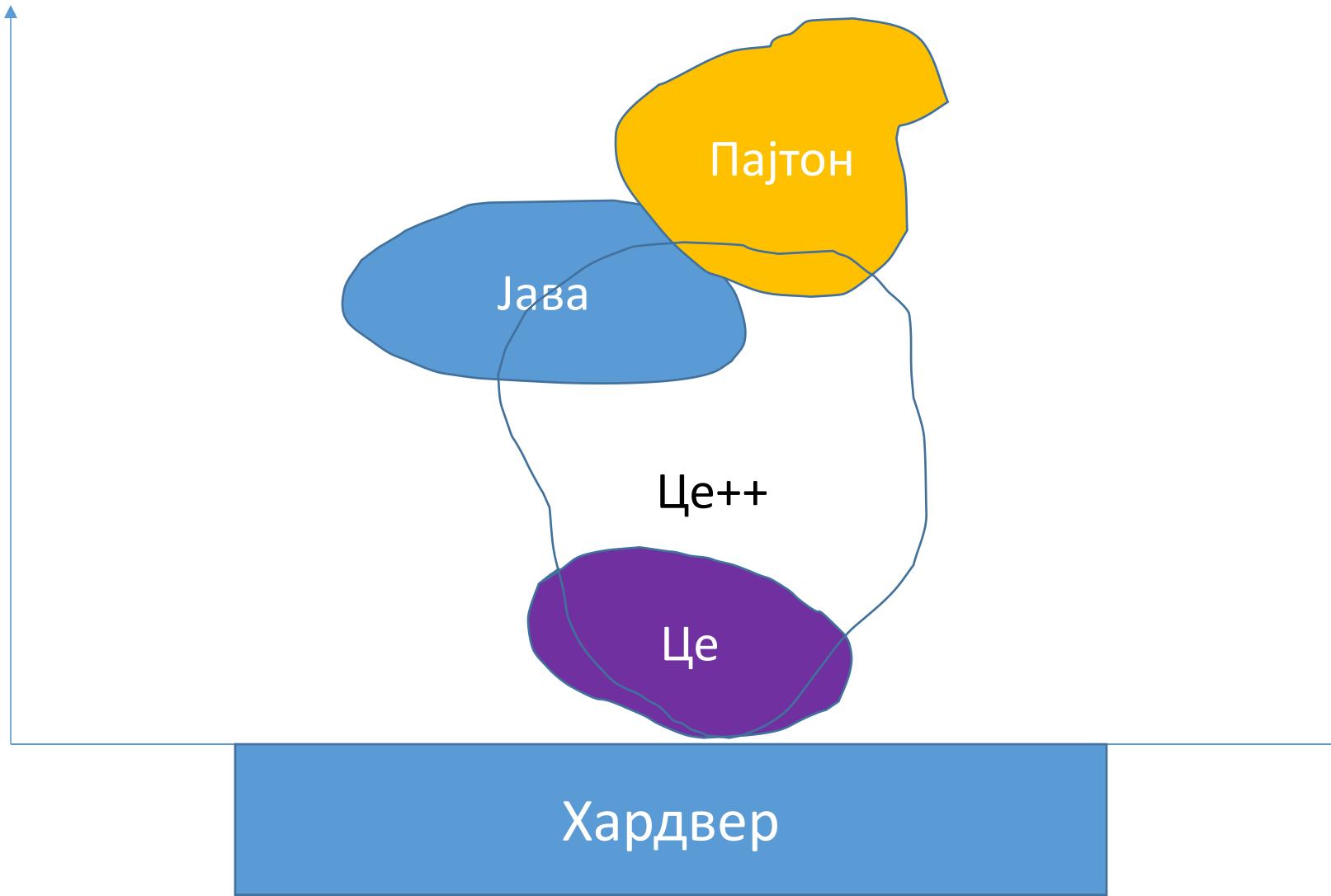


# Показивачи, низови и вектор

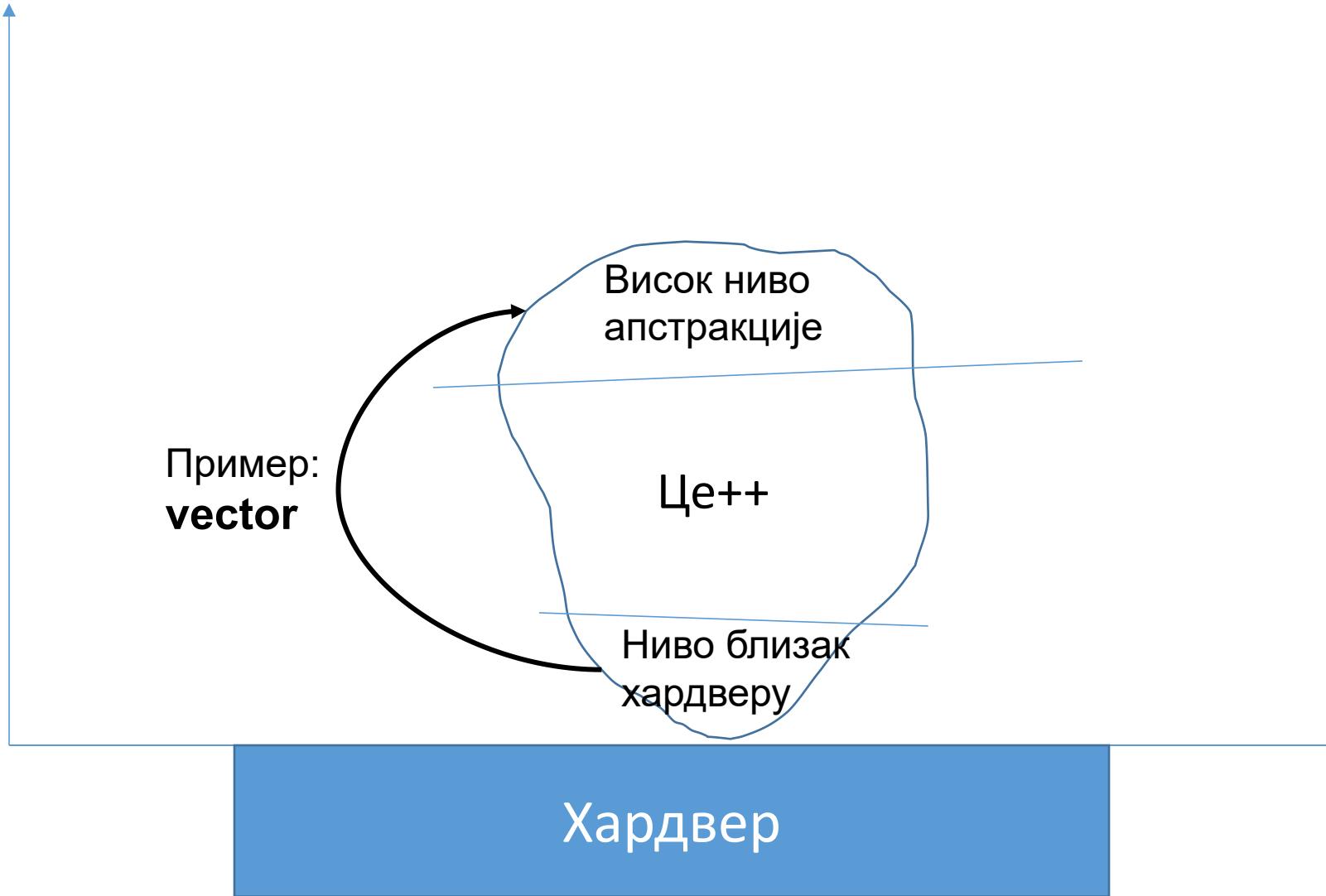
## Пожељна својства:

- Операције високог нивоа
- Типизирано
- Величина може да се мења
- Може се обавити провера опсега током извршавања програма
- Брзо

# Релативни ниво апстракције Це++-а



# Релативни ниво апстракције Це++-а



## Највиши ниво – Це++

## Међуниво – Це++

## Најнижи ниво - Асм

```
int g, a, b;
g = a + b;
```

```
mov eax, dword ptr [a]
add eax, dword ptr [b]
mov dword ptr [g], eax
```

```
MyCpxx g, a, b;
g = a + b;
```

```
struct MyCpxx
{
    int r;
    int i;
};

MyCpxx operator+
    const MyCpxx& x,
    const MyCpxx& y)
{
    return MyCpxx
    { x.r + y.r,
      x.i + y.i };
}
```

```
mov edx, dword ptr [b (0D5337Ch) ]
mov eax, dword ptr ds:[00D53380h]
add edx, dword ptr [a (0D53384h) ]
add eax, dword ptr ds:[0D53388h]
mov dword ptr [g (0D5338Ch)], edx
mov dword ptr ds:[00D53390h], eax
```

# Вектор

- Вектор је тип података која може да садржи више елемената другог типа података. Такви типови се зову контејнери.
- Вектор је најкориснији (најупотребљаванији) контејнер
  - Једноставан је
  - Компактно складишти елементе
  - Приступ елементима је брз
  - Може се проширити да садржи произвољан број елемената
  - Омогућава и проверу индекса при приступу
- Вектор је подразумевани контејнер
  - Ако вам треба контејнер – користите вектор... осим у случајевима када имате јасан разлог због ког бисте користили неки други контејнер.

