Esercizio 1

Definire un template di classe `stack<T,num>` i cui oggetti rappresentano uno stack di valori di un generico tipo `T` con al massimo `num` elementi. Si ricorda che lo stack implementa la politica di inserimento/rimozione LIFO: Last In First Out. Lo stack si dice pieno quando memorizza `num` elementi. Il template `stack<T,num>` deve soddisfare i seguenti vincoli e rendere disponibili le seguenti funzionalità:

1. `stack<T,num>` non può usare i contenitori STL come campi dati (inclusi puntatori e riferimenti a contenitori STL).
2. Il template `stack<T,num>` ha come tipo `T` default `char` e come valore `num` default `100`.
3. Un costruttore di default che costruisce lo stack vuoto.
4. Un costruttore `stack(t,k)` che costruisce uno stack di `k` elementi che memorizzano il valore `t`; se `k` $>$ `num` allora lo stack sarà di `num` elementi, se `k` < 0 allora lo stack sarà vuoto.
5. Metodi `bool isEmpty()` e `bool isFull()` che testano se lo stack è vuoto o pieno.
6. Metodo `unsigned int size()` che ritorna il numero di elementi memorizzati dallo stack.
7. Gestione della memoria senza condivisione.
8. Operatore esplicito di conversione ad `int` che ritorna la dimensione dello stack.
9. Metodo `bool push(const T&)`: in una chiamata `s.push(t)`, inserisce al top dello stack `s` un nuovo elemento che memorizza il valore `t` se ciò non provoca il superamento del limite `num`, altrimenti lascia lo stack `s` invariato. Ritorna `true` se l'inserimento è avvenuto, `false` altrimenti.
10. Metodo `void pop()`: in una chiamata `s.pop()` rimuove l'elemento al top dello stack `s` non vuoto; se `s` è vuoto lo lascia inalterato.
11. Metodo `T top()`: `s.top()` ritorna una copia dell'elemento al top dello stack `s` non vuoto; se `s` è vuoto, allora `s.top()` provoca undefined behaviour (da definirsi opportunamente).
12. Metodo `T bottom()`: `s.bottom()` ritorna una copia dell'elemento al bottom dello stack `s` non vuoto; se `s` è vuoto, allora `s.bottom()` provoca undefined behaviour (da definirsi opportunamente).
13. Metodo `bool search(const T&)`: in una chiamata `s.search(t)` ritorna true se il valore `t` occorre nello stack `s`, altrimenti ritorna false.
14. Metodo `void flush()` che svuota lo stack di invocazione.
15. Overloading dell'operatore di somma tra stack: `s1` + `s2` deve ritornare un nuovo stack ottenuto impilando `s2` sopra `s1` (il bottom di `s2` è quindi sopra il top `s1`), sino all'eventuale raggiungimento del massimo `num` di elementi.
16. Overloading dell'operatore di uguaglianza.
17. Overloading dell'operatore di output.

Esercizio 2

Definire un template di classe `Coda<T>` i cui oggetti rappresentano una struttura dati coda per elementi di uno stesso tipo `T`, ossia la coda implementa l'usuale politica FIFO (First In First Out) di inserimento/estrazione degli elementi: gli elementi vengono estratti nello stesso ordine in cui sono stati inseriti. Il template `Coda<T>` deve soddisfare i seguenti vincoli:

1. `Coda<T>` non può usare i contenitori STL come campi dati (inclusi puntatori e riferimenti a contenitori STL).
2. Il parametro di tipo del template `Coda<T>` ha come valore di default `double`.
3. Gestione della memoria senza condivisione.
4. Deve essere disponibile un costruttore di default che costruisce la coda vuota.
5. Deve essere disponibile un costruttore `Coda(int k, const T& t)` che costruisce una coda contenente `k` copie dell'elemento `t`.
6. Deve essere disponibile un metodo `void insert(const T&)` con il seguente comportamento: `c.insert(t)` inserisce l'elemento `t` in coda a `c` in tempo costante.

7. Deve essere disponibile un metodo `T removeNext()` con il seguente comportamento: se la coda `c` non è vuota, `c.removeNext()` rimuove l'elemento in testa alla coda `c` in tempo costante e lo ritorna; se invece `c` è vuota allora provoca undefined behaviour (da definirsi opportunamente).

