# Objektno orijentisano programiranje u C++-u Projektni uzorci

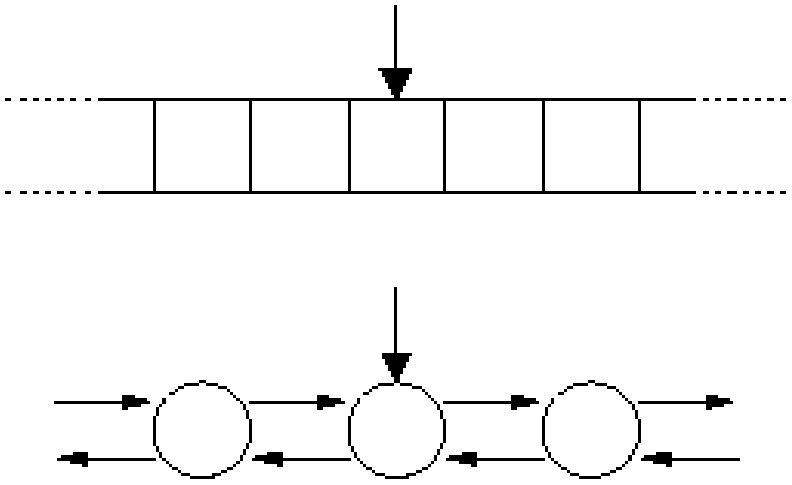


## IMPLEMENTACIJA LANČANE LISTE

# ITERATOR
## Definicija iteratora



Iteratori predstavlja generalizaciju pokazivača koji ukazuje na element nekog kontejnera. Omogućava pomeranje po kontejneru (`operator++`) i pristup elementima kontejnera (`operator*`).



Svaki generički algoritam nad sekvecnom (ili kontejnerom) podrazumeva da su sledeći problemi rešeni.

```
/* PRIMER: find algoritam */
template <typename iterator_t, typename T>
iterator_t find(iterator_t begin, iterator_t end, const T& value){
  while (begin != end && *begin != value) ++begin;
  return begin;
}
```

1. Kako je definisan početak sekvence?
   **ODGOVOR: Iterator koji ukazuje na početak sekvence.**

2. Kako da se krećem po sekvenci?
   **ODGOVOR: operatori ++, -- nad iteratorima, aritmetika iteratora.**

3. Kako da prepoznam kraj sekvence?
   **ODGOVOR: Iterator koji ukazuje na kraj sekvence.**

4. Kako da pristupim elementima sekvence?
**ODGOVOR: operator* (dereferenciranja)**

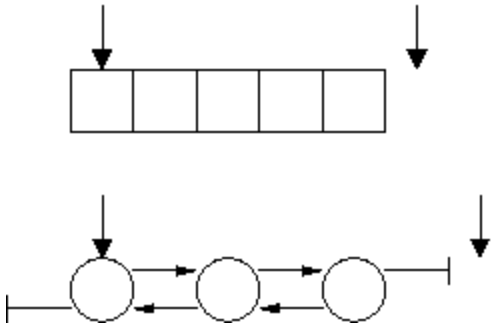
5. Šta je rezultat algoritma find (trazi specificirani element u sekvenci) u slučaju uspeha?
**ODGOVOR: Iterator koji ukazuje na nadjenu poziciju.**

6. Šta je rezultat algoritma find (trazi specificirani element u sekvenci) u slučaju neuspeha?
**ODGOVOR: end iterator (Iterator koji ukazuje na element iza poslednjeg elementa sekvence)**

Kontejneri i sekvence kao tipovi podataka definišu početak i kraj (begin i end iterator)

**Opseg predstavla STL (Standard Template Library)** način definisanja **sekvence**. Većina STL algoritama nad sekvencama uzima opseg kao argument.

Opseg je par iteratora koji ukazuju na početak i kraj sekvence nad kojom generički algoritam radi. Prvi iterator ukazuje na prvi element sekvence, a drugi iterator ukazuje na **jedan korak iza poslednjeg elementa** sekvence. Kažemo da je opseg poluotvoren.



Opseg `[begin,end)` je validan ako do `end` može da se dodje polazeći od `begin`, uzastopnim ponavljanjem `operator++` nad iteratorom.

