

# О курсу

- C++11
  - C++14
  - C++17
  - C++20
- 
- Уведено много разних новина.
  - Циљ овог курса је да укаже на најважније (мада је то већина тих новина).
  - Такође, упознаћемо се и са неким могућностима које су постојале и раније, али и даље спадају напредне технике.

# Одабрана једноставнија проширења Це++ језика

`nullptr`, `auto`, `for` по опсегу, атрибути, `enum` класе,  
`constexpr` (compile-time константе), `static_assert`,  
новине са стринг литералима, `string_view`

# nullptr

- Које вредности показивач може да има?
- Вредност можемо видети:

```
Struct* p = new Struct();  
std::cout << p;
```

- Али нам је то јако ретко потребно (суштински никада, осим за нека озбиљна дебаговања).
- Међутим, потребно нам је да разликујемо два случаја:
  - показивач показује на неки објект
  - показивач не показује ни нашта

# nullptr

- Од свих могућих вредности (адреса) које показивач може да има, потребно је да једна буде специјална – резервисана да значи „ни нашта не показујем“. Та вредност се назива „Нул вредност“.
- Стандард не специфицира која то бројчана вредност треба да буде.
- Да бисмо разлучили између показивача који показује на неки објект и показивача који не показује ни нашта, морамо имати неку ознаку за ту вредност.
- У пракси се за то користио (до C++11) знак: 0.
- У зависности од контекста, 0 се интерпретира као цео број 0, или као вредност показивача ни нашта (на тој конкретној платформи).
- То решење је донело бар два проблема:
  - Логички се меша цео број нула, са Нул вредношћу (која није целобројног типа, и нити мора бити бројчано једнака нули)
  - У одређеним контекстима употреба доводи до вишезначности:  

```
void foo(int x);  
void foo(char* x);  
foo(0); // на шта се овде мисли?
```

# nullptr

- Као полурешење се (опет пре Це++11) наметнула употреба претпроцесорског симбола **NULL**, који је по стандарду дефинисан у заглављу **stddef.h** (односно **<stddef>**).
- Међутим, у Це++-у главни недостатак тог решења произилази из чињенице да конверзија из **void\*** у друге показивачке типове мора бити експлицитна.

```
#define NULL (void*)0
void foo(char* x);
foo(NULL); // грешка, конверзија мора бити експлицитна
```

- У Це++11 је уведена нова резервисана реч да означава Нул вредност: **nullptr**, што коначно представља потпуно решење.
- Употреба знака 0 је и даље могућа, због подршке за постојеће програме, али се њена употреба обесхрабрује.

```
void foo(int x);
void foo(char* x);
foo(nullptr); // сада је ствар јасна
```

# auto

- Којег типа треба да су ове променљиве, да би ове иницијализације биле без имплицитних конверзија и ваљане?

```
[REDACTED] a = 5;
```

```
[REDACTED] b = 5.0;
```

```
[REDACTED] c = 1.0 + 2.0i;
```

```
[REDACTED] d = new Struct();
```

```
[REDACTED] e = v.begin();
```

# auto

- Којег типа треба да су ове променљиве, да би ове иницијализације биле без имплицитних конверзија и ваљане?

```
int                a = 5;
```

```
double            b = 5.0;
```

```
complex<double>    c = 1.0 + 2.0i
```

```
Struct*           d = new Struct();
```

```
std::vector<int>::iterator e = v.begin();
```

# auto

- **auto** кључна реч сада значи да тип променљиве треба да одговара типу иницијализационог израза.

```
auto          a = 5;
```

```
auto          b = 5.0;
```

```
auto          c = 1.0 + 2.0i
```

```
auto          d = new Struct();
```

```
auto          e = v.begin();
```



# auto

- **auto** није нова кључна реч, али има нову сврху.
- Шта је до Це++11 **auto** значило и зашто смо га ретко виђали (практично никада)?
- **auto** кључна реч не значи да се не мора знати тип променљиве, или да се о томе не мора мислити.
- Три главна разлога за употребу auto су: а) у случајевима сложеног типа, б) у имплементацији и употреби шаблона (генеричког програмирања), в) код ламбда функција.
- Међутим, постоје заговорници AAA принципа (Almost Always Auto), али та прича превазилази оквире овог курса.

