Predavanje 5.

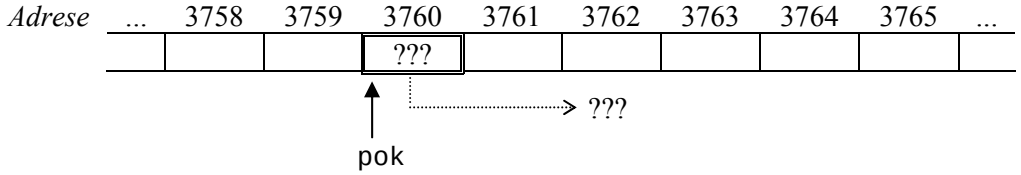
Jezik C++ je znatno osavremenio mehanizme za *dinamičku alokaciju* (*dodjelu*) *memorije* u odnosu na jezik C. Pod dinamičkom alokacijom memorije podrazumijevamo mogućnost da program *u toku izvršavanja* zatraži da mu se dodijeli određena količina memorije koja ranije nije bila zauzeta, i kojom on može raspolagati sve dok eventualno ne zatraži njeno oslobađanje. Takvim, dinamički dodijeljenim blokovima memorije, može se pristupati *isključivo pomoću pokazivača*. Dinamička alokacija memorije se u jeziku C ostvaruje pozivom funkcije "`malloc`" ili neke srodne funkcije iz biblioteke "`cstdlib`" (zaglavlje ove biblioteke u jeziku C zove se "`stdlib.h`"). Mada ovaj način za dinamičku alokaciju načelno radi i u jeziku C++, C++ nudi mnogo fleksibilnije načine, tako da u C++ programima po svaku cijenu treba izbjegavati načine za dinamičku dodjelu memorije naslijeđene iz jezika C. Također treba napomenuti da je potreba za dinamičkom alokacijom memorije u svakodnevnim primjenama uveliko opala nakon što su u C++ uvedeni *dinamički tipovi podataka*, kao što su "`vector`", "`string`", itd. Međutim, dinamički tipovi podataka su izvedeni tipovi podataka, koji su definirani u standardnoj biblioteci jezika C++, i koji su implementirani upravo pomoću dinamičke alokacije memorije! Drugim riječima, da ne postoji dinamička alokacija memorije, ne bi bilo moguće ni kreiranje tipova poput "`vector`" i "`string`". Stoga, ukoliko želimo u potpunosti da ovladamo ovim tipovima podataka i da samostalno kreiramo *vlastite tipove podataka* koji su njima srodni, moramo shvatiti mehanizam koji stoji u pozadini njihovog funkcioniranja, a to je upravo dinamička alokacija memorije! Pored toga, vidjećemo kasnije da dinamička alokacija memorije može biti korisna za izgradnju drugih složenijih tipova podataka, koje nije moguće jednostavno svesti na postojeće dinamičke tipove podataka.

U jeziku C++, preporučeni način za dinamičku alokaciju memorije je pomoću operatora "**`new`**", koji ima više različitih oblika. U svom osnovnom obliku, iza njega treba da slijedi *ime nekog tipa*. On će tada potražiti u memoriji slobodno mjesto u koje bi se mogao smjestiti podatak navedenog tipa. Ukoliko se takvo mjesto *pronađe*, operator "**`new`**" će kao rezultat vratiti *pokazivač na pronađeno mjesto u memoriji*. Pored toga, pronađeno mjesto će biti označeno kao *zauzeto*, tako da se neće moći desiti da isti prostor u memoriji slučajno bude upotrijebljen za neku drugu namjenu. Ukoliko je sva memorija već zauzeta, operator "**`new`**" će *baciti izuzetak*, koji možemo "uhvatiti". Raniji standardi jezika C++ predviđali su da operator "**`new`**" u slučaju neuspjeha vrati kao rezultat *nul-pokazivač*, ali je ISO 98 standard jezika C++ precizirao da u slučaju neuspjeha operator "**`new`**" baca izuzetak (naime, programeri su često zaboravljali ili su bili lijeni da nakon svake upotrebe operatora "**`new`**" provjeravaju da li je vraćena vrijednost možda nul-pokazivač, što je, u slučaju da alokacija ne uspije, moglo imati ozbiljne posljedice).

Osnovnu upotrebu operatora "**`new`**" najbolje je ilustrirati na konkretnom primjeru. Pretpostavimo da nam je data sljedeća deklaracija:

```
int *pok;
```

Ovom deklaracijom definirana je pokazivačka promjenljiva "`pok`" koja inicijalno ne pokazuje ni na šta konkretno, ali koja, u načelu, treba da pokazuje na *cijeli broj*. Stanje memorije nakon ove deklaracije možemo predstaviti sljedećom slikom (adrese su pretpostavljene proizvoljno):

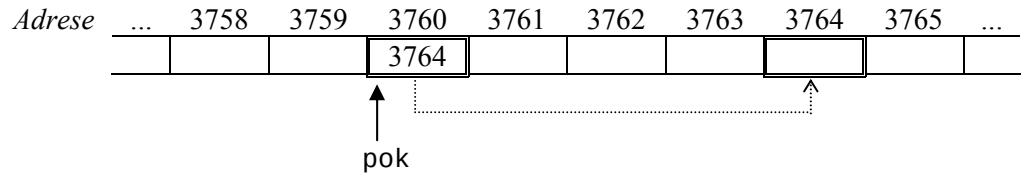


Ukoliko sad izvršimo naredbu

```
pok = new int;
```

operator "**`new`**" će potražiti slobodno mjesto u memoriji koje je dovoljno veliko da prihvati *jedan cijeli broj*. Ukoliko se takvo mjesto pronađe, njegova adresa će biti vraćena kao rezultat operatora "**`new`**", i

dodijeljena pokazivaču "pok". Pretpostavimo, na primjer, da je slobodno mjesto pronađeno na adresi 3764. Tada će rezultat operatora "**new**" biti *pokazivač na cijeli broj koji pokazuje na adresu 3764*, koji se može dodijeliti pokazivaču "pok", tako da će on pokazivati na adresu 3764. Stanje u memoriji će sada izgledati ovako:



Pored toga, lokacija 3764 postaje *zauzeta*, u smislu da će biti evidentirano da je ova lokacija rezervirana za upotrebu od strane programa, i ona do daljnjeg sigurno neće biti iskorištena za neku drugu namjenu. Stoga je sasvim sigurno pristupiti njenom sadržaju putem pokazivača "pok". Znamo da se svaki dereferencirani pokazivač u potpunosti ponaša *kao promjenljiva* (preciznije, dereferencirani pokazivač je *l-vrijednost*) iako lokacija na koju on pokazuje *nema svoje vlastito simboličko ime*. Stoga su naredbe poput sljedećih posve korektne (i ispisaće redom vrijednosti 5 i 18):

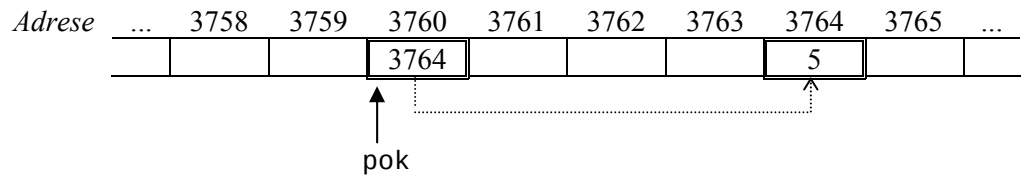
```
*pok = 5;
cout << *pok << endl;
*pok = 3 * *pok + 2;
(*pok)++;                  // Oprez: ovo nije isto što i *pok++
cout << *pok << endl;
```

Kako se lokacija rezervirana pomoću operatora "**new**" u potpunosti ponaša kao promjenljiva (osim što nema svoje vlastito ime), kažemo da ona predstavlja *dinamičku promjenljivu*. Dakle, dinamičke promjenljive se *stvaraju* primjenom operatora "**new**", i njihovom sadržaju se može pristupiti *isključivo putem pokazivača* koji na njih pokazuju. Pored *automatskih promjenljivih*, koje se automatski stvaraju na mjestu deklaracije i automatski uništavaju na kraju bloka u kojem su deklarirane (takve su sve *lokalne promjenljive* osim onih koje su deklarirane sa atributom "**static**"), i *statičkih promjenljivih* koje "žive" cijelo vrijeme dok se program izvršava (takve su sve *globalne promjenljive* i lokalne promjenljive deklarirane sa atributom "**static**"), dinamičke promjenljive se mogu smatrati za treću vrstu promjenljivih koje postoje. One se *stvaraju na zahtjev*, i kao što ćemo uskoro vidjeti, *uništavaju na zahtjev*. Samo, za razliku od automatskih i statičkih promjenljivih, dinamičke promjenljive nemaju imena (stoga im se može pristupiti samo pomoću pokazivača).

Sadržaj dinamičkih promjenljivih je nakon njihovog stvaranja tipično *nedefiniran*, sve dok im se ne izvrši prva dodjela vrijednosti (putem pokazivača). Preciznije, sadržaj zauzete lokacije zadržava svoj raniji sadržaj, jedino što se lokacija označava kao zauzeta. Međutim, moguće je izvršiti inicijalizaciju dinamičke promjenljive *odmah po njenom stvaranju*, tako što iza oznake tipa u operatoru "**new**" u zagradama navedemo izraz koji predstavlja željenu *inicijalnu vrijenost promjenljive*. Na primjer, sljedeća naredba će stvoriti novu cjelobrojnu dinamičku promjenljivu, inicijalizirati njen sadržaj na vrijednost 5, i postaviti pokazivač "pok" da pokazuje na nju:

```
pok = new int(5);
```

Sada će stanje u memoriji biti kao na sljedećoj slici:



Razumije se da se rezultat operatora "**new**" mogao odmah iskoristiti za inicijalizaciju pokazivača "pok" već prilikom njegove deklaracije, kao na primjer u sljedećim deklaracijama:

```
int *pok = new int(5);        // Ove dvije deklaracije su
int *pok(new int(5));         //   ekvivalentne...
```

Za dinamičku promjenljivu je, kao i za svaku drugu promjenljivu, moguće vezati *referencu*, čime možemo indirektno *dodijeliti ime* novostvorenoj dinamičkoj promjenljivoj. Tako, ukoliko je "pok" pokazivačka promjenljiva koja pokazuje na novostvorenu dinamičku promjenljivu, kao u prethodnom primjeru, moguće je izvršiti sljedeću deklaraciju:

```
int &dinamicka(*pok);
```

Na ovaj način smo kreirali referencu "dinamicka" koja je vezana za novostvorenu dinamičku promjenljivu, i koju, prema tome, možemo smatrati kao alternativno ime te dinamičke promjenljive (zapravo, *jedino ime*, s obzirom da ona svog vlastitog imena i nema). U suštini, isti efekat smo mogli postići i vezivanjem reference direktno na dereferencirani rezultat operatora "**new**" (bez posredstva pokazivačke promjenljive). Naime, rezultat operatora "**new**" je *pokazivač*, dereferencirani pokazivač je *l-vrijednost*, a na svaku l-vrijednost se može vezati referenca odgovarajućeg tipa:

```
int &dinamicka(*new int(5));
```

Ovakve konstrukcije se ne koriste osobito često, mada se ponekad mogu korisno upotrijebiti (npr. dereferencirani rezultat operatora "**new**" može se prenijeti u funkciju kao parametar po referenci). U svakom slučaju, opis ovih konstrukcija pomaže da se shvati prava suština pokazivača i referenci.