**Zahvaljujući asimetriji, poluotvoreni opseg** `[begin,end)` poseduje sledeće osobine:

- `[iter,iter)` je prazan opseg
- Duzina opsega `[begin,end)` jednaka je **end-begin**.
- `[A,A+N)` je opseg niza `int A[N]`
- `end` se koristi da bi se signalizirao neuspeh pretrazivanja.
- Petlje nad elementima opsega su:

```
for (iter = begin; iter != end; ++iter) {
  /* radi sa  *iter     */
}

while (begin != end) {
 /* radi sa  *begin     */
  ++begin;
}
```

- Ako se **iter** nalazi unutar opsega **[begin,end)**, tada je **[begin,end)** konkatenacija opsega **[begin,iter)** i **[iter,end)**.
- Kada se umeće novi element na poziciju **[iter,end)**, smešta se neposredno ispred elementa na koji **iter** ukazuje. Rezultujuća sekvenca je **[begin,iter)[novi][iter,end)**. Rezultat operacije insert je iterator koji ukazuje na (prvi) umetnuti element.
- Umetanje na pocetak **begin** i kraj **end** funkcioniše po prethodnom principu.

**Iterator može da ima tri vrste vrednosti:**

- **Dereferencijabilan** (engl. **dereferenceable**) iterator ukazuje na element sekvence. (Vrednosti iteratora je dereferencijabilna). Napomenimo da je begin() dereferencijabilan iterator
- **End()** (engl. **past the end iterator**) iterator ukazuje na jedan korak iza poslednje pozicije kontejnera (sekvence).
- **Singularni iterator** ne ukazuje ni na jedan element. Odgovara null pointeru.

**operator* je legalan samo nad dereferencijabilnim iteratorima.**

**Koncept je skup zahteva koji neki tip mora da ispuni**

**Sintaksni zahtevi definišu izraze koji moraju da budu legalni.**

**Semantički zahtevi definišu efekte koje takvi izrazi moraju da imaju.**

```
/* PRIMER: find algoritam */
template <typename iterator_t, typename T>
iterator_t find(iterator_t begin, iterator_t end, const T& value){
  while (begin != end && *begin != value) ++begin;
  return begin;
}
```

Algoritam `find` koristi sledeće operatore nad iteratorom

- Preinkrementiranje `(operator++)`
- Dereferenciranje `(opertor*)`
- Poredjenje `(operator!=)`
- Konstruktor kopije

To znači da će algoritam raditi sa bilo kojim tipom iteratora koji definiše te operatore na odgovarajući način. **Takva lista zahteva naziva se koncept.**

# LANČANA LISTA
## 1. Čvor lančane liste



```
#ifndef _MLIST_H_
#define _MLIST_H_
// Forward deklaracije
template <typename T> class list;
template <typename T> class iterator;
template <typename T> class const_iterator;

template <typename T>
class node{
  friend list<T>;
  friend typename list<T>::iterator;
  friend typename list<T>::const_iterator;

  node(T val): next(NULL), value(val){}
  node(T val, node<T> *p): next(p), value(val){}

  node<T>* next;
  T value;
};
....
#endif
```

Zašto je typename neophodno u prethodnim definicijama aliasa?

**Zbog kvalifikovanih zavisnih imena (T::iterator), tj. T::iterator pripada tipu T i zavisi od T**

Primer

Sledeća linija koda

```
template <class T> void Fun() { T::iterator * iter; ... }
```

podrazumeva da postoji kod

```
class ContainsAType { class iterator { ... }; ... };
```

ili kod

```
template <typename T> class list{
public:
  //typedef deklaracije aliasa (drugih imena tipova)
  typedef iterator<T> iterator;
};
```

i da funkcija Fun trebada bude pozvana na sledeći način:

```
Fun< ContainsAType >(); ili Fun< list<int> >();
```

U tom slučaju `T::iterator * iter` predstavlja deklaraciju pokazivača tipa `ContainsAType::iterator`, tj. pokazivača na iterator.

Zašto je typename neophodno u prethodnim definicijama aliasa?

**Zbog kvalifikovanih zavisnih imena (T::iterator), tj. T::iterator pripada tipu T i zavisi od T**

Primer

Ono što programer ne očekuje, a može da se desi da neko deklariše statičku promenljivu iteator u klasi `ContainsAType`.

```
class ContainsAType { public: static int iterator; ... };
```

U tom slučaju

```
T::iterator * iter
```

predstavlja numerički izraz, tj. proizvod dve promenljive.