## for по опсегу

- **for** петља сада има додатан облик:

```
for (vector<int>::iterator it = c.begin(); it != c.end(); ++it) {  
    cout << *it;  
}
```

```
for (int x : c) {  
    cout << x;  
}
```

```
for (vector<int>::iterator it = c.begin(); it != c.end(); ++it) {  
    cin >> *it;  
}
```

```
for (int& x : c) {  
    cin >> x;  
}
```

## for по опсегу

- **for** петља сада има додатан облик:

```
for (vector<Huge>::iterator it = c.begin(); it != c.end(); ++it)
{
    cout << *it;
}
```

```
for (const Huge& x : c) {
    cout << x;
}
```

# for по опсегу

- Ради и за Цеоовски низ

```
int niz[] = {1, 2, 3, 4, 5};
```

```
for (int x : niz) {  
    cout << x;  
}
```

```
for (int x : {1, 2, 3, 4, 5}) {  
    cout << x;  
}
```

## for по опсегу

- Да би радило за корисничке типове (а сви STL контејнери јесу кориснички типови), потребно је да кориснички тип има итераторе и методе `begin()` и `end()`
- Итератор је тип који има дефинисане следеће операције:
  - `==` (и `!=`)
  - `*`
  - `++`
- `begin()` и `end()` враћају итератор на почетак, односно итератор иза краја контејнера.
- али, могу и `begin` и `end` функције:

```
for (auto it = begin(c); it != end(c); ++it) {  
    std::cout << *it;  
}
```

```
for (auto x : c) {  
    std::cout << x;  
}
```

# Атрибути

- Да ли сте некада видели ово у Це/Це++ коду који се преводи GCC-ом?

```
__attribute__((НЕШТО))
```

- Или ово:

```
#pragma НЕШТО
```

- Модерни Це++ нуди нову синтаксу која би требало да буде стандардизована замена за `__attribute__` (и друга компајлерска проширења те намене) и алтернатива претпроцесорској конструкцији `#pragma`

- Овако изгледа стандардни Це++ атрибут:

```
[[НЕШТО]]
```

- И може да стоји уз типове, променљиве, функције, блокове кода и наредбе, па чак и да важи за целе јединице превођења.

# Атрибути

- Стандард дефинише следеће атрибуте:

```
[[noreturn]]  
[[deprecated]] [[deprecated("razlog")]]  
[[fallthrough]]  
[[nodiscard]]  
[[maybe_unused]]  
[[carries_dependency]]
```

- Стандард, такође, прописује и ово правило, да би подржао атрибуте који су специфични за платформу: Ако компајлер не препозна неки атрибут, треба да га игнорише (ни упозорење не треба да се пријави).
- Атрибути могу имати име као да су дефинисани у именском простору (нпр. **[[gnu::always\_inline]]**, **[[clang::availability]]**). То је обично случај са нестандартним атрибутима, специфичним за одређени компајлер или платформу.

# Атрибути

- Када функција неће вратити контролу позивајућој функцији.

```
[[noreturn]] void foo() {  
    throw "error";  
}
```

```
[[noreturn]] void bar() {  
    while (true);  
}
```

- Помаже компајлеру да оптимизује у неким случајевима, плус елиминише нека упозорења које би можда била генерисана.
- Неколико функција у стандардној библиотеци је овако декларисано.



# Атрибути

- Када је неки елемент програма/библиотеке застарео и требало би га полако напустити.

```
[[deprecated ("Now use >> operator")]]  
void readFile(const char* name);
```

```
[[deprecated]]  
class SomeClass;
```

```
[[deprecated]] namespace Djuradj {  
...  
}
```

- А може да стоји и уз енуме, дефиниције типова, променљиве...