Prilikom korištenja operatora "**new**", treba voditi računa da uvijek postoji mogućnost da on *baci izuzetak*. Doduše, vjerovatnoća da u memoriji neće biti pronađen prostor za jedan jedini cijeli broj veoma je mala, ali se treba naviknuti na takvu mogućnost, jer ćemo kasnije dinamički alocirati znatno glomaznije objekte (npr. nizove) za koje lako može nestati memorije. Stoga bi se svaka dinamička alokacija trebala izvoditi unutar "**try**" bloka, na način koji je principijelno prikazan u sljedećem isječku:

```
try {
  int *pok(new int(5));
  …
}
catch(...) {
  cout << "Problem: Nema dovoljno memorije!\n";
}
```

Tip izuzetka koji se pri tome baca je tip "bad_alloc". Ovo je izvedeni tip podataka (poput tipa "string" i drugih izvedenih tipova podataka) definiran u biblioteci "new". Tako, ukoliko želimo da specifiziramo da hvatamo baš izuzetke ovog tipa, možemo koristiti sljedeću konstrukciju (pod uvjetom da smo uključili zaglavlje biblioteke "new" u program):

```
try {
  int *pok(new int(5));
  …
}
catch(bad_alloc) {
  cout << "Problem: Nema dovoljno memorije!\n";
}
```

Primijetimo da nismo imenovali parametar tipa "bad_alloc", s obzirom da nam on nije ni potreban. Inače, specifikacija tipa "bad_alloc" unutar "**catch**" bloka je korisna ukoliko želimo da razlikujemo izuzetke koje baca operator "**new**" od drugih izuzetaka. Ukoliko smo sigurni da jedini mogući izuzetak unutar "**try**" bloka može poteći samo od operatora "**new**", možemo koristiti varijantu "**catch**" bloka sa tri tačke umjesto formalnog parametra, što ćemo ubuduće pretežno koristiti.

Dinamičke promjenljive se mogu ne samo *stvarati*, već i *uništavati* na zahtjev. Onog trenutka kada zaključimo da nam dinamička promjenljiva na koju pokazuje pokazivač više nije potrebna, možemo je uništiti primjenom operatora "**delete**", iza kojeg se prosto navodi pokazivač koji pokazuje na dinamičku promjenljivu koju želimo da uništimo. Na primjer, naredbom

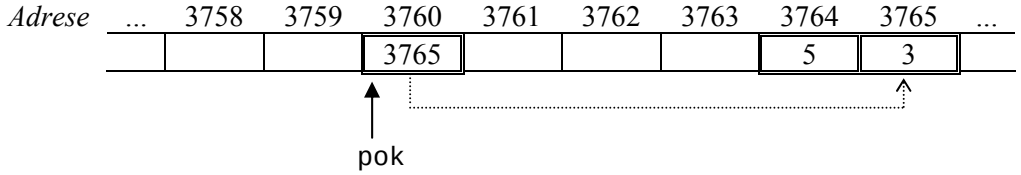
```
delete pok;
```

uništićemo dinamičku promjenljivu na koju pokazivač "pok" pokazuje. Pri tome je važno da razjasnimo šta se podrazumijeva pod tim *uništavanjem*. Pokazivač "pok" će i dalje pokazivati na istu adresu na koju je pokazivao i prije, samo što će se izbrisati evidencija o tome da je ta lokacija zauzeta. Drugim riječima, ta lokacija može nakon uništavanja dinamičke promjenljive biti iskorištena od strane nekog drugog (npr. operativnog sistema), pa čak i od strane samog programa za neku drugu svrhu (na primjer, sljedeća primjena operatora "**new**" može ponovo iskoristiti upravo taj prostor za stvaranje neke druge dinamičke promjenljive). Stoga, više nije sigurno pristupati sadržaju na koju "pok" pokazuje. Pokazivači koji pokazuju na objekte koji su iz bilo kojeg razloga *prestali da postoje* nazivaju se ***viseći pokazivači*** (engl. *dangling pointers*). Jedan od mogućih načina da kreiramo viseći pokazivač je eksplicitno uništavanje sadržaja na koji pokazivač pokazuje pozivom operatora "**delete**", ali postoje i mnogo suptilniji načini da stvorimo viseći pokazivač (npr. da iz funkcije kao rezultat vratimo pokazivač na neki lokalni objekat unutar funkcije, koji prestaje postojati po završetku funkcije). Treba se čuvati visećih pokazivača, jer su oni veoma čest uzrok fatalnih grešaka u programima, koje se teško otkrivaju, s obzirom da im se posljedice obično uoče tek naknadno. Zapravo, ranije smo vidjeli da postoje i *viseće reference*, koje su jednako opasne kao i viseći pokazivači, samo što je viseći pokazivač, na žalost, mnogo lakše "napraviti" od viseće reference!

Sve dinamičke promjenljive se automatski uništavaju po završetku programa. Međutim, bitno je napomenuti da ni jedna dinamička promjenljiva *neće biti uništena sama od sebe* prije završetka programa, osim ukoliko je eksplicitno ne uništimo primjenom operatora "**delete**". Po tome se one bitno razlikuju od *automatskih promjenljivih*, koje se automatski uništavaju na kraju bloka u kojem su definirane. Ukoliko smetnemo sa uma ovu činjenicu, možemo zapasti u probleme. Pretpostavimo, na primjer, da smo izvršili sljedeću sekvencu naredbi:

```
int *pok(new int);
*pok = 5;
pok = new int;
*pok = 3;
```

U ovom primjeru, prvo se stvara jedna dinamička promjenljiva (recimo, na adresi 3764), nakon čega se njen sadržaj postavlja na vrijednost 5. Iza toga slijedi nova dinamička alokacija kojom se stvara nova dinamička promjenljiva (recimo, na adresi 3765), a njen sadržaj se postavlja na vrijednost 3. Pri tome, pokazivač "pok" pokazuje na novostvorenu dinamičku promjenljivu (na adresi 3765). Međutim, dinamička promjenljiva na adresi 3764 *nije uništena*, odnosno dio memorije u kojem se ona nalazi još uvijek se smatra zauzetim. Kako pokazivač "pok" više ne pokazuje na nju, njenom sadržaju više ne možemo pristupiti preko ovog pokazivača. Zapravo, na ovu dinamičku promjenljivu više ne pokazuje *niko*, tako da je ova dinamička promjenljiva postala *izgubljena* (niti joj možemo pristupiti, niti je možemo uništiti). Ova situacija prikazana je na sljedećoj slici:



Dio memorije koji zauzima izgubljena dinamička promjenljiva ostaje rezerviran sve do kraja programa, i trajno je izgubljen za program. Ovakva pojava naziva se *curenje memorije* (engl. *memory leak*) i predstavlja dosta čestu grešku u programima. Mada je curenje memorije manje fatalno u odnosu greške koje nastaju usljed visećih pokazivača, ono se također teško uočava i može da dovede do ozbiljnih problema. Razmotrimo na primjer sljedeću sekvencu naredbi:

```
int *pok;
for(int i = 1; i <= 30000; i++) {
  pok = new int;
  *pok = i;
}
```

U ovom primjeru, unutar "**for**" petlje je stvoreno 30000 dinamičkih promjenljivih, jer je svaka primjena operatora "**new**" stvorila novu dinamičku promjenljivu, od kojih niti jedna nije uništena (s obzirom da nije korišten operator "**delete**"). Međutim, od tih 30000 promjenljivih, 29999 je izgubljeno, jer na kraju pokazivač "`pok`" pokazuje samo na posljednju stvorenu promjenljivu! Uz pretpostavku da jedna cjelobrojna promjenljiva zauzima 4 bajta, ovim smo bespotrebno izgubili 119996 bajta memorije, koje ne možemo osloboditi, sve dok se ne oslobode automatski po završetku programa!

Jedna od tipičnih situacija koje mogu dovesti do curenja memorije je stvaranje dinamičke promjenljive unutar neke funkcije preko pokazivača koji je lokalna automatska (nestatička) promjenljiva unutar te funkcije. Ukoliko se takva dinamička promjenljiva ne uništi prije završetka funkcije, pokazivač koji na nju pokazuje biće uništen (poput svake druge automatske promjenljive), tako da će ona postati izgubljena. Na primjer, sljedeća funkcija demonstrira takvu situaciju:

```
void Curenje(int n) {
  int *pok(new int);
  *pok = n;
}
```

Prilikom svakog poziva ove funkcije stvara se nova dinamička promjenljiva, koja postaje posve nedostupna nakon završetka funkcije, s obzirom da se pokazivač koji na nju pokazuje uništava. Ostatak programa nema mogućnost niti da pristupi takvoj promjenljivoj, niti da je uništi, tako da svaki poziv funkcije "`Curenje`" stvara novu izgubljenu promjenljivu, odnosno prilikom svakog njenog poziva količina izgubljene memorije se povećava. Stoga bi svaka funkcija koja stvori neku dinamičku promjenljivu trebala i da je uništi. Izuzetak od ovog pravila može imati smisla jedino ukoliko se novostvorena dinamička promjenljiva stvara putem *globalnog pokazivača* (kojem se može pristupiti i izvan funkcije), ili ukoliko funkcija *vraća kao rezultat* pokazivač na novostvorenu promjenljivu. Na primjer, sljedeća funkcija može imati smisla:

```
int *Stvori(int n) {
  int *pok(new int(n));
  return pok;
}
```

Ova funkcija stvara novu dinamičku promjenljivu, inicijalizira je na vrijednost zadanu parametrom, i *vraća kao rezultat* pokazivač na novostvorenu promjenljivu (zanemarimo to što je ova funkcija posve beskorisna, s obzirom da ne radi ništa više u odnosu na ono što već radi sam operator "**new**"). Na taj način omogućeno je da rezultat funkcije bude dodijeljen nekom pokazivaču, preko kojeg će se kasnije moći pristupiti novostvorenoj dinamičkoj promjenljivoj, i eventualno izvršiti njeno uništavanje. Na primjer, sljedeći slijed naredbi je posve smislen:

```
int *pok;
pok = Stvori(10);
cout << *pok;
delete pok;
```

