Hijerarhijska dekompozicija

- Za *polimorfne klase* (*polymorphc class*), a to su klase sa bar jednom (makar i nasleđenom) virtuelnom funkcijom, odnosno klase čiji objekti imaju u sebi VTP, ovakav *downcast* se može izvršiti i bezbednije, *dinamičkom konverzijom* (*dynamic cast*):

```
Circle* crc = dynamic_cast<Circle*>(fig);
```

- Ukoliko se iza pokazivača kojim rezultuje izraz unutar zagrada ovog operatora krije zaista objekat koji jeste (direktna ili indirektna) instanca tražene ciljne klase, rezultat će biti validan pokazivač na objekat te klase; u suprotnom, rezultat će biti nula pokazivač (*null*) (ako je odredišni tip referenca, u ovom slučaju biće bačen izuzetak)

- Upotreba dinamičke konverzije u ovakve svrhe je opravdana u situacijama kada se zna ili se očekuje da je iza pokazivača instanca potrebne specijalizacije; međutim, pogrešna upotreba može da signalizira propuštanje neke apstrakcije i polimorfizma:

```
Circle* crc = dynamic_cast<Circle*>(fig);
if (crc) drawCircle(crc);

Rectangle* rct = dynamic_cast<Rectangle*>(fig);
if (rct) drawRectangle(rct);
…
```

Očigledno polimorfizam!

Hijerarhijska dekompozicija

- Posmatrajmo sledeći zahtev:
  - realizujemo apstraktnu strukturu podataka *lista* (*list*), koja će imati operacije smeštanja novog elementa na proizvoljnu poziciju u listi, iza nekog drugog elementa u toj listi, i uzimanja elementa sa proizvoljnog mesta u listi
  - želimo da obe operacije budu kompleksnosti $O(1)$, pa ćemo koristiti dvostruko ulančanu listu
  - ne želimo da za strukture pokazivača koje koristimo dinamički alociramo potreban prostor, već želimo da te pokazivače ugradimo u same objekte koji će biti ulančavani, koji god da su; zato nećemo koristiti šablone, pa ni one bibliotečne
  - pravimo *spisak zadataka* (*task list*), u koju ćemo smeštati *zadatke* (*task*)
  - *zadatak* ima i mnoge druge osobine, ponašanje, koristi se u mnogim drugim, različitim kontekstima aplikacije itd.

```
class ListElem {
public:
  void insert (ListElem* prev, ListElem* next);

protected:
  ListElem () { prev = next = nullptr; };

private:
  friend class List;
  ListElem *prev, *next;
};

void ListElem::insert (ListElem* p, ListElem* n) {
  if (p) p->next = this;
  if (n) n->prev = this;
  this->prev = p;
  this->next = n;
}

class List {
public:
  List () { head = tail = nullptr; }

  void addAtHead (ListElem* e);
  void addAtTail (ListElem* e);
  void addAfter  (ListElem* e, ListElem* prev);

private:
  ListElem *head, *tail;
};

void List::addAfter (ListElem* e, ListElem* p) {
  if (!e) return;
  if (!p) insertAtHead(e);
  else
    if (!p->next) insertAtTail(e);
    else e->insert(p,p->next);
}
```