# Изградња вектора од доле

- На нивоу хардвера, барата се појмовима као што су меморија и адресе.
  - Ниског нивоа
  - Нетипизирано
  - Парчад меморије фиксне величине
  - Никакве провере
  - Али зато најбрже могуће
- На вишем нивоу апстракције треба нам нешто као вектор.
  - Операције високог нивоа
  - Типизирано
  - Величина може да се мења
  - Провера опсега током извршавања програма
  - И даље врло (довољно) брзо

# Изградња вектора од доле

- На ниским нивоима, близу хардвера, живот је једноставан... и бруталан.
  - Све морате програмирати сами.
  - Нема провере типова да вам помогне.
  - Грешке током извршавања се откривају тако што добијете неисправне резултате или се програм „слеши“.
- Тежимо да се дигнемо у више нивое апстракције што пре можемо. На ниске нивое скачемо само када морамо.
  - То повећава продуктивност и поузданост
  - Људима је читљивије, тј. разумљивије
- У наредних неколико предавања показаћемо како се то ради.
  - У супротном, морали бисмо да верујемо у магију.
  - Технике које се користе у изградњи вектора су управо технике које су основа за сав рад са структурама података на високом нивоу апстракције.

# Животни век (трајност) променљиве (објекта)

- Сваком објекту требају ресурси.
- Ресурси морају бити придружен објекту докле год је он жив, тј. током његовог животног века.
- Три питања: а) колико ресурса?, б) када их треба заузети? (када почиње животни век променљиве), в) када се могу ослободити? (када се завршава животни век).
- Одговор на а) може бити сложен у општем случају, али је суштински одређен типом променљиве.\*
- Ограниччење за б) је: не касније од места прве употребе променљиве, а не раније од почетка програма
- Ограниччење за в) је: не раније од места последње употребе променљиве, а не касније од kraja програма

\*Типове можемо сврстати у две групе: типови код којих се потреба за ресурсима мења током животног века и они код којих је непроменљива. Ради једноставности, претпоставимо другу групу, јер нам је то сасвимово за разумевање одговора на друга два питања, а то нам је сада приоритет.

# Животни век (трајност) променљиве

- Један конкретан одговор:
  - б) Живот променљиве почиње одмах на почетку програма
  - в) Завршава се на самом крају програма
- Дакле, ресурси променљивој морају бити придржени током трајања целог програма.
- Овако се третирају све (а) глобалне променљиве, (б) локалне декларисане као **static** и (в) променљиве чланови класе (атрибути) декларисане као **static**. (Овакве променљиве се називају „**променљиве статичке трајности**“).

```
int x;  
  
void foo() {  
    ...  
    static int y;  
    ...  
}  
  
class Klasa {  
    ...  
    static int z;  
    ...  
};
```

Често ћете чути да се каже „глобалне променљиве“, а заправо се мисли на променљиве статичке трајности. То је непрецизно, али разумљиво, јер највећи број променљивих статичке трајности јесу управо глобалне променљиве.

# Животни век (трајност) променљиве

- Други конкретан одговор: нека компајлер сам (автоматски) одреди када почиње животни век и када се завршава.
- Одговорност се пребацује на компајлер, а он то одређује током превођења. Такође се труди да то све оптимизује.
- Овако се третирају све локалне променљиве („променљиве аутоматске трајности“).
- Међутим, компајлер се ослања на једну **претпоставку: Животни век променљиве сигурно није шире од њеног досега.**

```
void foo() {      void foo() {  
    ...  
    int x;  
    ...  
    x = 5;  
    ...  
    y = x + 3;  
    ...  
}
```

{  
 ...  
 int x;  
 ... // нека употреба x  
 }  
 ... // претпоставка да је x мртво

```
void foo() {  
    int* p;  
    {  
        int x;  
        ...  
        p = &x;  
    }  
    y = *p; // ???  
}
```

# Заузимање и ослобађање ресурса

- Када размишљамо о заузимању и ослобађању ресурса при дружених променљивама статичке и аутоматске трајности, не треба да замишљамо неки сложен процес који се обавља током извршавања. Код већине типова, све је припремљено у напред, током превођења.
- За сваку променљиву се зна (компајлер одлучи) које ресурсе користи и онда за сваки приступ тој променљивој генеришу се одговарајуће инструкције који раде управо са тим ресурсима (а не са неким другим).
- Да би се ово могло обавити током превођења, компајлер мора знати колико је ресурса потребно.

```
int x;  
...  
x = 5;           r3 <- 5           mem[_x] <- 5           mem[_x] <- 5  
...  
y = x + 3;       r4 <- r3 + 3     r4 <- mem[_x] + 3     r2 <- mem[_x]  
...           ...           ...           r4 <- r2 + 3  
...  
...
```