8. Deve essere disponibile un metodo `T* getNext()` con il seguente comportamento: se la coda `c` non è vuota, `c.getNext()` ritorna un puntatore all'elemento in testa a `c` in tempo costante; se invece `c` è vuota ritorna il puntatore nullo.

9. Overloading di `operator<` che implementa il confronto lessicografico tra code.

10. Overloading dell'operatore di somma che agisca come concatenazione: `c + d` ritorna la coda che si ottiene aggiungendo `d` in coda a `c`.

11. `Coda<T>` rende disponibile un tipo iteratore costante `Coda<T>::const_iterator` i cui oggetti permettono di iterare sugli elementi di una coda `c` senza permettere modifiche al contenuto di `c`.

Esercizio 3

Il seguente programma compila correttamente. L'esecuzione quali stampe provoca in output su `cout`? Scrivere "VALORE CASUALE" quando si prevede che una certa istruzione provochi una stampa di un valore casuale, e "RUN-TIME ERROR" quando si prevede che una certa istruzione provochi un errore run-time.

```
class C {
public:
  int a[2];
  C(int x=0,int y=1) {
     a[0]=x; a[1]=y; cout << "C(" << a[0] << "," << a[1] << ") ";
  }
};

class D {
private:
  C c1;
  C *c2;
  C& cr;
public:
  D() : c2(&c1), cr(c1) { cout << "D() ";}
  D(const D& d) : cr(c1) { cout << "Dc ";}
  ~D() { cout << "~D ";}
};

class E {
public:
  static C cs;
};
C E::cs;

int main() {
  C c; cout << "UNO" << endl;
  C x(c); cout << x.a[0] << " " << x.a[1] << " DUE" << endl;
  D d=D(); cout << "TRE" << endl;
  E e; cout << "QUATTRO" << endl;
}
```

Esercizio 4

Il seguente programma compila correttamente. L'esecuzione quali stampe provoca in output su `cout`? Scrivere "VALORE CASUALE" quando si prevede che una certa istruzione provochi una stampa di un valore casuale, e "RUN-TIME ERROR" quando si prevede che una certa istruzione provochi un errore run-time.

```
class C {
private:
  int number;
public:
  C(int n=0) : number(n) { cout << "C(" << number << ") "; }
```

```
  ~C() { cout << "~C "; }
  C(const C& c) : number(c.number) { cout << "Cc(" << number << ") "; }
  operator int() { cout << "int() "; return 3;}
};

int F(C c) {return c;}

int main() {
  C *c=new C; cout << "UNO" << endl;
  C d; cout << "DUE" << endl;
  int x=F(d); cout << "TRE" << endl;
  C e=F(d); cout << "QUATTRO" << endl;
}
```

Esercizio 5

Si consideri il seguente frammento di codice:

```
namespace ns {
  class C {
  private:
    friend int f();
    int x;
  public:
    C(int n=0) : x(n) {}
  };
}

int f() {
  ns::C c;
  return c.x;
}

int main() {
  f();
}
```

Quali tra le seguenti affermazioni è esatta?

1. non compila perché `f` non può accedere alla parte privata di `C`
2. non compila perché `f` è una funzione privata
3. dà un errore di accesso illegale a `C.x` a runtime
4. compila, linka ed esegue correttamente

Esercizio 6

Il seguente programma compila correttamente. L'esecuzione quali stampe provoca in output su `cout`? Scrivere "VALORE CASUALE" quando si prevede che una certa istruzione provochi una stampa di un valore casuale, e "RUN-TIME ERROR" quando si prevede che una certa istruzione provochi un errore run-time.

```
class C {
public:
  int number;
  C(int n=1) : number(n) { cout << "C(" << number << ") "; }
  ~C() { cout << "~C(" << number << ") "; }
  C& operator=(const C& c) { number=c.number; cout << "operator=(" << number << ") "; }
};

int F(C c) {return c.number;}

int main() {
  C *c=new C; cout << "UNO" << endl;
```

```
  C d; d=*c; cout << "DUE" << endl;
  int x=F(d); cout << "TRE" << endl;
  int y=F(F(d)); cout << "QUATTRO" << endl;
}
```