Isti kod se tumaci od strane kompajlera razlicito, a konacna odluka ne moze da se donese pre instanciranja. Da bi se izbegao ovakav problem uveden je zahtev u stanard:

**Ispred kvalifikovanog zavisnog imena mora da stoji typename.**

```
// _MLIST_H_
template <typename T>
class iterator{
public:
  friend list<T>;
  friend const_iterator<T>;
  typedef T  value_type;
  typedef T* pointer;
  typedef T& reference;

  iterator(node<T>* ptr = NULL): m_ptrNode(ptr)  {}
  reference operator*(){return m_ptrNode->value;}
  pointer operator->(){ return m_ptrNode;}

  iterator& operator++(){
    m_ptrNode = m_ptrNode->next;
    return *this;
  }
....
};
#endif // _MLIST_H_
```

```
// _MLIST_H_
template <typename T>
class iterator{
public:
....
  const iterator operator++(int){
    iterator toRet = *this;
    ++(*this);
    return toRet;
  }
  bool operator ==(const iterator& other){
    return m_ptrNode == other.m_ptrNode;
  }
  bool operator !=(const iterator& other)  {
    return !(*this == other);
  }
...
};  // KRAJ SABLONA ITERATOR

#endif // _MLIST_H_
```

```
// _MLIST_H_
template <typename T>
class iterator{
public:
....
  bool operator ==(const const_iterator& other){
    return m_ptrNode == other.m_ptrNode; /* dozvoljen pristup
privatnom clanu od strane prijatelja */
  }
  bool operator !=(const const_iterator& other)  {
    return !(*this == other);
  }
protected:
    node<T>* m_ptrNode;
};  // KRAJ SABLONA ITERATOR

#endif // _MLIST_H_
```

```
// _MLIST_H_
template <typename T>
class const_iterator{
public:
  friend list<T>;
  friend iterator<T>;

  typedef T value_type;
  typedef const T* pointer;
  typedef const T& reference;
  const_iterator(node<T>* ptr = NULL): m_ptrNode(ptr)  {}
  const_iterator(const const_iterator<T>& other):
                                          m_ptrNode(other.m_ptrNode){}
  const_iterator(const iterator<T>& other):
                                          m_ptrNode(other.m_ptrNode){}

  reference operator*(){return m_ptrNode->value;}
  pointer operator->(){return m_ptrNode;}

  const_iterator& operator++(){
    m_ptrNode = m_ptrNode->next;
    return *this;
  }
```

```
// _MLIST_H_
template <typename T>
class const_iterator{
public:
...
  const const_iterator operator++(int){
    const_iterator toRet = *this;
    ++(*this);
    return toRet;
  }
  bool operator ==(const const_iterator& other){
    return m_ptrNode == other.m_ptrNode;
  }
  bool operator !=(const const_iterator& other)  {
    return !(*this == other);
  }
protected:
    node<T>* m_ptrNode;
};

#endif // _MLIST_H_
```

```
// _MLIST_H_
template <typename T>
class const_iterator{
public:
...
  bool operator ==(const iterator& other){
    return m_ptrNode == other.m_ptrNode; /* dozvoljen pristup
privatnom clanu od strane prijatelja */
  }
  bool operator !=(const iterator& other)  {
    return !(*this == other);
  }
protected:
    node<T>* m_ptrNode;
};

#endif // _MLIST_H_
```

```
// _MLIST_H_
template <typename T>
class list{
private:
  node<T>* m_ptrHead;  // Pokazivac na prvi element liste
  node<T>* m_ptrTail;  // Pokazivac na poslednji element liste
  int m_size;  // Broj elemenata u listi
public:
  //typedef deklaracije aliasa (drugih imena tipova)
  typedef iterator<T> iterator;
  typedef typename iterator::value_type value_type;
  typedef typename iterator::reference reference;
  typedef typename iterator::pointer pointer;

  typedef const_iterator<T> const_iterator;
  typedef typename const_iterator::reference const_reference;
  typedef typename const_iterator::pointer const_pointer;

  size_t size() const { return m_size;}
  bool empty() const { return m_size == 0;}
.. . ..
};
#endif