Napomenimo da se funkcija "`Stvori`" mogla napisati i kompaktnije, bez deklariranja lokalnog pokazivača:

```
int *Stvori(int n) {
  return new int(n);
}
```

Naime, "**new**" je *operator* koji vraća pokazivač kao rezultat, koji kao takav smije biti vraćen kao rezultat iz funkcije. Također, interesantno je napomenuti da je na prvi pogled veoma neobična konstrukcija

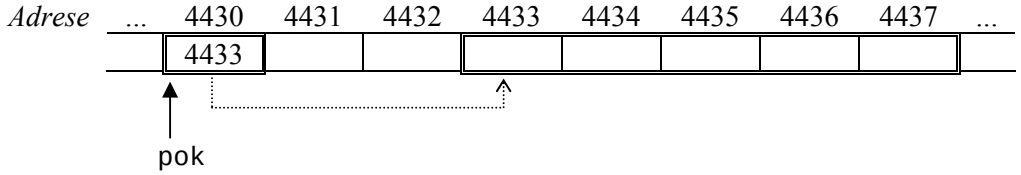
```
*Stvori(5) = 8;
```

sintaksno posve ispravna (s obzirom da funkcija "`Stvori`" vraća kao rezultat *pokazivač*, a dereferencirani pokazivač je *l-vrijednost*), mada je logički smisao ovakve konstrukcije prilično upitan (prvo se stvara dinamička promjenljiva sa vrijednošću 5, nakon toga se njena vrijednost mijenja na 8, i na kraju se gubi svaka veza sa dinamičkom promjenljivom, jer pokazivač na nju nije nigdje sačuvan). Ipak, mogu se pojaviti situacije u kojima slične konstrukcije mogu biti od koristi, stoga je korisno znati da su one moguće.

Kreiranje individualnih dinamičkih promjenljivih nije od osobite koristi ukoliko se kreiraju promjenljive prostih tipova (kao što su npr. cjelobrojne promjenljive), tako da će kreiranje individualnih dinamičkih promjenljivih postati interesantno tek kada razmotrimo složenije tipove podataka, kakvi su *strukture* i *klase*. Međutim, znatno je interesantnija činjenica da se pomoću dinamičke alokacije memorije mogu indirektno kreirati nizovi čija veličina *nije unaprijed poznata* (ovaj mehanizam zapravo leži u pozadini funkcioniranja tipova poput tipa "`vector`"). Za tu svrhu koristi se također operator "**new**" iza kojeg ponovo slijedi *ime tipa* (koje ovaj put predstavlja *tip elemenata niza*), nakon čega *u uglastim zagradama* slijedi broj elemenata niza koji kreiramo. Pri tome, traženi broj elemenata niza *ne mora biti konstanta*, već može biti proizvoljan izraz. Operator "**new**" će tada potražiti slobodno mjesto u memoriji u koje bi se mogao smjestiti niz tražene veličine navedenog tipa, i vratiti kao rezultat pokazivač na pronađeno mjesto u memoriji, ukoliko takvo postoji (u suprotnom će biti bačen izuzetak tipa "`bad_alloc`"). Na primjer, ukoliko je promjenljiva "`pok`" deklarirana kao pokazivač na cijeli broj (kao i u dosadašnjim primjerima), tada će naredba

```
pok = new int[5];
```

potražiti prostor u memoriji koji je dovoljan da prihvati *pet cjelobrojnih vrijednosti*, i u slučaju da pronađe takav prostor, *dodijeliće njegovu adresu* pokazivaču "`pok`" (u uglastoj zagradi nije morala biti konstanta 5, nego je mogao biti i proizvoljan cjelobrojni izraz). Na primjer, neka je pronađen prostor na adresi 4433, a neka se sama pokazivačka promjenljiva "`pok`" nalazi na adresi 4430. Tada memorijska slika nakon uspješnog izvršavanja prethodne naredbe izgleda kao na sljedećoj slici (radi jednostavnosti je pretpostavljeno da jedna cjelobrojna promjenljiva zauzima jednu memorijsku lokaciju, što u stvarnosti nije ispunjeno):



Ovako kreiranom "dinamičkom" nizu možemo pristupiti samo preko pokazivača. Međutim, kako se na pokazivače mogu primjenjivati operatori indeksiranja (podsjetimo se da se izraz "`pok[n]`" u slučaju kada je "`pok`" pokazivač interpretira kao "`*(pok + n)`" uz primjenu pokazivačke aritmetike), dinamički niz možemo koristiti *na posve isti način kao i obični niz*, pri čemu *umjesto imena niza koristimo pokazivač*. Na primjer, da bismo postavili sve elemente novokreiranog dinamičkog niza na nulu, možemo koristiti "**for**" petlju kao da se radi o običnom nizu:

```
for(int i = 0; i < 5; i++) pok[i] = 0;
```

Stoga, sa aspekta programera gotovo da nema nikakve razlike između korištenja običnih i dinamičkih nizova. Strogo rečeno, dinamički nizovi *nemaju imena* i njima se može pristupati samo preko pokazivača koji pokazuju na njihove elemente. Tako, ukoliko izvršimo deklaraciju poput

```
int *dinamicki_niz(new int[100]);
```

mi zapravo stvaramo *dva objekta*: dinamički niz od 100 cijelih brojeva koji *nema ime*, i pokazivač "`dinamicki_niz`" koji je inicijaliziran tako da pokazuje na *prvi element ovako stvorenog dinamičkog niza*. Međutim, kako promjenljivu "`dinamicki_niz`" možemo koristiti na gotovo isti način kao da se radi o običnom nizu (pri čemu, zahvaljujući pokazivačkoj aritmetici, preko nje zaista pristupamo dinamičkom nizu), dopustićemo sebi izvjesnu slobodu izražavanja i ponekad ćemo, radi kratkoće izražavanja, govoriti da promjenljiva "`dinamicki_niz`" predstavlja dinamički niz (iako je prava istina da je ona zapravo pokazivač koji pokazuje na prvi element dinamičkog niza).

Za razliku od običnih nizova koji se mogu inicijalizirati *pri deklaraciji*, i običnih dinamičkih promjenljivih koje se mogu inicijalizirati *pri stvaranju*, dinamički nizovi se ne mogu inicijalizirati u trenutku stvaranja (tj. njihov sadržaj je nakon stvaranja nepredvidljiv). Naravno, inicijalizaciju je moguće uvijek naknadno izvršiti *ručno* (npr. petljom iz prethodnog primjera). Također, nije problem napisati funkciju koja će stvoriti niz, inicijalizirati ga, i vratiti kao rezultat pokazivač na novostvoreni i inicijalizirani niz. Tada tako napisanu funkciju možemo koristiti za stvaranje nizova koji će odmah po kreiranju biti i inicijalizirani. Na primjer, razmotrimo sljedeću generičku funkciju:

```
template <typename NekiTip>
NekiTip *StvoriNizPopunjenNulama(int n) {
  NekiTip *pok(new NekiTip[n]);
  for(int i = 0; i < n; i++) pok[i] = 0;
  return pok;
}
```

Uz pomoć ovakve funkcije možemo stvarati inicijalizirane dinamičke nizove proizvoljnog tipa kojem se može dodijeliti nula. Na primjer,

```
double *dinamicki_niz(StvoriNizPopunjenNulama<double>(100));
```

Primijetimo da smo prilikom poziva funkcije eksplicitno morali navesti tip elemenata niza u šiljastim zagradama "`<>`". To je potrebno stoga što iz samog poziva funkcije nije moguće zaključiti šta tip "`NekiTip`" predstavlja, s obzirom da se ne pojavljuje u popisu formalnih parametara funkcije.

Mada je prethodni primjer veoma univerzalan, on se može učiniti još univerzalnijim. Naime, prethodni primjer podrazumijeva da se elementima novostvorenog niza može dodijeliti *nula*. To je tačno ukoliko su elementi niza *brojevi*. Međutim, šta ukoliko želimo da stvorimo npr. niz čiji su elementi *stringovi* (odnosno objekti tipa "`string`") kojima se *ne može* dodijeliti nula? Da bi se povećala univerzalnost generičkih funkcija, uvedena je konvencija da se ime tipa može *pozvati kao funkcija bez parametara*, pri čemu je rezultat takvog poziva *podrazumijevana vrijednost* za taj tip (ovo vrijedi samo za tipove koji posjeduju podrazumijevane vrijednosti, a većina tipova je takva). Na primjer, podrazumijevana vrijednost za sve brojčane tipove je 0, a za tip "`string`" podrazumijevana vrijednost je prazan string. Tako je vrijednost izraza "**`int()`**" jednaka nuli, kao i izraza "**`double()`**" (mada tipovi ovih izraza nisu isti: prvi je tipa "**`int`**" a drugi tipa "**`double`**"), dok je vrijednost izraza "`string()`" prazan string. Stoga ne treba da čudi da će naredba

```
cout << int();
```

ispisati nulu. Zahvaljujući ovoj, na prvi pogled čudnoj konstrukciji, moguće je napisati veoma univerzalnu generičku funkciju poput sljedeće:

```
template <typename NekiTip>
NekiTip *StvoriInicijaliziraniNiz(int n) {
  NekiTip *pok(new NekiTip[n]);
  for(int i = 0; i < n; i++) pok[i] = NekiTip();
  return pok;
}
```

Sa ovako napisanom funkcijom, moguće je pisati konstrukcije poput

```
double *niz_brojeva(StvoriInicijaliziraniNiz<double>(100));
string *niz_stringova(StvoriInicijaliziraniNiz<string>(150));
```

koje će stvoriti dva dinamička niza "niz_brojeva" i "niz_stringova" od 100 i 150 elemenata respektivno, čiji će elementi biti respektivno inicijalizirani nulama, odnosno praznim stringovima.

Kada nam neki dinamički niz više nije potreban, možemo ga također uništiti (tj. osloboditi prostor koji je zauzimao) pomoću operatora "**delete**", samo uz neznatno drugačiju sintaksu u kojoj se koristi par uglastih zagrada. Tako, ukoliko pokazivač "pok" pokazuje na dinamički niz, uništavanje dinamičkog niza realizira se pomoću naredbe

```
delete[] pok;
```

Neophodno je napomenuti da su uglaste zagrade *bitne*. Naime, postupci dinamičke alokacije običnih dinamičkih promjenljivih i dinamičkih nizova interno se obavljaju na potpuno drugačije načine, tako da ni postupak njihovog brisanja nije isti.. Mada postoje situacije u kojima bi se dinamički nizovi mogli obrisati primjenom običnog operatora "**delete**" (bez uglastih zagrada), takvo brisanje je uvijek veoma rizično, pogotovo ukoliko se radi o nizovima čiji su elementi složeni tipovi podataka poput struktura i klasa (na primjer, brisanje niza pomoću običnog operatora "**delete**" sigurno neće biti obavljeno kako treba ukoliko elementi niza posjeduju tzv. *destruktore*, o kojima ćemo govoriti kasnije). Stoga, ne treba mnogo filozofirati, nego se treba držati pravila: dinamički nizovi se uvijek moraju brisati pomoću konstrukcije "**delete[]**". Ovdje treba biti posebno oprezan zbog činjenice da nas kompajler *neće upozoriti* ne upotrijebimo li uglaste zagrade, s obzirom na činjenicu da kompajler *ne može znati* na šta pokazuje pokazivač koji se navodi kao argument operatoru "**delete**".

