör föremål), notera att p_1 och p_2 är ekvivalenta.
   - Om varken p_1 eller p_2 tillhör föremål, låt $a = n$ och öka n med 1.
3. Gå därefter igenom bildens alla pixlar en andra gång. Varje pixel som tillhör ett föremål får sitt värde a ersatt med det minsta pixelvärde som är ekvivalent med a.

Skriv en global funktion med rubriken `void label(Image& im)` som utför märkningen.

std::bitset

Defined in header <bitset>

```
template< std::size_t N >
class bitset;
```

The class template `bitset` represents a fixed-size sequence of `N` bits. Bitsets can be manipulated by standard logic operators and converted to and from strings and integers.

`bitset` meets the requirements of `CopyConstructible` and `CopyAssignable`.

Template parameters

N - the number of bits to allocate storage for

Member types

reference	proxy class representing a reference to a bit (class)

Member functions

(constructor)	constructs the bitset (public member function)
operator== **operator!=**	compares the contents (public member function)

Element access

operator[]	accesses specific bit (public member function)
test	accesses specific bit (public member function)
all (C++11) **any** **none**	checks if all, any or none bits are set to `true` (public member function)
count	returns the number of bits set to `true` (public member function)

Capacity

size	returns the size number of bits that the bitset can hold (public member function)

Modifiers

operator&= **operator|=** **operator^=** **operator~**	performs binary AND, OR, XOR and NOT (public member function)
operator<<= **operator>>=** **operator<<** **operator>>**	performs binary shift left and shift right (public member function)
set	sets bits to `true` or given value (public member function)
reset	sets bits to `false` (public member function)
flip	toggles the values of bits (public member function)

Conversions

to_string	returns a string representation of the data (public member function)
to_ulong	returns an unsigned long integer representation of the data (public member function)
to_ullong (C++11)	returns an unsigned long long integer representation of the data (public member function)

Non-member functions

operator& **operator|** **operator^**	performs binary logic operations on bitsets (function template)
operator<< **operator>>**	performs stream input and output of bitsets (function template)

Helper classes

std::hash<std::bitset> (C++11)	hash support for **std::bitset** (class template specialization)

Notes

If the size of the bitset is not known at compile time, std::vector<bool> or boost::dynamic_bitset (http://www.boost.org/doc/libs/release/libs/dynamic_bitset/dynamic_bitset.html) may be used.

Retrieved from "http://en.cppreference.com/mwiki/index.php?title=cpp/utility/bitset&oldid=59501"

std::bitset::bitset

`constexpr bitset();`	(1)	
`constexpr bitset( unsigned long val );`	(2)	(until C++11)
`constexpr bitset( unsigned long long val );`		(since C++11)

`template< class CharT, class Traits, class Alloc >`
`explicit bitset( const std::basic_string<CharT,Traits,Alloc>& str,`
`typename std::basic_string<CharT,Traits,Alloc>::size_type pos = 0,`
`typename std::basic_string<CharT,Traits,Alloc>::size_type n =`
`std::basic_string<CharT,Traits,Alloc>::npos);`
	(3)	(until C++11)

`template< class CharT, class Traits, class Alloc >`
`explicit bitset( const std::basic_string<CharT,Traits,Alloc>& str,`
`typename std::basic_string<CharT,Traits,Alloc>::size_type pos = 0,`
`typename std::basic_string<CharT,Traits,Alloc>::size_type n =`
`std::basic_string<CharT,Traits,Alloc>::npos,`
`CharT zero = CharT('0'),`
`CharT one = CharT('1'));`
		(since C++11)

`template< class CharT >`
`explicit bitset( const CharT* str,`
`typename std::basic_string<CharT>::size_type n =`
`std::basic_string<CharT>::npos,`
`CharT zero = CharT('0'),`
`CharT one = CharT('1'));`
	(4)	(since C++11)

Constructs a new bitset from one of several optional data sources:

1) Default constructor. Constructs a bitset with all bits set to zero.
2) Constructs a bitset, initializing the first (rightmost, least significant) `M` bit positions to the corresponding bit values of `val`, where `M` is the smaller of the number of bits in an `unsigned long long` and the number of bits `N` in the bitset being constructed. If M is less than N (the bitset is longer than 32 (until C++11) 64 (since C++11) bits, for typical implementations of unsigned long (since C++11) long), the remaining bit positions are initialized to zeroes.
3) Constructs a bitset using the characters in the `std::basic_string` `str`. An optional starting position pos and length n can be provided, as well as characters denoting alternate values for set (one) and unset (`zero`) bits. `Traits::eq()` is used to compare the character values.

   The effective length of the initializing string is min(n, `str.size() - pos`).

   If `pos > str.size()`, this constructor throws `std::out_of_range`. If any characters examined in `str` are not `zero` or one, it throws `std::invalid_argument`.
4) Similar to (3), but uses a `CharT*` instead of a `std::basic_string`. Equivalent to `bitset(n == basic_string<charT>::npos ? basic_string<charT>(str) : basic_string<charT>(str, n), 0, n, zero, one)`

Parameters

val	-	number used to initialize the bitset
str	-	string used to initialize the bitset
pos	-	a starting offset into `str`
n	-	number of characters to use from `str`
one	-	alternate character for set bits in `str`
zero	-	alternate character for unset bits in `str`

Exceptions

1) none 2) none	(until C++11)
1) noexcept specification: `noexcept` 2) noexcept specification: `noexcept`	(since C++11)

3) `std::out_of_range` if `pos > str.size()`, `std::invalid_argument` if any character is not one or zero
4) `std::invalid_argument` if any character is not one or zero

Example

Run this code

```
#include <bitset>
#include <string>
#include <iostream>
#include <climits>

int main()
```

```
{
    // empty constructor
    std::bitset<8> b1; // [0,0,0,0,0,0,0,0]

    // unsigned long long constructor
    std::bitset<8> b2(42);          // [0,0,1,0,1,0,1,0]
    std::bitset<70> bl(ULLONG_MAX); // [0,0,0,0,0,0,1,1,1,...,1,1,1] in C++11
    std::bitset<8> bs(0xfff0);      // [1,1,1,1,0,0,0,0]

    // string constructor
    std::string bit_string = "110010";
    std::bitset<8> b3(bit_string);       // [0,0,1,1,0,0,1,0]
    std::bitset<8> b4(bit_string, 2);    // [0,0,0,0,0,0,1,0]
    std::bitset<8> b5(bit_string, 2, 3); // [0,0,0,0,0,0,0,1]

    // string constructor using custom zero/one digits
    std::string alpha_bit_string = "aBaaBBaB";
    std::bitset<8> b6(alpha_bit_string, 0, alpha_bit_string.size(),
                      'a', 'B');         // [0,1,0,0,1,1,0,1]

    // char* constructor using custom digits
    std::bitset<8> b7("XXXXYYYY", 8, 'X', 'Y'); // [0,0,0,0,1,1,1,1]

    std::cout << b1 << '\n' << b2 << '\n' << bl << '\n' << bs << '\n'
              << b3 << '\n' << b4 << '\n' << b5 << '\n' << b6 << '\n'
              << b7 << '\n';
}
```

Output:

```
00000000
00101010
0000001111111111111111111111111111111111111111111111111111111111111111
11110000
00110010
00000010
00000001
01001101
00001111
```

See also

set	sets bits to `true` or given value (public member function)
reset	sets bits to `false` (public member function)

Retrieved from "http://en.cppreference.com/mwiki/index.php?title=cpp/utility/bitset/bitset&oldid=74090"