```

```
// _MLIST_H_
template <typename T>
class list{
public:
  /* Podrazumevani konstruktor */
  list(): m_ptrHead(NULL), m_ptrTail(NULL), m_size(0){}
  /* Konstruktor kopije */
  list(const list<T>& other){
    for (const_iterator it = other.begin(); it != other.end(); ++it){
      push_back(*it);
    }
  }
  /* Operator dodele vrednosti */
  list<T>& operator=(const list<T>& other){
    clear();
    for (const_iterator it = other.begin(); it != other.end(); ++it){
      push_back(*it);
    }
    return *this;
  }
};
#endif
```

```
// _MLIST_H_
template <typename T>
class list{
public:
  ~list(){ clear();} //Destruktor
   void clear(){ while (begin() != end()) pop_front();}

  iterator begin(){return iterator(m_ptrHead);}
  iterator end() {return iterator(NULL);}

  const_iterator begin() const {return const_iterator(m_ptrHead);}
  const_iterator end() const {return const_iterator(NULL);}

  /* DODAVANJE ELEMENTA NA POCETAK LISTE
  void push_front(const_reference elem){
    m_ptrHead = new node<T>(elem, m_ptrHead);
    if(m_ptrTail == NULL) m_ptrTail = m_ptrHead;
    ++m_size;
  }
. . . .
};
#endif
```

```
// _MLIST_H_
template <typename T>
class list{
public:
  void pop_front(){/* BRISANJE ELEMENTA SA POCETKA LISTE */
    if(m_size){ /* lista nije prazna */
      node<T>* ptrToDel = m_ptrHead;
      m_ptrHead = m_ptrHead->next;
      if (m_ptrHead == NULL) m_ptrTail = NULL;
      --m_size;
      delete ptrToDel;
    }
  }
  . . . .
};
#endif
```

```
// _MLIST_H_
template <typename T>
class list{
public:
/* DODAVANJE NOVOG ELEMENTA IZA NEKOG ELEMENTA LISTE
   pos - pozicija elementa iza koga se dodaje novi
   elem - info sadrzaj novog elementa
   rezultat je pozicija novog elementa u listi  */

  iterator insert_after(const const_iterator &pos, const_reference
elem){
   if(pos.m_ptrNode){ /* Iterator nije singularan */
     pos.m_ptrNode->next =
          new node<value_type>(elem, pos.m_ptrNode->next);
     if(m_ptrTail == pos.m_ptrNode)  m_ptrTail = pos.m_ptrNode->next;
     ++m_size;
     return iterator(pos.m_ptrNode->next);
   }else {
     printf(" Singularan iterator “);
     exit(1);
   }
  }
. . . . .
```

```
// _MLIST_H_
template <typename T>
class list{
public:

  // DODAVANJE ELEMENTA NA KRAJ LISTE
  void push_back(const_reference elem){
    if (m_ptrTail != NULL) insert_after(iterator(m_ptrTail), elem);
    else push_front(elem);
  }

#endif
```

```
// _MLIST_H_
template <typename T>
class list{
public:

/* BRISANJE ELEMENTA IZA NEKOG ELEMENTA LISTE
 pos - pozicija elementa iza koga se brise
  rezultat je pozicija elementa koji je dosao na mesto obrisanog */
  iterator erase_after(const const_iterator &pos){
    if(pos.m_ptrNode && pos.m_ptrNode!=m_ptrTail){
      node<value_type>* ptrToDel = pos.m_ptrNode->next;
      pos.m_ptrNode->next = pos.m_ptrNode->next->next;
      --m_size;
      if (pos.m_ptrNode->next == NULL)
        m_ptrTail = pos.m_ptrNode;
      delete ptrToDel;
      return iterator(pos.m_ptrNode->next);
    } else return end(); /* vracamo end iterator */
  }
};
#endif
```

```
// _MLIST_H_
template <typename T>
class list{
public:
/* BRISANJE ELEMENTA NA DATOJ POZICIJI: pos - pozicija na kojoj se briše
element; rezultat je pozicija elementa koji je dosao na mesto obrisanog
*/
  iterator erase(const const_iterator &pos){
    if(pos.m_ptrNode){
       node<value_type>* ptrToDel = m_ptrHead;
       if(m_ptrHead == pos.m_ptrNode){
          m_ptrHead = m_ptrHead->next;
          delete ptrToDel;
          return iterator(m_ptrHead);
       }
       else{
            while(ptrToDel && ptrToDel->next != pos.m_ptrNode)
                ptrToDel = ptrToDel->next;