Već je rečeno da bi dinamička alokacija memorije trebala uvijek da se vrši unutar "**try**" bloka, s obzirom da se može desiti da alokacija ne uspije. Stoga, sljedeći primjer, koji alocira dinamički niz čiju veličinu zadaje korisnik, a zatim unosi elemente niza sa tastature i ispisuje ih u obrnutom poretku, ilustrira kako se ispravno treba raditi sa dinamičkim nizovima:

```
try {
  int br_elemenata;
  cout << "Koliko želite brojeva? ";
  cin >> br_elemenata;
  int *niz(new int[br_elemenata]);
  cout << "Unesite brojeve:\n";
  for(int i = 0; i < br_elemenata; i++) cin >> niz[i];
  cout << "Niz brojeva ispisan naopako glasi:\n";
  for(int i = br_elemenata - 1; i >= 0; i--) cout << niz[i] << endl;
  delete[] niz;
}
catch(...) {
  cout << "Nema dovoljno memorije!\n";
}
```

Obavljanje dinamičke alokacije memorije unutar "**try**" bloka je posebno važno kada se vrši dinamička alokacija *nizova*. Naime, alokacija sigurno neće uspjeti ukoliko se zatraži alokacija niza koji zauzima više prostora nego što iznosi količina slobodne memorije (npr. prethodni primjer će sigurno baciti izuzetak u slučaju da zatražite alociranje niza od recimo 100000000 elemenata)

Treba napomenuti da se rezervacija memorije za čuvanje elemenata vektora i dekova također interno realizira pomoću dinamičke alokacije memorije i operatora "**new**". Drugim riječima, prilikom deklaracije vektora ili dekova također može doći do bacanja izuzetka (tipa "bad_alloc") ukoliko količina raspoložive memorije nije dovoljna da se kreira vektor ili dek odgovarajućeg kapaciteta. Zbog toga bi se i deklaracije vektora i dekova načelno trebale nalaziti unutar "**try**" bloka. Stoga, ukoliko bismo u prethodnom primjeru željeli izbjeći eksplicitnu dinamičku alokaciju memorije, i umjesto nje koristiti tip "vector", modificirani primjer trebao bi izgledati ovako:

```
try {
  int br_elemenata;
  cout << "Koliko želite brojeva? ";
  cin >> br_elemenata;
  vector<int> niz(br_elemenata);
  cout << "Unesite brojeve:\n";
  for(int i = 0; i < br_elemenata; i++) cin >> niz[i];
  cout << "Niz brojeva ispisan naopako glasi:\n";
  for(int i = br_elemenata; i >= 0; i--) cout << niz[i] << endl;
}
catch(...) {
  cout << "Nema dovoljno memorije!\n";
}
```

Izuzetak tipa "`bad_alloc`" također može biti bačen kao posljedica operacija koje povećavaju veličinu vektora ili deka (poput "`push_back`" ili "`resize`") ukoliko se ne može udovoljiti zahtjevu za povećanje veličine (zbog nedostatka memorijskog prostora).

Možemo primijetiti jednu suštinsku razliku između primjera koji koristi operator "**`new`**" i primjera u kojem se koristi tip "`vector`". Naime, u primjeru zasnovanom na tipu "`vector`" ne koristi se operator "**`delete`**". Očigledno, lokalne promjenljive tipa "`vector`" se ponašaju kao i sve druge automatske promjenljive – njihov kompletan sadržaj se uništava nailaskom na kraj bloka u kojem su definirane (uključujući i oslobađanje memorije koja je bila alocirana za potrebe smještanja njihovih elemenata). Kasnije ćemo detaljno razmotriti na koji je način ovo postignuto.

Slično običnim dinamičkim promjenljivim, dinamički nizovi se također uništavaju tek na završetku programa, ili eksplicitnom upotrebom operatora "**`delete[]`**". Stoga, pri njihovoj upotrebi također treba voditi računa da ne dođe do curenja memorije, koje može biti znatno ozbiljnije nego u slučaju običnih dinamičkih promjenljivih. Naročito treba paziti da dinamički niz koji se alocira unutar neke funkcije preko pokazivača koji je lokalna promjenljiva obavezno treba i uništiti prije završetka funkcije, inače će taj dio memorije ostati trajno zauzet do završetka programa, i niko ga neće moći osloboditi (izuzetak nastaje jedino u slučaju ako funkcija vraća kao rezultat pokazivač na alocirani niz – u tom slučaju onaj ko poziva funkciju ima mogućnost da oslobodi zauzetu memoriju kada ona više nije potrebna). Višestrukim pozivom takve funkcije (npr. unutar neke petlje) možemo veoma brzo nesvjesno zauzeti svu raspoloživu memoriju! Dakle, svaka funkcija bi prije svog završetka morala osloboditi svu memoriju koju je dinamički zauzela (osim ukoliko vraća pokazivač na zauzeti dio memorije), i to bez obzira kako se funkcija završava: nailaskom na kraj funkcije, naredbom "**`return`**", ili bacanjem izuzetka! Naročito se često zaboravlja da funkcija, prije nego što baci izuzetak, također treba da za sobom "počisti" sve što je "zabrljala", što uključuje i oslobađanje dinamički alocirane memorije! Još je veći problem ukoliko funkcija koja dinamički alocira memoriju pozove neku drugu funkciju koja može da baci izuzetak. Posmatrajmo, na primjer, sljedeći isječak:

```
void F(int n) {
  int *pok(new int[n]);
  …
  G(n);
  …
  delete[] pok;
}
```

U ovom primjeru, funkcija "`F`" zaista briše kreirani dinamički niz po svom završetku, ali problem nastaje ukoliko funkcija "`G`" koju ova funkcija poziva baci izuzetak! Kako se taj izuzetak ne hvata u funkciji "`F`", ona će također biti prekinuta, a zauzeta memorija neće biti oslobođena. Naravno, prekid funkcije "`F`" dovodi do automatskog uništavanja automatske lokalne pokazivačke promjenljive "`pok`", ali dinamički niz na čiji početak "`pok`" pokazuje nikada se ne uništava automatski, već samo eksplicitnim pozivom operatora "**`delete[]`**". Stoga, ukoliko se operator "**`delete[]`**" ne izvrši eksplicitno, zauzeta memorija neće biti oslobođena! Ovaj problem se može riješiti na sljedeći način:

```
void F(int n) {
  int *pok(new int[n]);
  ...
  try {
    G(n);
  }
  catch(...) {
    delete[] pok;
    throw;
  }
  ...
  delete[] pok;
}
```

U ovom slučaju u funkciji "`F`" izuzetak koji eventualno baca funkcija "`G`" hvatamo samo da bismo mogli izvršiti brisanje zauzete memorije, nakon čega uhvaćeni izuzetak prosljeđujemo dalje, navođenjem naredbe "**`throw`**" bez parametara.

Iz ovog primjera vidimo da je bitno razlikovati sam dinamički niz od pokazivača koji se koristi za pristup njegovim elementima. Ovdje je potrebno ponovo napomenuti da se opisani problemi *ne bi pojavili* ukoliko bismo umjesto dinamičke alokacije memorije koristili automatsku promjenljivu tipa "`vector`" – ona bi automatski bila uništena po završetku funkcije "`F`", bez obzira da li je do njenog završetka došlo na prirodan način, ili bacanjem izuzetka iz funkcije "`G`". U suštini, kad god možemo koristiti tip "`vector`", njegovo korištenje je jednostavnije i sigurnije od korištenja dinamičke alokacije memorije. Stoga, tip "`vector`" treba koristiti kad god je to moguće – njegova upotreba *sigurno* neće nikada dovesti do curenja memorije. Jedini problem je u tome što to *nije uvijek moguće*. Na primjer, nije moguće *napraviti* tip koji se ponaša slično kao tip "`vector`" (ili sam tip "`vector`") bez upotrebe dinamičke alokacije memorije i dobrog razumijevanja kao dinamička alokacija memorije funkcionira.

Iz svega što je do sada rečeno može se zaključiti da dinamička alokacija memorije sama po sebi nije komplicirana, ali da je potrebno preduzeti dosta mjera predostrožnosti da ne dođe do curenja memorije. Dodatne probleme izaziva činjenica da i sam operator "**`new`**" može također baciti izuzetak. Razmotrimo, na primjer, sljedeći isječak:

```
try {
  int *a(new int[n1]), *b(new int[n2]), *c(new int[n3]);
  // Radi nešto sa a, b i c
  delete[] a; delete[] b; delete[] c;
}
catch(...) {
  cout << "Problemi sa memorijom!\n";
}
```

U ovom isječku vrši se alokacija tri dinamička niza, a u slučaju da alokacija ne uspije, prijavljuje se greška. Međutim, problemi mogu nastati u slučaju da, na primjer, alokacije prva dva niza uspiju, a alokacija trećeg niza ne uspije. Tada će treći poziv operatora "**`new`**" baciti izuzetak, i izvršavanje će se nastaviti unutar "**`catch`**" bloka kao što je i očekivano, ali memorija koja je zauzeta sa prve dvije uspješne alokacije ostaje zauzeta! Ovo može biti veliki problem. Jedan način da se riješi ovaj problem, mada prilično rogobatan, je da se koriste *ugniježdene* "**`try`**" – "**`catch`**" *strukture*, kao na primjer u sljedećem isječku:

```
try {
  int *a(new int[n1]);
  try {
    int *b(new int[n2]);
    try {
      int *c(new int[n3]);
      // Radi nešto sa a, b i c
      delete[] a; delete[] b; delete[] c;
    }
```

```
      catch(...) {
        delete[] b; throw;
      }
    }
    catch(...) {
      delete[] a; throw;
    }
}
catch(...) {
  cout << "Problemi sa memorijom!\n";
}
```

Najbolje je da sami analizirate tok navedenog isječka uz raličite pretpostavke, koje mogu biti: prva alokacija nije uspjela; druga alokacija nije uspjela; treća alokacija nije uspjela; sve alokacije su uspjele. Na taj način ćete najbolje shvatiti kako ovo rješenje radi.