# Атрибути

- Може и да се назначи да је пропадање кроз другу case лабелу намерно:

```
switch (x) {  
case 1:  
    a();  
    break;  
case 2:  
    b();  
    [[fallthrough]]  
case 3:  
    c();  
}
```

# Атрибути

- Код неких функција желимо да осигурамо да ће повратна вредност бити уважена:

```
[[nodiscard]] bool sanityCheck();
```

```
void foo() {  
    sanityCheck(); // Грешка!  
    // ...  
}
```

- А некада је корисно назначити да се нека променљива можда неће увек користити (па да компајлер то не пријављује као упозорење):

```
void foo() {  
    [[maybe_unused]] bool sanityOK = sanityCheck();  
    assert(sanityOK); // неће бити присутно у рилис билду  
    // ...  
}
```

# Енум класе

- Постоје три проблемчића са досадашњим набројивим типовима:

1. Дефинисани симболи упадају у тренутни досег.

```
enum PeriniDrugari { SIMA, DJURA, STEVA };  
enum MikiniDrugari { DJOLE, SIMA, MILE };  
SIMA // на шта се мисли овде?
```

2. Целобројни тип на који се енум своди је врло лабаво дефинисан

- Па се на то не може ослонити, а последице су да није поуздано радити аритметику са енум вредностима, нити је могуће само декларисање енума унапред.

3. Енум вредности се имплицитно конвертују у цео број.

- У комбинацији са претходним олакшава да се случајно направи грешка.

# Енум класе

- Ти проблеми су се до сва овако решавали:

## 1. Дефинисани симболи упадају у тренутни досег.

```
enum PeriniDrugari { PD_SIMA, PD_DJURA, PD_STEVA };  
enum MikiniDrugari { MD_DJOLE, MD_SIMA, MD_MILE };  
PD_SIMA // Сада је јасно.
```

## 2. Целобројни тип на који се енум своди је врло лабаво дефинисан

- Треба пажљиво писати код да се на то ни не ослања. Укратко: користити енуме само за складиштење вредности и поређење и то само са симболима из истог тог енума.

## 3. Енум вредности се имплицитно конвертују у цео број.

- Просто пазити мало више. Неки компајлери пријављују упозорење.

# Енум класе

- А сада имамо „набројиву класу“:

## 1. Прави свој именски простор.

```
enum class PeriniDrugari { SIMA, DJURA, STEVA };  
enum struct MikiniDrugari { DJOLE, SIMA, MILE };  
PeriniDrugari::SIMA
```

## 2. Своди се увек на int, осим ако експлицитно није наведено другачије.

```
enum class Primer1 { A, B, V }; // своди увек на int  
enum class Primer2 : long { X, Y, Z }; // сада на long  
enum class Primer3;  
...  
enum class Primer3 { P, Q, R }
```

## 3. Вредности из енум класе се не конвертују имплицитно у цео број.

- А ако треба, могу се експлицитно конвертовати помоћу static\_cast.

# const

- const у Це++-у, у суштини има значење „само за читање“ („read-only“)

```
int g_a;
```

```
void foo() {  
    const int k = g_a;  
    k += 1; // грешка у превођењу  
    g_a += 1; // ОК  
}
```

```
void bar(const int& x);
```

```
const volatile int* x = FLAG_ADDRESS;
```

# Компајлирање насупрот интерпретацији

- Код компајлирања фаза превођења је временски одвојена од фазе извршавања.



- Код интерпретирања, те две фазе су временски испреплетене.



# const

- `const` говори компајлеру да током превођења пријави као грешку наредбе које би током извршавања измениле (или могле да измене) то што је обележено као `const`. Због тога још каже и да је то „константно током извршавања“ (енгл. „run-time constant“).



- Компајлер у општем случају не зна која ће то конкретно вредност бити (нити га то занима, осим у неколико специфичних случајева када покушава да оптимизује код), једино му је битно да се само једном иницијализује (на почетку дела где се користи) и да се после не мења током извршавања.

# const

- Ово су све примери где компајлер никако не може знати вредност током превођења.

```
int g_a;
```

```
void foo() {  
    const int k = g_a;  
    k += 1; // грешка у превођењу  
    g_a += 1; // OK  
}
```

```
void bar(const int& x);
```

```
const volatile int* x = FLAG_ADDRESS;
```

# const

- Али, ово су примери где може:

```
const int g_a = 5;
```

```
void foo() {  
    const int k = g_a;  
    const int j = 7;  
    std::cout << k << j << std::endl;  
}
```

- И компајлер ће у већини случајева горњи код свести на доњи:

```
const int g_a = 5;
```

```
void foo() {  
    std::cout << 5 << 7 << std::endl;  
}
```

- У овим случајевима можемо говорити о „константама током превођења“, али то је са становишта језика само питање оптимизације.