         /* erase_after resava problem ako ptrToDel ima vrednost NULL */
          return erase_after(iterator(ptrTodel));
       }
    }
  }
};
```

```
template <typename T>
void print_list(const list<T>& lst){
  typename list<T>::const_iterator it;
  for (it = lst.begin(); it != lst.end(); ++it)
    std::cout << *it << " ";
  std::cout << std::endl;
}
```

```
 int main(){

   list<int> lst;
   lst.push_back(2);
   lst.push_back(3);
   lst.push_front(1);
   lst.push_back(4);
   print_list(lst);
   list<int>::iterator it;
   it = lst.begin();
   it++;
   lst.insert_after(it, 42);
   ++it;
   ...
 }
```

```
int main(){
  // NASTAVAK main
  ...
  print_list(lst);
  lst.erase_after(++it);
  print_list(lst);
  it = lst.begin();
  *it = 10;
  print_list(lst);
  lst.clear();
  print_list(lst);
  return 0;
}
```

```
#ifndef _MLIST_H_
#define _MLIST_H_
// Sabloni struktura koje omogucavaju da razliciti tipovi budu vezani
za jedan alias cndt::type

template<bool VAL, typename FIRST_T, typename SECOND_T>
struct cndt { typedef FIRST_T type; };

// Specijalizacija prvog sablona za VAL = false
template<typename FIRST_T, typename SECOND_T>
struct cndt<false, FIRST_T, SECOND_T> { typedef SECOND_T type; };

template <typename T>
class node{
  friend list<T>;
  friend typename list<T>::iterator;
  friend typename list<T>::const_iterator;
  node(T val): next(NULL), value(val){}
  node(T val, node<T> *p): next(p), value(val){}
  node<T>* next;
   T value;
};
```

```
#ifndef _MLIST_H_
#define _MLIST_H_
//Sablon iz koga nastaju iterator i const_iterator
template <typename T, bool is_const = false>
class c_nc_iterator{
public:
  friend list<T>;
  friend c_nc_iterator<T, ~is_const>;

  typedef T  value_type;
  typedef typename cndt<is_const, const T*, T*>::type pointer;
  typedef typename cndt<is_const, const T&, T&>::type reference;

  c_nc_iterator(node<T>* ptr = NULL) : m_ptrNode(ptr)  {}
  c_nc_iterator(const c_nc_iterator<T>& other) :
m_ptrNode(other.m_ptrNode) {} /* Konstruktor kopije koji omogucava
neophodno kopiranje iterator-a u const_iterator */

  };
....
#endif
```

```
#ifndef _MLIST_H_
#define _MLIST_H_
//Sablon iz koga nastaju iterator i const_iterator
template <typename T, bool is_const = false>
class c_nc_iterator{
public:
. . . .
  reference operator*(){ return m_ptrNode->value; }
  pointer operator->(){ return m_ptrNode; }

  c_nc_iterator<T,is_const>& operator++(){
    m_ptrNode = m_ptrNode->next;
    return *this;
  }
  c_nc_iterator<T,is_const> operator++(int){
    c_nc_iterator<T, is_const> toRet = *this;
    ++(*this);
    return toRet;
  }
. . . .
};
. . . .
```

```
#ifndef _MLIST_H_
#define _MLIST_H_
//Sablon iz koga nastaju iterator i const_iterator
template <typename T, bool is_const = false>
class c_nc_iterator{
public:
. . . .
  bool operator ==(const c_nc_iterator<T,is_const>& other){
    return m_ptrNode == other.m_ptrNode;
  }

  bool operator !=(const c_nc_iterator<T, is_const>& other)  {
    return !(*this == other);
  }

protected:
  node<T>* m_ptrNode;
};
....
#endif
```

```
// _MLIST_H_
template <typename T>
class list{
. . . .
public:
  //izmene u odnosu na prethodnu implementaciju u typedef
deklaracijama aliasa (drugih imena tipova) u sablonu klase list

  // Preostali deo sablona je nepromenjen

  typedef c_nc_iterator<T> iterator;
  typedef typename iterator::value_type value_type;
  typedef typename iterator::reference reference;
  typedef typename iterator::pointer pointer;


  typedef c_nc_iterator<T,true> const_iterator;
  typedef typename const_iterator::reference const_reference;
  typedef typename const_iterator::pointer const_pointer;

.. . ..
};
#endif
```