Prikazano rješenje je zaista rogobatno, mada je prilično jasno i logično. Ipak, postoji i mnogo jednostavnije rješenje (ne računajući posve banalno rješenje koje se zasniva da umjesto dinamičke alokacije memorije koristimo tip "`vector`", kod kojeg ne moramo eksplicitno voditi računa o brisanju zauzete memorije). Ovo rješenje zasniva se na činjenici da standard jezika C++ garantira da operator "**`delete`**" *ne radi ništa* ukoliko im se kao argument proslijedi *nul-pokazivač*. To omogućava sljedeće veoma elegantno rješenje navedenog problema:

```
int *a(0), *b(0), *c(0);
try {
  a = new int[n1]; b = new int[n2]; c = new int[n3];
  // Radi nešto sa a, b i c
}
catch(...) {
  cout << "Problemi sa memorijom!\n";
}
delete[] a; delete[] b; delete[] c;
```

U ovom primjeru svi pokazivači su prvo inicijalizirani na nulu, a zatim je u "**`try`**" bloku pokušana dinamička alokacija memorije. Na kraju se na sve pokazivače primjenjuje "**`delete`**" operator, bez obzira da li je alokacija uspjela ili nije. Tako će oni nizovi koji su alocirani svakako biti uništeni, a ukoliko alokacija nekog od nizova nije uspjela, odgovarajući pokazivač će i dalje ostati *nul-pokazivač* (s obzirom da kasnija dodjela nije izvršena jer je operator "**`new`**" bacio izuzetak), tako da operator "**`delete[]`**" neće sa njim uraditi ništa, odnosno neće biti nikakvih neželjenih efekata. Usput, nula je jedina cjelobrojna vrijednost koja se smije neposredno dodijeliti promjenljivim pokazivačkog tipa (i ona tada predstavlja nul-pokazivač). Radi bolje čitljivosti, u nekim bibliotekama jezika C++ definirana je konstanta "`NULL`" koja se obično koristi za inicijalizaciju pokazivačkih promjenljivih na nul-pokazivač, ali je sasvim legalno koristiti i običnu konstantu "0" (čime izbjegavamo potrebu za uključivanjem zaglavlja biblioteka koje definiraju konstantu "`NULL`").

Bitno je napomenuti da je ponašanje operatora "**`delete`**" *nedefinirano* (i može rezultirati krahom programa) ukoliko se primijeni na pokazivač koji ne pokazuje na prostor koji je zauzet u postupku dinamičke alokacije memorije. Naročito česta greška je primijeniti operator "**`delete`**" na pokazivač koji pokazuje na prostor koji je *već obrisan* (tj. na *viseći pokazivač*). Ovo se, na primjer može desiti ukoliko dva puta uzastopno primijenimo ove operatore na *isti pokazivač* (kojem u međuvremenu između dvije primjene operatora "**`delete`**" nije dodijeljena neka druga vrijednost). Da bi se izbjegli ovi problemi, veoma dobra ideja je *eksplicitno dodijeliti nul-pokazivač* svakom pokazivaču nakon izvršenog uništavanja bloka memorije na koju on pokazuje. Na primjer, ukoliko "`pok`" pokazuje na prvi element nekog dinamičkog niza, uništavanje tog niza najbolje je izvesti sljedećom konstrukcijom:

```
delete[] pok;
pok = 0;
```

Eksplicitnom dodjelom "`pok = 0`" zapravo postižemo dva efekta. Prvo, takvom dodjelom eksplicitno naglašavamo da pokazivač "`pok`" više ne pokazuje ni na šta. Drugo, ukoliko slučajno ponovo primijenimo operator "**`delete`**" na pokazivač "`pok`", neće se desiti ništa, jer je on sada nul-pokazivač.

U praksi se često javlja potreba i za dinamičkom alokacijom *višedimenzionalnih nizova*. Mada dinamička alokacija višedimenzionalnih objekata nije direktno podržana u jeziku C++, ona se može vješto *simulirati*. Kako su višedimenzionalni nizovi zapravo *nizovi nizova*, to je za indirektni pristup njihovim elementima (pa samim tim i za njihovu dinamičku alokaciju) potrebno koristiti *složenije pokazivačke tipove*, kao što su *pokazivači na nizove*, *nizovi pokazivača* i *pokazivači na pokazivače*, zavisno od primjene. Nije prevelik problem deklarirati ovakve složene pokazivačke tipove. Na primjer, razmotrimo sljedeće deklaracije:

```
int *pok1[30];
int (*pok2)[10];
int **p3;
```

U ovom primjeru, "`pok1`" je *niz od 30 pokazivača na cijele brojeve*, "`pok2`" je *pokazivač na niz od 10 cijelih brojeva* (odnosno na *čitav niz kao cjelinu* a ne pokazivač na *prvi njegov element* – uskoro ćemo vidjeti u čemu je razlika), dok je "`pok3`" *pokazivač na pokazivač na cijele brojeve*. Obratimo pažnju između razlike u deklaraciji "`pok1`" koja predstavlja *niz pokazivača*, i deklaracije "`pok2`" koja predstavlja *pokazivač na niz*. Na izvjestan način, pojam "niz" prilikom deklaracija ima prioritet u odnosu na pojam "pokazivač", pa su u drugom primjeru zagrade bile neophodne da "izvrnu" ovaj prioritet. Ovakve konstrukcije se mogu usložnjavati unedogled (npr. principijelno je moguće napraviti *niz od 20 pokazivača na niz od 10 pokazivača na pokazivače na niz od 50 nizova od 30 realnih brojeva*), mada se potreba za tako složenim konstrukcijama javlja iznimno rijetko, i samo u veoma specifičnim primjenama. Čisto radi kurioziteta, pokažimo kako bi izgledala deklaracija ovakvog pokazivača:

```
double (**(*p[20])[10])[50][30];
```

Interesantno je navesti pravilo koje olakšava čitanje i konstrukciju složenih pokazivačkih tipova. Značenje promjenljivih najlakše čitamo tako što u deklaraciji promjenljive pođemo od njenog imena pa prvo čitamo sve što se eventualno nalazi iza imena promjeljive redom slijeva nadesno, sve do prve zatvorene male zagrade ili do kraja, a zatim čitamo sve što se eventualno nalazi ispred imena promjenljive redom zdesna nalijevo, sve do prve otvorene male zagrade ili do početka. Kada tako pročitamo sve što se nalazilo unutar malih zagrada (ako ih je bilo) nastavljamo dalje čitanje nadesno od mjesta gdje smo stali (dakle na prvoj zatvorenoj maloj zagradi) sve do sljedeće zatvorene male zagrade ili do kraja, a nakon toga nastavljamo dalje čitanje nalijevo od mjesta gdje smo stali (na otvorenoj maloj zagradi) sve do sljedeće otvorene male zagrade ili do početka. Takvo "cik cak" čitanje nastavljamo dalje sve dok ne dostignemo kraj odnosno početak deklaracije. Razumijevanje ovog pravila lako možete provjeriti na prethodnoj deklaraciji.

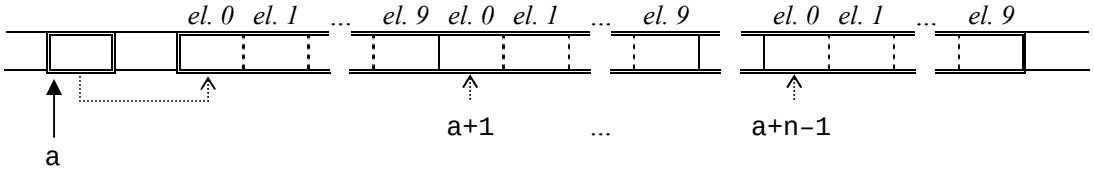
U nastavku izlaganja zanemarićemo egzistenciju ovako složenih pokazivačkih tipova i ograničićemo se samo na pokazivače na nizove, nizove pokazivača i pokazivače na pokazivače. Potreba za ovakvim konstrukcijama relativno se često javlja u izvjesnim specijalnim okolnostima. Klasični primjer situacije u kojoj se javlja potreba za složenim pokazivačkim tipovima je dinamička alokacija *višedimenzionalnih nizova*. Opći slučaj dinamičke alokacije za višedimenzionalne nizove sa većim brojem dimeznija je vrlo složen problem u koji se nećemo upuštati. Razmotrimo stoga kako bi se mogla realizirati dinamička alokacija *dvodimenzionalnih nizova*. Sjetimo se da su dvodimenzionalni nizovi zapravo *nizovi čiji su elementi nizovi*, a da se dinamička alokacija nizova čiji su elementi tipa "`Tip`" ostvaruje posredstvom pokazivača na tip "`Tip`". Slijedi da su nam za dinamičku alokaciju nizova čiji su elementi nizovi potrebni *pokazivači na nizove*. I zaista, sasvim je lako ostvariti dinamičku alokaciju dvodimenzionalnog niza kod kojeg je *druga dimenzija unaprijed poznata*, npr. matrice čiji je broj kolona unaprijed poznat (situacija u kojpk druga dimenzija nije apriori poznata je složenija, kao što ćemo uskoro vidjeti). Pretpostavimo npr. da druga dimenzija niza kojeg hoćemo da alociramo iznosi 10, a da je prva dimenzija zadana u promjenljivoj "`n`" (čija vrijednost nije unaprijed poznata). Tada ovu alokaciju možemo ostvariti pomoću sljedeće konstrukcije:

```
int (*a)[10] = new int[n][10];
```

Deklaracijom "`int (*a)[10]`" neposredno deklariramo "`a`" kao *pokazivač na niz od 10 cijelih brojeva*, dok konstrukciju "`new int[n][10]`" možemo čitati kao "potraži u memoriji prostor za `n` elemenata koji su nizovi od 10 cijelih brojeva, i vrati kao rezultat adresu pronađenog prostora (u formi odgovarajućeg pokazivačkog tipa)". Ovdje broj 10 u drugoj uglastoj zagradi iza operatora "`new`" ne predstavlja parametar operatora, već *sastavni dio tipa*, tako da on mora biti prava konstanta, a ne promjenljiva ili nekonstantni izraz. Operator "`new`" ima samo jedan parametar, mada on dopušta i drugi par uglastih zagrada, samo što se u njemu *obavezno mora nalaziti prava konstanta* (s obzirom da one čine sastavni dio tipa). Zbog toga nije moguće neposredno primjenom operatora "`new`" alocirati dvodimenzionalne dinamičke nizove čija druga dimenzija *nije unaprijed poznata*.