Esercizio 7

Il seguente programma compila correttamente. L'esecuzione quali stampe provoca in output su `cout`? Scrivere "VALORE CASUALE" quando si prevede che una certa istruzione provochi una stampa di un valore casuale, e "RUN-TIME ERROR" quando si prevede che una certa istruzione provochi un errore run-time.

```
class C {
public:
  int a[2];
  C(int x=0,int y=1) {a[0]=x; a[1]=y; cout << "C(" << a[0] << "," << a[1] << ") ";}
  C(const C&) {cout << "Cc ";}
};

class D {
private:
  C c1;
  C *c2;
  C& cr;
public:
  D() : c2(&c1), cr(c1) { cout << "D() ";}
  D(const D& d) : cr(c1) { cout << "Dc ";}
  ~D() { cout << "~D ";}
};

class E {
public:
  static C cs;
};
C E::cs=1;

int main() {
  C c; cout << "UNO" << endl;
  C x(c); cout << x.a[0] << " " << x.a[1] << " DUE" << endl;
  D d=D(); cout << "TRE" << endl;
  E e;cout << "QUATTRO" << endl;
}
```

Esercizio 8

Si considerino le seguenti definizioni di template di classe.

```
template<class T> class D; // dichiarazione incompleta

template<class T1, class T2>
class C {
  friend class D<T1>;
private:
  T1 t1;
  T2 t2;
};

template<class T>
class D {
public:
  void m() {C<T,T> c; cout << c.t1 << c.t2;}
  void n() {C<int,T> c; cout << c.t1 << c.t2;}
  void o() {C<T,int> c; cout << c.t1 << c.t2;}
  void p() {C<int,int> c; cout << c.t1 << c.t2;}
  void q() {C<int,double> c; cout << c.t1 << c.t2;}
```

```
  void r() {C<char,double> c; cout << c.t1 << c.t2;}
};
```

Determinare se i seguenti `main()` compilano correttamente o meno.

(1) `int main() { D<char> d; d.m(); }`

(2) `int main() { D<char> d; d.n(); }`

(3) `int main() { D<char> d; d.o(); }`

(4) `int main() { D<char> d; d.p(); }`

(5) `int main() { D<char> d; d.q(); }`

(6) `int main() { D<char> d; d.r(); }`

Esercizio 9

Il seguente programma compila ed esegue correttamente. Quali stampe produce in output la sua esecuzione?

```
class A {
  friend class C;
private:
  int k;
public:
  A(int x=2): k(x) {}
  void m(int x=3) {k=x;}
};

class C {
private:
  A* p;
  int n;
public:
  C(int k=3) {if (k>0) {p = new A[k]; n=k;}}
  A* operator->() const {return p;}
  A& operator*() const {return *p;}
  A* operator+(int i) const {return p+i;}
  void F(int k, int x) {if (k<n) p[k].m(x);}
  void stampa() const {
    for(int i=0; i<n; i++) cout << p[i].k << ' ';
  }
};

int main() {
  C c1;  c1.F(2,9);
  C c2(4); c2.F(0,8);
  *c1=*c2;
  (c2+3)->m(7);
  c1.stampa(); cout << "UNO\n";
  c2.stampa(); cout << "DUE\n";
  c1=c2;
  *(c2+1)=A(3);
  c1->m(1);
  *(c2+2)=*c1;
  c1.stampa(); cout << "TRE\n";
  c2.stampa(); cout << "QUATTRO";
}
```

Esercizio 10

Definire un template di classe `C<T,size>` con parametro di tipo `T` e parametro valore `size` di tipo intero senza segno che soddisfi le seguenti specifiche:

1. `MultiInfo<T>` è un template di classe associato ed annidato nel template `C<T,size>`. Un oggetto di `MultiInfo<T>` rappresenta un oggetto di tipo `T`, detto *informazione*, con una *molteplicità* $\mathtt{m} \geq 0$.

2. Un oggetto di `C<T,size>` rappresenta un array allocato dinamicamente di dimensione `size` di oggetti di `MultiInfo<T>`.