- Важно је да се ресурси неке променљиве не користе у друге сврхе докле год је она жива, и компајлер се стара о томе.

```
...  
r3 <- 5  
...  
r3 <- 3  
...  
r4 <- r3 + 3  
...
```

# Шта нам Це++ нуди на најнижем нивоу?

- Низови елемената

```
char ac[7];
int ai[100];

int f(int n)
{
    char lc[20];
    int li[60];
    double lx[n]; //грешка: величина низа мора бити позната током превођења

    // ...
}
```

# Својства низа – непроменљиве је величине

```
ifstream if;
int array[?];

{
    int i = 0;
    while (if)
    {
        int t;
        if >> t;
        array[i] = t; // овако се приступа појединачном елементу
        i += 1;
    }
}
```

# Својства низа – не подржава доделу и поређење

```
{  
    int a[300];  
    int b[300];  
  
    // ...  
  
    if (a == b) ... // грешка у превођењу  
  
    a = b; // грешка у превођењу  
  
    // не подржава ни преношење по вредности  
}
```

# Својства низа – не зна своју величину

```
void f(int pi[], int n, char pc[])
// исто што и void f(int* pi, int n, char* pc)
// упозорење: врло опасан код
{
    int buf2[300];      // не можемо написати int buf2[n]
    if (300 < n) error("not enough space");
    for (int i = 0; i < n; ++i)
        buf2[i] = pi[i]; // да ли pi има довољну величину?
}
```

# Иницијализација низа

```
int ai[] = { 1, 2, 3, 4, 5, 6 };      // низ од 6 int-ова
```

```
int ai2[100] = { 0,1,2,3,4,5,6,7,8,9 };
// осталите 90 елемената се поставља на 0
```

```
double ad3[100] = { }; // сви елементи се стављају на 0.0
```

```
int* ad4[100] = { }; // сви елементи се стављају на nullptr
```

# Да ли низ задовољава наше потребе?

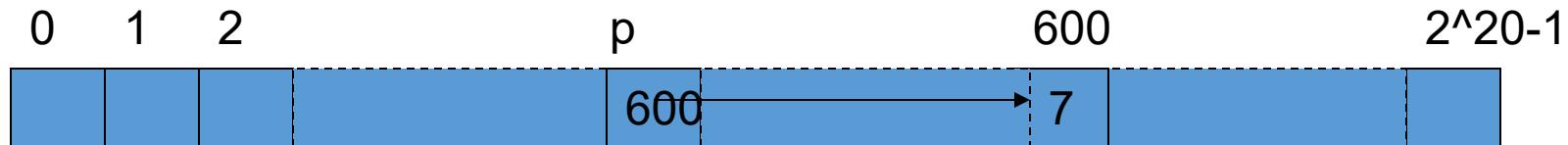
- Операције високог нивоа
- Типизирано
- Величина може да се мења
- Може се обавити провера опсега током извршавања програма
- И даље врло брзо

# Животни век (трајност) променљиве

- Трећи одговор: програмер сам одређује када ће променљива постати жива и када ће умрети.
- За такве променљиве се каже и да су **алоциране трајности**, због тога што се ресурси (сада је то искључиво меморија) за те променљиве алоцирају. (Убрзо ћемо видети да алокација, тј. заузимање меморије у овом случају није нешто што се обавља током превођења, већ током извршавања.)
- Заправо, назив „променљива“ и нећемо баш употребљавати за оно што ручно алоцирамо, већ ћемо то звати објектом. Објекат је, да се подсетимо, парче меморије у којем може бити вредност одређеног типа, а променљива је објекат са именом.
- Па ако нема име, како му се обраћамо?