Razmotrimo malo detaljnije šta smo ovom konstrukcijom postigli. Ovdje smo pomoću operatora "`new`" alocirali dinamički niz od "`n`" elemenata, koji su sami za sebe tipa "niz od 10 cijelih brojeva" (što nije ništa drugo nego dvodimenzionalni niz sa "`n`" redova i 10 kolona), i usmjerili pokazivač "`a`" da pokazuje na prvi element takvog niza. Ova dodjela je legalna, jer se pokazivaču na neki tip uvijek može dodijeliti adresa dinamičkog niza čiji su elementi tog tipa. Primijetimo da je u ovom slučaju pokazivač "`a`" *pokazivač na* (*čitav*) *niz*, odnosno *pokazivač na niz kao cjelinu.* Ovo treba razlikovati od običnih pokazivača, za koje znamo da se po potrebi mogu smatrati kao *pokazivači na elemente niza*. U brojnoj literaturi se ova dva pojma često brkaju, tako da se, kada se kaže *pokazivač na niz*, obično misli na *pokazivač na prvi element niza*, a ne na pokazivač na niz kao cjelinu (inače, ova dva tipa pokazivača razlikuju se u tome kako na njih djeluje dereferenciranje i pokazivačka aritmetika). Pokazivači na (čitave) nizove koriste se prilično rijetko, i glavna im je primjena upravo za dinamičku alokaciju dvodimenzionalnih nizova čija je druga dimenzija poznata.

U gore navedenom primjeru možemo reći da pokazivač "`a`" pokazuje na prvi element *dinamičkog niza* od "`n`" elemenata čiji su elementi *obični nizovi* od 10 cjelobrojnih elemenata. Bez obzira na činjenicu da je "`a`" pokazivač, sa njim možemo baratati na iste način kao da se radi o običnom dvodimenzionalnom nizu. Tako je indeksiranje poput "`a[i][j]`" posve legalno, i interpretira se kao "`(*(a + i))[j]`". Naime, "`a`" je pokazivač, pa se izraz "`a[i]`" interpretira kao "`*(a + i)`". Međutim, kako je "`a`" pokazivač na tip "niz od 10 cijelih brojeva", to nakon njegovog dereferenciranja dobijamo element tipa "niz od 10 cijelih brojeva" na koji se može primijeniti indeksiranje. Interesantno je kako na pokazivače na (čitave) nizove djeluje pokazivačka aritmetika. Ukoliko "`a`" pokazuje na prvi element niza, razumije se da "`a + 1`" pokazuje na *sljedeći element*. Međutim, kako su element ovog niza sami za sebe nizovi, adresa na koju "`a`" pokazuje uvećava se za *čitavu dužinu nizovnog tipa* "niz od 10 cijelih brojeva". U ovome je osnovna razlika između pokazivača na (čitave) nizove i pokazivača na elemente niza. Sljedeća slika može pomoći da se shvati kako izgleda stanje memorije nakon ovakve alokacije, i šta na šta pokazuje:



U gore prikazanoj konstrukciji, za inicijalizaciju složene pokazivačke promjenljive "`a`" iskorištena je sintaksa koja koristi znak "`=`". Razumljivo je da je moguće koristiti i konstruktorsku sintaksu sa zagradama, koja bi u konkretnom primjeru izgledala ovako:

```
int (*a)[10](new int[n][10]);
```

Međutim, u ovom primjeru, upotreba konstruktorske sintakse na samom početku izlaganja vjerovatno bi zbunila početnika zbog mnoštva zagrada, tako da je ona namjerno izbjegnuta u početnom razmatranju.

Prethodno pokazana primjena pokazivača na nizove kao cjeline uglavnom je i jedina primjena takvih pokazivača. Zapravo, indirektno postoji i još jedna primjena – imena dvodimenzionalnih nizova upotrijebljena sama za sebe automatski se konvertiraju u *pokazivače na* (*čitave*) *nizove* s obzirom da su dvodimenzionalni nizovi u suštini nizovi nizova. Također, formalni parametri funkcija koji su deklarirani kao dvodimenzionalni nizovi zapravo su pokazivači na (čitave) nizove (u skladu sa općim tretmanom formalnih parametara nizovnog tipa). Ovo ujedno dodatno pojašnjava zbog čega druga dimenzija u takvim formalnim parametrima mora biti apriori poznata. Interesantno je primijetiti da se pravilo o automatskoj konverziji nizova u pokazivače ne primjenjuje rekurzivno, tako da se imena dvodimenzionalnih nizova upotrijebljena sama za sebe konvertiraju u *pokazivače na nizove*, a ne u *pokazivače na pokazivače*, kako bi neko mogao pomisliti (u pokazivače na pokazivače se konvertiraju imena *nizova pokazivača* upotrijebljena sama za sebe). Također treba primijetiti da pokazivači na nizove i pokazivači na pokazivače nisu jedno te isto (razlika je opet u djelovanju pokazivačke aritmetike).

Razmotrimo sada kako bismo mogli dinamičku alokaciju dvodimenzionalnih nizova čija druga dimenzija nije poznata unaprijed. Strogo rečeno, takva dinamička alokacija zbog nekih tehničkih razloga zapravo *nije ni moguća*, ali se može veoma uspješno *simulirati*. Za tu svrhu su nam potrebni *nizovi pokazivača*. Sami po sebi, nizovi pokazivača nisu ništa osobito: to su nizovi *čiji su elementi pokazivači*. Na primjer, sljedeća deklaracija

```
int *niz_pok[5];
```

deklarira niz "niz_pok" od 5 elemenata koji su pokazivači na cijele brojeve. Stoga, ukoliko su npr. "br_1", "br_2" i "br_3" cjelobrojne promjenljive, sve dolje navedene konstrukcije imaju smisla:

```
niz_pok[0] = &br_1;           // "niz_pok[0]" pokazuje na "br_1"
niz_pok[1] = &br_2;           // "niz_pok[1]" pokazuje na "br_2"
niz_pok[2] = new int(3);      // Alocira dinamičku promjenljivu
niz_pok[3] = new int[10];     // Alocira dinamički niz
niz_pok[4] = new int[br_1];   // Također...
*niz_pok[0] = 1;              // Djeluje poput "br_1 = 1";
br_3 = *niz_pok[1];           // Djeluje poput "br_3 = br_2"
*niz_pok[1] = 5;              // Indirektno mijenja promjenljivu "br_2"
cout << niz_pok[2];           // Ovo ispisuje adresu...
cout << *niz_pok[2];          // A ovo sadržaj dinamičke promjenljive...
delete niz_pok[2];            // Briše dinamičku promjenljivu
delete[] niz_pok[4];          // Briše dinamički niz
niz_pok[3][5] = 100;          // Vrlo interesantno...
```

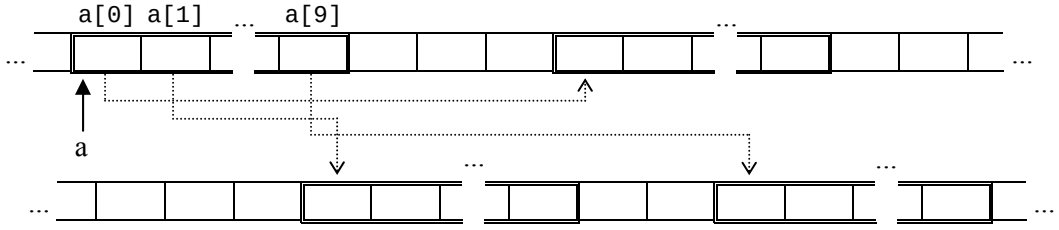
Posljednja naredba je posebno interesantna, jer pokazuje da se nizovima pokazivača može pristupati kao da se radi o *dvodimenzionalnim nizovima* (pri čemu je takav pristup smislen samo ukoliko odgovarajući element niza pokazivača pokazuje na element nekog drugog niza, koji može biti i dinamički kreiran). Zaista, svaki element niza pokazivača je pokazivač, a na pokazivače se može primijeniti indeksiranje. Stoga se izraz "niz_pok[i][j]" u suštini interpretira kao "*(niz_pok[i] + j)".

Navedeni primjer također ukazuje kako je moguće realizirati dinamički dvodimenzionalni niz čija je prva dimenzija poznata, ali čija *druga dimenzija nije poznata* (npr. matrice čiji je broj redova poznat, ali broj kolona nije). Kao što je već rečeno, jezik C++ ne podržava neposredno mogućnost kreiranja takvih nizova, ali se oni *veoma lako mogu efektivno simulirati*. Jedan od često korištenih načina za to je da se prvo formira *niz pokazivača*, a zatim se pokazivačima u tom nizu dodijele *adrese dinamički alociranih nizova koji predstavljaju redove matrice*. Na primjer, neka je potrebno realizirati dinamičku matricu sa 10 redova i "m" kolona, gdje je "m" promjenljiva. To možemo uraditi na sljedeći način:

```
int *a[10];
for(int i = 0; i < 10; i++) a[i] = new int[m];
```

U ovom slučaju elementi niza pokazivača "a" pokazuju na prve elemente svakog od redova matrice. Strogo rečeno, "a" uopće nije dvodimenzionalni niz, nego niz pokazivača koji pokazuju na početke neovisno alociranih dinamičkih nizova, od kojih svaki predstavlja po jedan red matrice, i od kojih svaki

može biti lociran u potpuno različitim dijelovima memorije! Međutim, sve dok "elementima" ovakvog "dvodimenzionalnog niza" možemo pristupati pomoću izraza "`a[i][j]`" (a možemo, zahvaljujući pokazivačkoj aritmetici) ne trebamo se mnogo brinuti o stvarnoj organizaciji u memoriji. Zaista, "`a`" možemo koristiti kao dvodimenzionalni niz sa 10 redova i "`m`" kolona, iako se radi samo o veoma vještoj simulaciji. Sa ovako simuliranim dvodimenzionalnim nizovima možemo raditi na gotovo isti način kao i sa pravim dvodimenzionalnim nizovima. Jedine probleme mogli bi eventualno izazvati "prljavi" trikovi sa pokazivačkom aritmetikom koji bi se oslanjali na pretpostavku da su redovi dvodimenzionalnih nizova smješteni u memoriji striktno jedan za drugim (što ovdje nije slučaj). Pravo stanje u memoriji kod ovako simuliranih dvodimenzionalnih nizova moglo bi se opisati recimo sljedećom slikom:



Pažljivijem posmatraču sigurno neće promaći upadljiva sličnost između opisane tehnike za dinamičku alokaciju dvodimenzionalnih nizova i kreiranja nizova ili vektora čiji su elementi vektori. Kasnije ćemo vidjeti da u oba slučaja u pozadini zapravo stoji isti mehanizam.

Kada smo govorili o vektorima čiji su elementi vektori, govorili smo o mogućnosti kreiranja struktura podataka nazvanih *grbave matrice*, koji predstavljaju strukture koje po načinu upotrebe liče na obične matrice, ali kod kojih različiti "redovi" imaju različit broj elemenata. Primijetimo da nam upravo opisana simulacija dvodimenzionalnih nizova preko nizova pokazivača također omogućava veoma jednostavno kreiranje grbavih nizova. Na primjer, ukoliko izvršimo sekvencu naredbi

```
int *a[10];
for(int i = 0; i < 10; i++) a[i] = new int[i + 1];
```

tada će se niz pokazivača "`a`" sa aspekta upotrebe ponašati kao dvodimenzionalni niz u kojem prvi red ima jedan element, drugi element dva reda, treći element tri reda, itd.