3. `C<T,size>` rende disponibile un costruttore `C(const T&, int)` con il seguente comportamento: una invocazione `C(t,k)` costruisce un oggetto di `C<T,size>` il cui array contiene in ogni posizione un oggetto di `MultiInfo<T>` con informazione `t` e quando `k` è ≥ 1 con molteplicità `k`, altrimenti (cioè quando `k` è < 1) con molteplicità 0.

4. Nel template `C<T,size>` il costruttore di copia, l'assegnazione e il distruttore devono essere "profondi".

5. `C<T,size>` fornisce l'overloading dell'operatore `T* operator[](int)` con il seguente comportamento: se $0 \leq$ `k` $<$ `size` allora una invocazione `c[k]` ritorna un puntatore all'informazione di tipo `T` memorizzata nell'array di `c` in posizione `k`, altrimenti ritorna il puntatore nullo.

6. `C<T,size>` fornisce un metodo `int occorrenze(const T&)` con il seguente comportamento: una invocazione `c.occorrenze(t)` ritorna la somma delle molteplicità di tutte le occorrenze dell'informazione `t` nell'array memorizzato in `c`.

7. Deve essere disponibile l'overloading dell'operatore di output per oggetti di `C<T,size>` che permette di stampare tutte le informazioni di tipo `T` con relativa molteplicità memorizzate nell'array di un oggetto di `C<T,size>`.

8. Deve essere disponibile un opportuno overloading dell'operatore booleano di confronto per uguaglianza tra oggetti di `C<T,size>`.

Esercizio 11

Il seguente programma compila ed esegue correttamente. Quali stampe produce in output la sua esecuzione?

```
class It {
  friend class C;
public:
  bool operator<(It i) const {return index < i.index;}
  It operator++(int) { It t = *this; index++; return t; }
  It operator+(int k) {index = index + k; return *this; }
private:
  int index;
};

class C {
public:
  C(int k) {
    if (k>0) {dim=k; p = new int[k];}
    for(int i=0; i<k; i++) *(p+i)=i;
  }
  It begin() const { It t; t.index = 0; return t; }
  It end() const { It t; t.index = dim; return t; }
  int& operator[](It i) {return *(p + i.index);}
private:
  int* p;
  int dim;
};

int main() {
  C c1(4), c2(8);
  for(It i = c1.begin(); i < c1.end(); i++) cout << c1[i] << ' ';
  cout << "UNO\n";
  It i = c2.begin();
  for(int n=0; i < c2.end(); ++n, i = i+n) cout << c2[i] << ' ';
  cout << "DUE";
}
```

Esercizio 12

I seguenti programmi compilano? La loro esecuzioni provoca errori run-time? Se compilano ed eseguono correttamente, quali stampe provocano le loro esecuzioni?

(1)

```
class C {
public:
  int x;
  void f() { x=1; }
};

class D: public C {
public:
  int y;
  void f() { C::f(); y=2; }
};

int main() {
  C c; D d; c.f(); d.f();
  cout << c.x << endl;
  cout << d.x << " " << d.y;
}
```

(2)

```
class C {
public:
  int a;
  void fC() { a=2; }
};

class D: public C {
public:
  double a;
  void fD() { a=3.14; C::a=4; }
};

class E: public D {
public:
  char a;
  void fE() { a='*'; C::a=5; D::a=6.28; }
};

int main() {
  C c; D d; E e;
  c.fC(); d.fD(); e.fE();
  D* pd = &d; E& pe = e;
  cout << pd->a << ' ' << pe.a << endl;
  cout << pd->a << ' ' << pd->D::a << ' ' << pd->C::a << endl;
  cout << pe.a << ' ' << pe.D::a << ' ' << pe.C::a << endl;
  cout << e.a << ' ' <<  e.D::a << ' ' << e.C::a << endl;
}
```

(3)

```
class C {
public:
  void f() {cout << "C::f" << endl;}
};

class D: public C {
public:
  void f() {cout << "D::f" << endl;}
};

class E: public D {
public:
  void f() {cout << "E::f" << endl;}
};

int main() {
  C c; D d; E e;
```

```
  C* pc = &c; E* pe = &e;
  c = d;
  c = e;
  d = e;
  d = c;
  C& rc=d;
  D& rd=e;
  pc->f();
  pc = pe;
  rd.f();
  c.f();
  pc->f();
}
```