# Показивачи

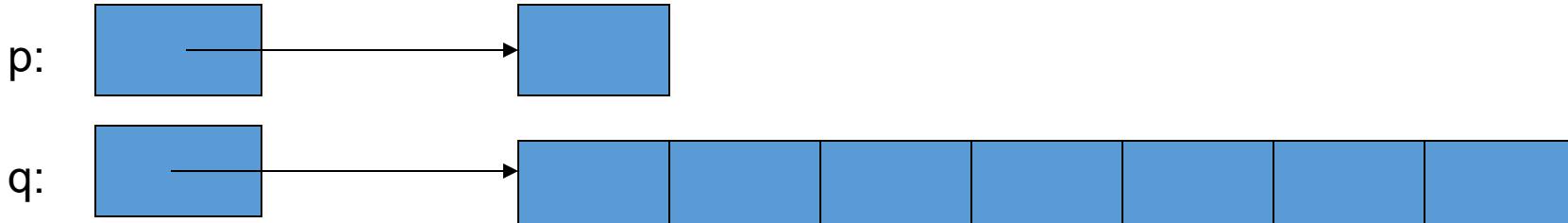
- Показивач је објекат показивачког типа
- Вредности показивачког типа су адресе у меморији
  - У питању су целобројне вредности
  - Нпр. прва адреса у меморији је 0, следећа 1, итд.
  - Неки показивач **p** може садржати адресу неке мем. локације



- Показивач показује на објекат одређеног типа
  - Нпр. **double\*** показује на **double**, не на, рецимо, **string**
  - Тип показивача одређује како се меморија (објекат) на коју показује може користити
    - На исти начин на који тип променљиве одређује како се она може користити.

# Показивачи и слободна меморија

- Програмер захтева заузимање (алоцирање) новог меморијског парчета у „слободној меморији“ (енгл. free store, heap) путем **new** оператора
  - Оператор **new** враћа адресу заузетог меморијског парчета
  - Та адреса се може сместити у показивач
  - Примери:
    - **int\* p = new int;** // заузми меморију за један **int**  
// **int\*** значи “показивач на **int**”
    - **int\* q = new int[7];** // заузми меморију за 7 **int**-ова  
// “низ од 7 **int**-ова”
    - **double\* pd = new double[n];** // овде се може користити и променљива
  - Показивач може садржати адресу само објекта одређеног типа
  - Показивач не зна колико таквих објекта се налази у низу



# Заузимање меморије: пример

```
double* calc(int result_size, int max)
{
    int* p = new int[max];
    double* result = new double[result_size];
    double* t = new double(7.0);
    return result;
}
```

```
double* r = calc(200, 100);
```

- Ако не ослободимо заузету меморију, она ће бити ослобођена на крају програма.
- Међутим, то неки пут нијеовољно добро, и обично се зове цурење меморије.
- Програм који треба дуго да се извршава обично не може дозволити цурење меморије.

## Заузимање меморије: пример

```
double* calc(int result_size, int max)
{
    int* p = new int[max];
    double* result = new double[result_size];
    double* t = new double(7.0);
    delete[] p;
    delete t;
    return result;
}

double* r = calc(200,100);
```

## Заузимање меморије: пример

```
double* calc(int result_size, int max)
{
    int* p = new int[max];
    double* result = new double[result_size];
    double* t = new double(7.0);
    delete[] p;
    delete t;
    return result;
}

double* r = calc(200,100);

delete[] r; // лако се заборави
```

- Цурење меморије није добро или лоше, већ може бити неповољно у одређеним условима

# Парови оператора за заузимање меморије

- Оператор **new** мора бити упарен са својим одговарајућим **delete!**

```
int* pin = new int[100];  
delete[] pin;
```

```
int* pi = new int;  
delete pi;
```

- Када се briше само један елемент, онда су током компајлирања доступни сви неопходни подаци: колико меморије (величина типа) и на којој адреси (вредност показивача).
- Када показивач показује на адресу низа елемената, онда количина меморије није позната, тј. недостаје информација о броју елемената.
- Та информација се складишти испод хаубе.
- Зато су **new** и **delete** за низове елемената посебна операција.

# Приступ објектима преко показивача



- **Појединачни елементи**

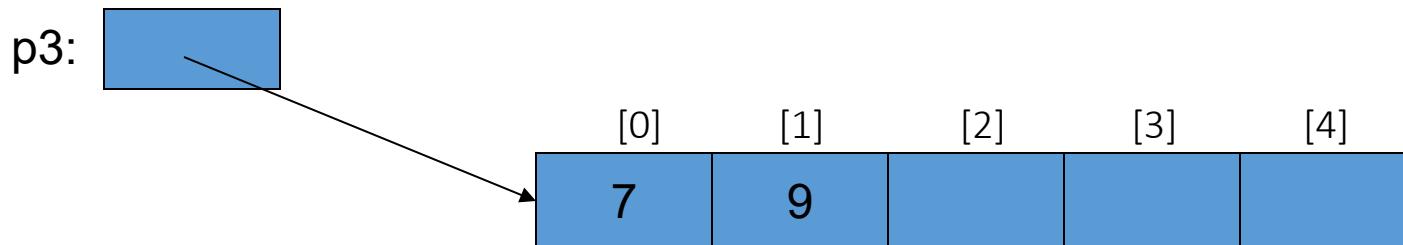
```
int* p1 = new int;
int* p2 = new int(5); // заузми меморију за int и иницијализуј на 5

int x = *p2;           // прочитај вредност на коју p2 показује
                       // у овом случају број 5
int y = *p1;           // чита недефинисану вредност
```