Važno je primijetiti da je u svim prethodnim primjerima zasnovanim na nizovima pokazivača, "dvodimenzionalni niz" zapravo sastavljen od gomile *posve neovisnih* dinamičkih jednodimenzionalnih nizova, koje međusobno objedinjuje niz pokazivača koji pokazuju na početke svakog od njih. Stoga, ukoliko želimo uništiti ovakve "dvodimenzionalne nizove" (tj. osloboditi memorijski prostor koji oni zauzimaju), moramo posebno uništiti svaki od jednodimenzionalnih nizova koji ih tvore. To možemo učiniti npr. na sljedeći način:

```
for(int i = 0; i < 10; i++) delete[] a[i];
```

Razmotrimo kako sada simulirati dvodimenzionalni niz u kojem niti jedna dimenzija nije unaprijed poznata. Rješenje se samo nameće: potrebno je i *niz pokazivača* (koji pokazuju na početke svakog od redova) *realizirati kao dinamički niz*. Pretpostavimo, na primjer, da želimo simulirati dvodimenzionalni niz sa "`n`" redova i "`m`" kolona, pri čemu niti "`m`" niti "`n`" nemaju unaprijed poznate vrijednosti. To možemo uraditi recimo ovako:

```
int **a(new int*[n]);
for(int i = 0; i < n; i++) a[i] = new int[m];
```

Obratite pažnju na zvjezdicu koja govori da alociramo niz *pokazivača* na cijele brojeve a ne niz cijelih brojeva. Nakon ovoga, "`a`" možemo koristiti kako da se radi o dvodimenzionalnom nizu i pisati "`a[i][j]`", mada je "`a`" zapravo *pokazivač na pokazivač* (*dvojni pokazivač*)! Zaista, kako je "`a`" pokazivač, na njega se može primijeniti indeksiranje, tako da se izraz "`a[i]` " interpretira kao "`*(a + i)`". Međutim, nakon dereferenciranja dobijamo ponovo pokazivač (jer je "`a`" pokazivač *na*

pokazivač), na koji se ponovo može primijeniti indeksiranje, tako da se na kraju izraz "`a[i][j]`" zapravo interpretira kao "`*(*(a + i) + j)`". Bez obzira na interpretaciju, za nas je važna činjenica da se "`a`" gotovo svuda može koristiti na način kako smo navikli raditi sa običnim dvodimenzionalnim nizovima). Naravno, ovakav "dvodimenzionalni niz" morali bismo brisati postupno (prvo sve redove, a zatim niz pokazivača koji ukazuju na početke redova):

```
for(int i = 0; i < n; i++) delete[] a[i];
delete[] a;
```

Vidjeli smo da se kod opisanog postupka dinamičke alokacije dvodimenzionalnih nizova čija druga dimenzija nije poznata svaki od redova tako kreiranog niza alocira posebno, što za posljedicu ima da individualni redovi tako kreiranog niza nisu nužno kontinualno raspoređeni u memoriji, odnosno mogu biti razbacani svuda po memoriji. Stoga, opisani postupak dinamičke alokacije nazivamo *fragmentirana alokacija*. Nekontinualni razmještaj individualnih redova rijetko predstavlja problem u praksi. Međutim, moguće je izvršiti i takvu dinamičku alokaciju dvodimenzionalnih nizova koja garantira da će se njihovi redovi nalaziti kontinuirano u memoriji, redom jedan za drugim. U tom slučaju govorimo o *kontinualnoj alokaciji*. Da bismo izvršili kontinualnu alokaciju dinamičkog dvodimenzionalnog niza sa "`n`" redova i "`m`" kolona, umjesto da individualno kreiramo "`n`" dinamičkih nizova od kojih svaki ima "`m`" elemenata (što daje ukupno "`n * m`" elemenata), kreiraćemo *jedan* dinamički niz od "`n * m`" elemenata, koji predstavlja prostor koji na jednom mjestu čuva sve elemente dvodimenzionalnog niza, red po red. Nakon toga ćemo prvom pokazivaču u nizu pokazivača dodijeliti adresu prvog elementa alociranog prostora, a svakom sljedećem pokazivaču adresu koja se nalazi "`m`" elemenata ispred adrese na koju pokazuje prethodni pokazivač. Na taj način, svaki pokazivač u nizu pokazuje na prostor koji je dovoljan za smještanje tačno "`m`" elemenata jednog reda, a da pri tome ne dođe do konflikta sa sadržajem ostalih redova elemenata. Konkretno, to bi se moglo izvesti recimo ovako:

```
int **a(new int*[n]);
a[0] = new int[n * m];
for(int i = 1; i < n; i++) a[i] = a[i – 1] + m;
```

Jasno je da je na sličan način moguće izvršiti i kontinualnu alokaciju grbavih nizova. Ovaj način alokacije ima izvjesne prednosti nad fragmentiranom alokacijom. Na prvom mjestu, kontinualna alokacija omogućava legalne trikove sa upotrebom pokazivačke aritmetike koji se zasnivaju na kontinualnom razmještaju redova u memoriji. Pored toga, kontinualno alocirane dvodimenzionalne dinamičke nizove je jednostavnije uništiti nego u slučaju fragmentirane alokacije. Zaista, da bismo uništili kontinualno alocirani dvodimenzionalni dinamički niz, dovoljno je izvršiti

```
delete[] a[0]; delete[] a;
```

S druge strane, osnovni nedostatak kontinualne alokacije leži u činjenici da je slobodni prostor u memoriji veoma često također fragmentiran, tako da postoji mogućnost da se ne može pronaći prostor za smještanje "`n * m`" elemenata, iako se može naći "`n`" prostora (razbacanih po memoriji) dovoljnih za smještanje "`m`" elemenata. Drugim riječima, ukoliko su "`n`" i "`m`" veliki, kontinualna alokacija ima znatno više šansi za neuspjeh u odnosu na fragmentiranu alokaciju.

Interesantno je uporediti obične (statičke) dvodimenzionalne nizove, dinamičke dvodimenzionalne nizove sa poznatom drugom dimenzijom, simulirane dinamičke dvodimenzionalne nizove sa poznatom prvom dimenzijom, i simulirane dinamičke dvodimenzionalne nizove sa obje dimenzije nepoznate. U sva četiri slučaja za pristup individualnim elementima niza možemo koristiti sintaksu "`a[i][j]`", s tim što se ona u drugom slučaju logički interpretira kao "`(*(a + i))[j]`", u trećem slučaju kao "`*(a[i] + j)`", a u četvrtom slučaju kao "`*(*(a + i) + j)`". Međutim, uzmemo li u obzir da se ime niza upotrijebljeno bez indeksiranja automatski konvertira u pokazivač na prvi element tog niza, kao i činjenicu da su istinski elementi dvodimenzionalnih nizova zapravo jednodimenzionalni nizovi, doći ćemo do veoma interesantnog zaključka da su u sva četiri slučaja sve četiri sintakse ("`a[i][j]`", "`(*(a + i))[j]`", "`*(a[i] + j)`" i "`*(*(a + i) + j)`") u potpunosti ekvivalentne, i svaka od njih se može koristiti u sva četiri slučaja! Naravno da se u praksi gotovo uvijek koristi sintaksa "`a[i][j]`", ali je za razumijevanje suštine korisno znati najprirodniju interpretaciju za svaki od navedenih slučajeva.

Pokazivači na pokazivače početnicima djeluju kompliciranо zbog dvostruke indirekcije, ali oni nisu ništa kompliciraniji od klasičnih pokazivača, ako se ispravno shvati njihova suština. U jeziku C++ pokazivači na pokazivače pretežno se koriste za potrebe dinamičke alokacije dvodimenzionalnih nizova, dok se u jeziku C dvojni pokazivači intenzivno koriste i u nekim situacijama u kojima je u jeziku C++ prirodnije upotrijebiti *reference na pokazivač* (tj. reference vezane za neki pokazivač). Na primjer, pomoću deklaracije

```
int *&ref(pok);
```

deklariramo referencu "`ref`" vezanu na pokazivačku promjenljivu "`pok`" (tako da je "`ref`" zapravo referenca na pokazivač). Sada se "`ref`" ponaša kao *alternativno ime* za pokazivačku promjenljivu "`pok`". Obratite pažnju na redoslijed znakova "`*`" i "`&`". Ukoliko bismo zamijenili redoslijed ovih znakova, umjesto reference na pokazivač pokušali bismo deklarirati *pokazivač na referencu*, što nije dozvoljeno u jeziku C++ (postojanje pokazivača na referencu omogućilo bi pristup internoj strukturi reference, a dobro nam je poznato da tvorci jezika C++ nisu željeli da ni na kakav način podrže takvu mogućnost). Inače, u skladu sa ranije navedenim pravilom o čitanju složenijih deklaracija, kada čitanje vršimo od imena promjenljive *zdesna nalijevo*, jasno vidimo da "`ref`" zaista predstavlja *referencu na pokazivač na cijele brojeve*.

Reference na pokazivače najčešće se koriste ukoliko je potrebno neki pokazivač prenijeti po referenci kao parametar u funkciju, što je potrebno npr. ukoliko funkcija treba da izmijeni sadržaj samog pokazivača (a ne objekta na koji pokazivač pokazuje). Na primjer, sljedeća funkcija vrši razmjenu dva pokazivača koji su joj proslijeđeni kao stvarni parametri (zanemarimo ovom prilikom činjenicu da će generička funkcija "`Razmijeni`" koju smo razmatrali na prethodnim predavanjima sasvim lijepo razmijeniti i dva pokazivača, pri čemu će se u postupku dedukcije tipa izvesti zaključak da nepoznati tip predstavlja tip pokazivača na realne brojeve):

```
void RazmijeniPokazivace(double *&x, double *&y) {
  double *pomocna(x);
  x = y; y = pomocna;
}
```

Kako u jeziku C ne postoje reference, sličan efekat se može postići jedino upotrebom *dvojnih pokazivača*, kao u sljedećoj funkciji

```
void RazmijeniPokazivace(double **p, double **q) {
  double *pomocna(*p);
  *p = *q; *q = pomocna;
}
```

Naravno, prilikom poziva ovakve funkcije "`RazmijeniPokazivace`", kao stvarne argumente morali bismo navesti *adrese* pokazivača koje želimo razmijeniti (a ne same pokazivače), pri čemu nakon uzimanja adrese dobijamo dvojni pokazivač. Drugim riječima, za razmjenu dva pokazivača "`p1`" i "`p2`" (na tip "**`double`**") morali bismo izvršiti sljedeći poziv:

```
RazmijeniPokazivace(&p1, &p2);
```

Očigledno, upotreba referenci na pokazivače (kao uostalom i upotreba bilo kakvih referenci) oslobađa korisnika funkcije potrebe da eksplicitno razmišlja o adresama.