# Вредности показивача

```
// Можете видети вредности показивача, али то вам је ретко када потребно
int* p1 = new int(7);
double* p2 = new double(7);
cout << "p1==" << p1 << " *p1==" << *p1 << "\n"; // p1==??? *p1==7
cout << "p2==" << p2 << " *p2==" << *p2 << "\n"; // p2==??? *p2==7
```

# Приступ објектима преко показивача



- Низови елемената

```
int* p3 = new int[5];
```

```
p3[0] = 7;           // упиши први елемент низа
p3[1] = 9;
```

```
int x2 = p3[1];
```

```
int x3 = *p3;       // *p3 значи p3[0]
```

# Приступ објектима преко показивача

- Показивач не зна број елемената на које показује (само адресу првог елемента)

```
double* p1 = new double;
```

```
*p1 = 7.3;           // ok
```

```
p1[0] = 8.2;         // ok
```

p1:



```
p1[17] = 9.4;        // проблем!
```

```
p1[-4] = 2.4;        // проблем!
```



```
double* p2 = new double[100];
```

```
*p2 = 7.3;           // ok
```

p2:



```
p2[17] = 9.4;         // ok
```

```
p2[-4] = 2.4;         // и даље проблем!
```

```
p2[106] = 6.3;        // проблем!
```

# Приступ објектима преко показивача

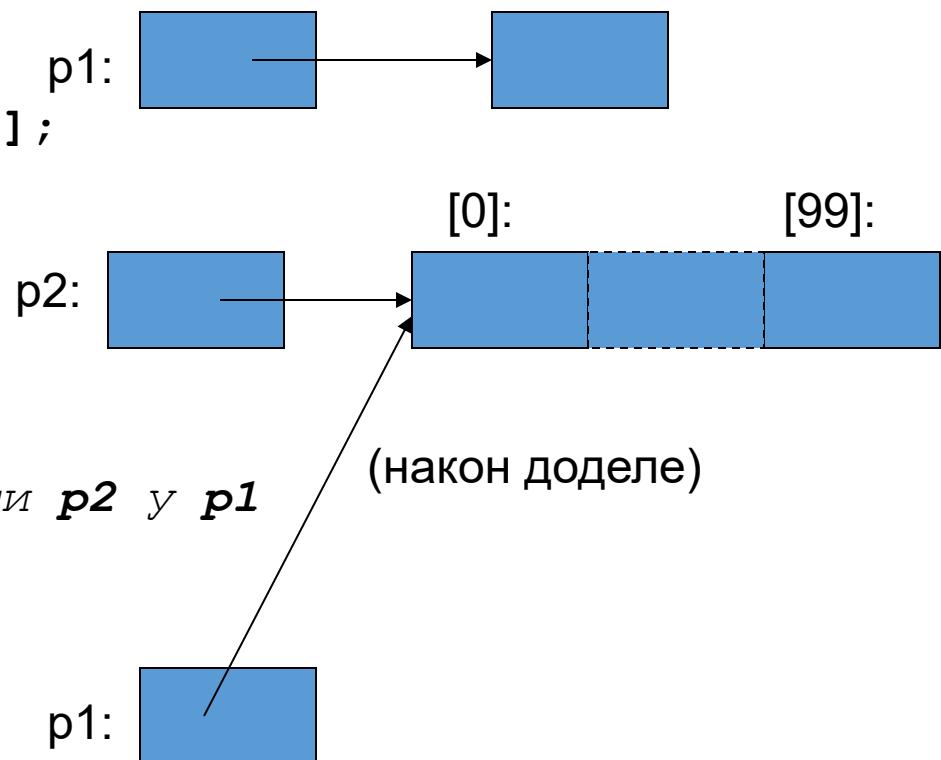
- Показивач не зна број елемената на које показује (само адресу првог елемента)

```
double* p1 = new double;  
double* p2 = new double[100];
```

```
p1[17] = 9.4; // грешка
```

```
p1 = p2; // додела вредности p2 у p1
```

```
p1[17] = 9.4; // сада је у реду
```



# Приступ објектима преко показивача

- Показивач зна тип објекта на који показује

```
int* pi1 = new int(7);
```

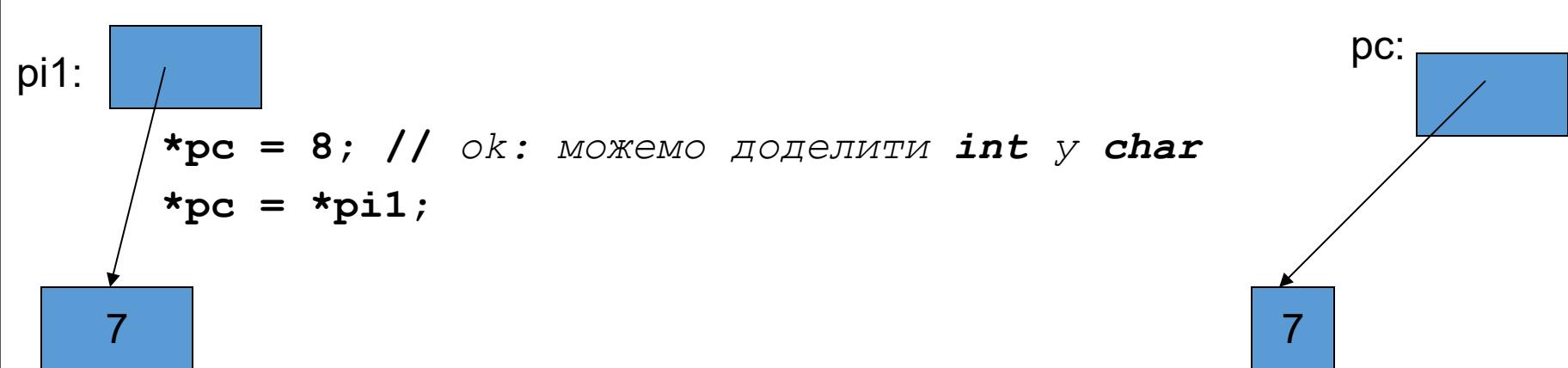
```
int* pi2 = pi1; // ok
```

```
double* pd = pi1; // грешка: не може int* у double*
```

```
char* pc = pi1; // грешка: не може int* у char*
```

- Не постоје имплицитне конверзије између два различита типа показивача

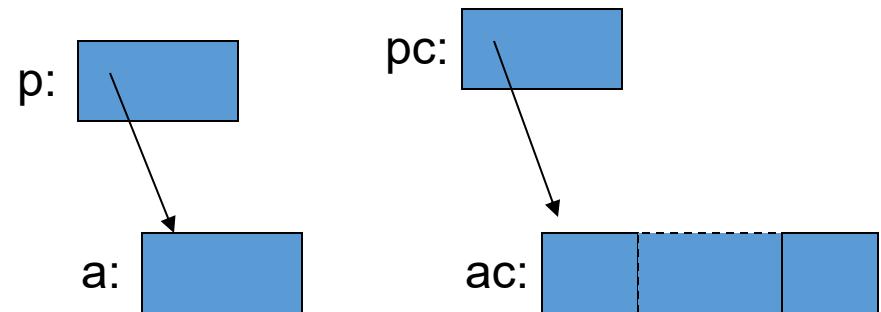
- међутим, постоје имплицитне конверзије између типова на које показују:



# Оператор „адреса“: &

- Можемо добити адресу и променљивих
  - не само објеката алоцираних у слободној меморији

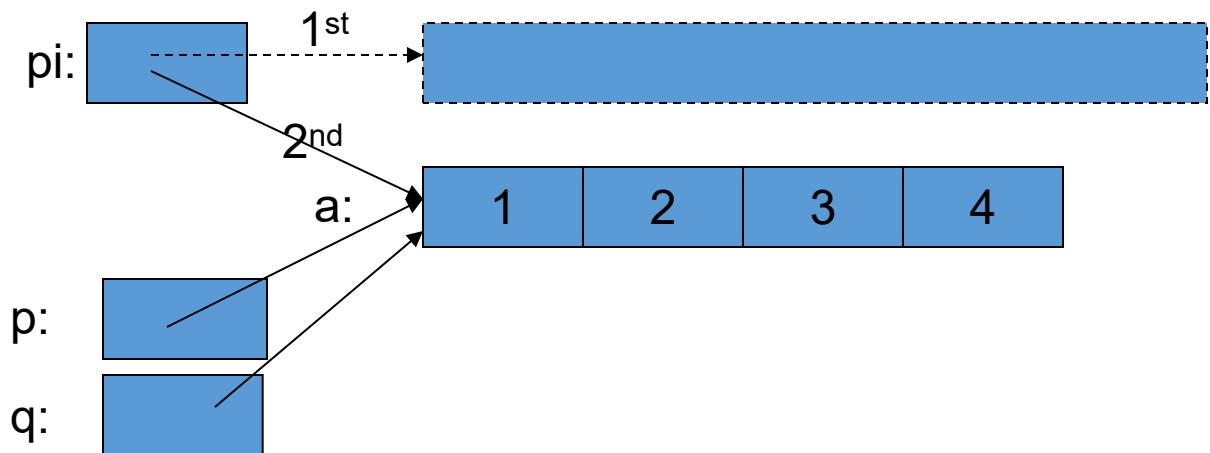
```
int a;  
char ac[20];  
  
void f(int n)  
{  
    int b;  
    int* p = &b;  
    p = &a;  
    char* pc = ac;  
    pc = &ac[0];// исто што и pc = ac  
    pc = &ac[n];// показивац на n+1 елемент (елемент са индексом n)  
                // упозорење: нема провере опсега  
    // ...  
}
```