Svi do sada prikazani postupci dinamičke alokacije matrica nisu vodili računa o tome da li su alokacije zaista uspjele ili nisu. U realnim situacijama moramo i o tome voditi računa. Naročito treba paziti da u slučaju da alokacija ne uspije do kraja, prije nego što nastavimo dalje išta raditi neophodno je ukloniti ono što je u postupku alokacije bilo alocirano (u suprotnom, alocirani prostor neće niko osloboditi, tako da će doći do curenja memorije). Postoji više načina da to uradimo. Recimo, jedan od načina da korektno obavimo fragmentiranu dinamičku alokaciju matrice realnih brojeva sa "`n`" redova i "`m`" kolona može izgledati recimo ovako:

```
try {
  double **a(new double*[n]);
  for(int i = 0; i < n; i++) a[i] = 0;
  try {
    for(int i = 0; i < n; i++) a[i] = new double[m];
  }
  catch(...) {
    for(int i = 0; i < n; i++) delete[] a[i];
    delete[] a;
    throw;
  }
}
catch(...) {
  cout << "Problemi sa memorijom!\n";
}
```

U ovom primjeru, prije same dinamičke alokacije redova matrice, svim pokazivačima na redove matrice su prvo dodijeljeni nul-pokazivači. Time smo postigli da u slučaju da neke od alokacija ne uspiju, u "**catch**" bloku ne moramo voditi računa koje su alokacije redova uspjele a koje nisu, nego prosto primjenjujemo operator "**delete**" na sve pokazivače, znajući da on ne radi ništa ukoliko mu se kao parametar ponudi nul-pokazivač. Ovo je trik koji se često koristi kod dinamičke alokacije složenih i fragmentiranih struktura podataka. Drugi način za rješavanje problema neuspješnih alokacija je da brojimo koliko je bilo uspješnih alokacija, i da u slučaju neuspjeha primijenimo operator "**delete**" samo na one pokazivače koji pokazuju na uspješno alocirane blokove. Mada je ovaj način neznatno efikasniji, njega je teže generalizirati na slučajeve još složenijih alokacija od ovdje opisanih. Za slučaj klasične fragmentirane dinamičke alokacije dvodimenzionalnih nizova, ovaj način bi se mogao implementirati recimo na sljedeći način:

```
try {
  double **a(new double*[n]);
  int brojac(0);
  try {
    while(brojac < n) a[brojac++] = new double[m];
  }
  catch(...) {
    for(int i = 0; i < brojac; i++) delete[] a[i];
    delete[] a;
    throw;
  }
}
catch(...) {
  cout << "Problemi sa memorijom!\n";
}
```

Ukoliko umjesto fragmentirane koristimo kontinualnu alokaciju, situacija je jednostavnija zbog znatno manjeg broja alociranih blokova. Tako se kontinualna dinamička alokacija matrice sa "n" redova i "m" kolona uz vođenje računa o uspješnosti alokacija može izvesti recimo ovako:

```
try {
  double **a(new double*[n]);
  try {
    a[0] = new int[n * m];
    for(int i = 1; i < n; i++) a[i] = a[i – 1] + m;
  }
  catch(...) {
    delete[] a[0]; delete[] a;
    throw;
  }
}
catch(...) {
  cout << "Problemi sa memorijom!\n";
}
```

Vidimo da dinamička alokacija matrica proizvoljnih dimenzija može biti mukotrpna ukoliko vodimo računa o mogućim memorijskim problemima. Naročito je mukotrpno identičan postupak ponavljati za svaku od matrica koju želimo da kreiramo. Zbog toga se kao prirodno rješenje nameće *pisanje funkcija koje obavljaju dinamičko stvaranje odnosno uništavanje matrica proizvoljnih dimenzija*, koje na sebe preuzimaju opisani postupak (naravno, još jednostavnije rješenje je koristiti vektore vektora, ali ovdje želimo da naučimo mehanizme nižeg nivoa koji leže u pozadini rada takvih tipova podataka). Sljedeći primjer pokazuje kako bi takve funkcije mogle izgledati za slučaj fragmentirane alokacije (funkcije su napisane kao generičke funkcije, tako da omogućavaju stvaranje odnosno uništavanje matrica čiji su elementi proizvoljnog tipa):

```
template <typename TipElemenata>
void UnistiMatricu(TipElemenata **mat, int broj_redova) {
  if(mat == 0) return;
  for(int i = 0; i < broj_redova; i++) delete[] mat[i];
  delete[] mat;
}
template <typename TipElemenata>
TipElemenata **StvoriMatricu(int broj_redova, int broj_kolona) {
  TipElemenata **mat(new TipElemenata*[broj_redova]);
  for(int i = 0; i < broj_redova; i++) mat[i] = 0;
  try {
    for(int i = 0; i < broj_redova; i++)
      mat[i] = new TipElemenata[broj_kolona];
  }
  catch(...) {
    UnistiMatricu(mat, broj_redova);
    throw;
  }
  return mat;
}
```

Naravno, trivijalno je prepraviti ove funkcije da umjesto fragmentirane koriste kontinualnu alokaciju. Međutim, u ovim funkcijama potrebno je obratiti pažnju na nekoliko detalja. Prvo, funkcija "`UnistiMatricu`" je napisana ispred funkcije "`StvoriMatricu`", s obzirom da se ona poziva iz funkcije "`StvoriMatricu`". Također, funkcija "`UnistiMatricu`" napisana je tako da ne radi ništa u slučaju da joj se kao parametar prenese nul-pokazivač, čime je ostvarena konzistencija sa načinom na koji se ponaša operator "**`delete`**". Ova konzistencija će nam kasnije biti od velike koristi. Konačno, funkcija "`StvoriMatricu`" vodi računa da iza sebe "počisti" sve dinamički alocirane nizove prije nego što eventualno baci izuzetak u slučaju da dinamička alokacija nije uspjela do kraja (ovo "čišćenje" je realizirano pozivom funkcije "`UnistiMatricu`").

Sljedeći primjer ilustrira kako možemo koristiti napisane funkcije za dinamičko kreiranje i uništavanje matrica:

```
int n, m;
cin >> n >> m;
double **a(0), **b(0);
try {
  a = StvoriMatricu<double>(n, m);
  b = StvoriMatricu<double>(n, m);
  // Radi nešto sa matricama a i b
}
catch(...) {
  cout << "Problemi sa memorijom!\n";
}
UnistiMatricu(a, n);
UnistiMatricu(b, n);
```

Primijetimo da smo dvojne pokazivače "`a`" i "`b`" koje koristimo za pristup dinamički alociranim matricama prvo inicijalizirali na nulu, prije nego što zaista pokušamo dinamičku alokaciju pozivom

funkcije "`StvoriMatricu`". Na taj način garantiramo da će u slučaju da alokacija ne uspije, odgovarajući pokazivač biti nul-pokazivač, tako da kasniji poziv funkcije "`UnistiMatricu`" neće dovesti do problema u slučaju da kreiranje matrice uopće nije izvršeno (ovdje imamo situaciju identičnu situaciji o kojoj smo već govorili prilikom razmatranja dinamičke alokacije običnih nizova). Generalno, kad god dinamički alociramo *više od jednog objekta*, trebamo izbjegavati obavljanje dinamičke alokacije odmah prilikom inicijalizacije pokazivača, jer u suprotnom prilikom brisanja objekata možemo imati problema u slučaju da nije uspjela alokacija svih objekata (naime, ne možemo znati koji su objekti alocirani, a koji nisu). Primijetimo još da u funkciju "`UnistiMatricu`" moramo kao parametar prenositi i broj redova matrice, s obzirom da nam je on potreban u "**`for`**" petlji, a njega nije moguće saznati iz samog pokazivača.

Prilikom poziva generičke funkcije "`StvoriMatricu`" morali smo unutar šiljastih zagrada "`<>`" eksplicitno specificirati tip elemenata matrice, s obzirom da se tip elemenata ne može odrediti dedukcijom tipa iz parametara funkcije. Ovaj problem možemo izbjeći ukoliko modificiramo funkciju "`StvoriMatricu`" tako da kao jedan od parametara prihvata dvojni pokazivač za pristup elementima matrice. Tako modificirana funkcija treba da u odgovarajući parametar *smjesti* adresu dinamički alocirane matrice (umjesto da tu adresu *vrati kao rezultat*). Naravno, u tom slučaju ćemo za dinamičko alociranje matrice umjesto poziva poput

```
a = StvoriMatricu<double>(n, m);
```

koristiti poziv poput

```
StvoriMatricu(a, n, m);
```

Naravno, prvi parametar se mora prenositi po referenci da bi uopće mogao biti promijenjen, što znači da odgovarajući formalni parametar mora biti *referenca na dvojni pokazivač* (ne plašite se što ovdje imamo *trostruku indirekciju*). Ovakvu modifikaciju možete sami uraditi kao korisnu vježbu. Treba li uopće napominjati da se u jeziku C (koji ne posjeduje reference) sličan efekat može ostvariti jedino upotrebom *trostrukih pokazivača* (odnosno *pokazivača na pokazivače na pokazivače*)!?

Dinamički kreirane matrice (npr. matrice kreirane pozivom funkcije "`StvoriMatricu`") u većini konteksta možemo koristiti kao i obične dvodimenzionalne nizove. Moguće ih je i prenositi u funkcije, samo što odgovarajući formalni parametar koji služi za pristup dinamičoj matrici mora biti definiran kao dvojni pokazivač (kao u funkciji "`UnistiMatricu`"). Ipak, razlike između dinamički kreiranih matrica kod kojih nijedna dimenzija nije poznata unaprijed i običnih dvodimenzionalnih nizova izraženije su u odnosu na razlike između običnih i dinamičkih jednodimenzionalnih nizova. Ovo odražava činjenicu da u jeziku C++ zapravo ne postoje pravi dvodimenzionalni dinamički nizovi kod kojih druga dimenzija nije poznata unaprijed. Oni se mogu veoma vjerno simulirati, ali ne u potpunosti savršeno. Tako, na primjer, nije moguće napraviti jedinstvenu funkciju koja bi prihvatala kako statičke tako i dinamičke matrice, jer se ime dvodimenzionalnog niza upotrijebljeno samo za sebe ne konvertira u dvojni pokazivač, nego u pokazivač na niz, o čemu smo već govorili (naravno, u slučaju potrebe, moguće je napraviti dvije funkcije sa *identičnim tijelima a različitim zaglavljima*, od kojih jedna prima statičke a druga dinamičke matrice).