# Низови се (врло често) конвертују у показиваче

```
void f(int pi[ ])    // исто што и void f(int* pi)
{
    int a[ ] = { 1, 2, 3, 4 };
    int b[ ] = a; // грешка!
    b = pi; // грешка! На име низа се може гледати
              // као на непроменљив показивач (показивачки литерал)
    pi = a; // OK, али не копира садржај:
              // pi сада показује a-ов први елемент

    int* p = a;
    int* q = pi;
}
```



# Опредено са показивачима и низовима

```
char* f()
{
    char ch[20];
    char* p = &ch[90];
    // ...
    *p = 'a';           // шта ће ово преписати?
    char* q;
    *q = 'b';           // шта ће ово преписати?
    return &ch[10];      // ch ће нестати по повратку из f()
                        // (чувени „висећи показивач“)
}

void g()
{
    char* pp = f();
    // ...
    *pp = 'c';           // шта ће ово преписати?
}
```

# Што се уопште бакћемо са низовима?

- Представљају примитивну меморију у Це++ програмима
  - Потребни су нам (обично приликом динамичког заузимања путем new) да би имплементирали контејнере вишег нивоа
- То је све што Це језик има од „контејнера“
  - Рецимо, нема вектора
  - Мнооого кода је написано у Цеу
    - Реда величине милијарди линија
  - Много Це++ кода је написани у Це стилу
    - Реда величине 100 милиона линија
  - Пре или касније ћете наћи на код који користи низове
- Али, не користите их ако баш не морате
  - Они су највећи извор багова у Це и Це++ програмима
  - Једни су од највећих узрока безбедносних пропуста (најчешће кроз прекорачење бафера, тј. заузете меморије)

# Зашто се уопште бакћемо са заузимањем меморије?

- Да би користили променљиву као величину низа
- Да направимо објекте који ће живети дуже од досега у којима су направљени:
  - На пример:

```
double* make(int n)
{
    return new double[n];
}
```

# Показивачи, низови и вектори

- Са показивачима и низовима директно додирујемо хардвер, уз минималну помоћ језика. У таквом режиму врло озбиљне грешке се могу направити, које се касније могу врло тешко открити и исправити.
  - Будите опрезни са овим и користите само када морате и кад тачно знате шта радите
  - Вектор представља један начин којим се добија велика флексибилност и скоро потпуна ефикасност низова, уз много већу подршку језика (што се своди на мање багова и мање дебаговања)

# Резиме заузимања меморије

- Заузимање коришћењем оператора **new**
  - `int* pi = new int;`
  - `char* pc = new char('a');`
  - `double* pd = new double[10];`
- New баца **bad\_alloc** изузетак уколико не може да заузме меморију
- Ослобађање коришћењем **delete** или **delete[ ]**
  - `delete pi; //` ослобађа појединачни објекат
  - `delete pc; //` ослобађа појединачни објекат
  - `delete[] pd; //` ослобађа низ објеката
- Ослобађање нул показивача не ради ништа
  - `char* p = nullptr;`
  - `delete p; //` безопасно

# Животни век (трајност) променљиве

- Четврти одговор: програмер сам одређује када ће променљива постати жива, а рантајм окружење (сакупљач смећа) одређује када ће умрети.
- У Це++-у постоји подршка и за ово, али се ретко користи. (Зашто?)
- Пети одговор: променљива постаје жива при стварању нове нити, а умире при гашењу нити.
- Ово су променљиве нитске трајности, али се тиме нећемо бавити.