# Компајлирање насупрот интерпретацији

- Јасно нам је по чему је компајлер различит од интерпретера, али: И компајлер током превођења ради интерпретацију у одређеним случајевима.

```
int foo() {  
    int x = 5 * 6 + 2;  
    return x;  
}
```

- Нпр. да ли очекујемо да ће генерисани код личити на ово лево или ово десно?

```
int foo() {  
    int x = 32;  
    return x;  
}
```

```
int foo() {  
    return 32;  
}
```

Или чак ово, у  
недебаг режиму.

```
int foo() {  
    int t1 = 5;  
    int t2 = 6;  
    int t3 = t1 * t2;  
    int x = t3 + 2;  
    return x;  
}
```

# Компајлирање наспрот интерпретацији

- Шта више, компајлер би могао и ово да уради (и често то ради):

```
int foo() {  
    int x = 5 * 6 + 2;  
    return x;  
}  
int main() {  
    ...  
    int a = foo();  
    ...  
}
```



```
int main() {  
    ...  
    int a = 32;  
    ...  
}
```

```
int foo(int x) {  
    return x * 6 + 2;  
}
```



```
int main() {  
    ...  
    int a = foo(6);  
    ...  
}
```

```
int main() {  
    ...  
    int a = 38;  
    ...  
}
```

# Компајлирање наспрот интерпретацији

- А ово?

```
int foo(int x) {  
    if (x > 7)  
        return x * 3 - 2;  
    else  
        return x * 2 - 3;  
}
```



```
int a = ??;  
int b = ??;
```

```
int a = foo(6);  
int b = foo(8);
```

```
int foo(int x) {  
    int s = 1;  
    for (int i = 0; i < x; ++i)  
        s *= x;  
    return s;  
}
```



```
int a = ??;  
int b = ??;
```

```
int a = foo(6);  
int b = foo(8);
```

# Компајлирање наспрот интерпретацији

- И то... зашто да не?

```
int foo(int x) {  
    if (x > 7)  
        return x * 3 - 2;  
    else  
        return x * 2 - 3;  
}
```



```
int a = 9;  
int b = 22;
```

```
int a = foo(6);  
int b = foo(8);
```

```
int foo(int x) {  
    int s = 1;  
    for (int i = 0; i < x; ++i)  
        s *= x;  
    return s;  
}
```



```
int a = 720;  
int b = 40'320;
```

```
int a = foo(6);  
int b = foo(8);
```

# constexpr

- Сада постоји нова реч која означава да нешто баш треба да буде „константно током превођења“ (енгл. „compile-time constant“).
- Дакле, компајлер тада мора знати вредност тога током превођења.



```
constexpr int g_a = 5;
```

```
void foo() {  
    constexpr int k = g_a;  
    constexpr int j = 7;  
    std::cout << k << j << std::endl;  
}
```



# constexpr

- Компајлер ће се сада бунити ако вредност не може да зна током превођења.

```
int g_a;
```

```
void foo() {  
    constexpr int k = g_a; // грешка у превођењу  
}
```

```
void bar(constexpr int& x); // грешка у превођењу
```

```
constexpr volatile int* x = FLAG_ADDRESS; // грешка у превођењу
```

- Где год нам је намера да буде константа током превођења, треба да употребимо constexpr. Јасније изражавамо намеру, компајлер проверава да ли постигли то што желимо, а уједно и осигуравамо оптимизацију по том питању.

# constexpr

- Ако употребимо constexpr, онда је срачунавање тог израза током превођења обавезно.

```
int foo() {  
    constexpr int x = 5 * 6 + 2;  
    return x;  
}
```

- Сад можемо приметити и да је цела повратна вредност функције константна током превођења.

```
constexpr int foo() {  
    return 5 * 6 + 2;  
}
```

```
constexpr int a = foo(); // Скроз ОК. Као да је a = 32;
```

# constexpr функције

- constexpr функције могу и да примају параметре.

```
constexpr int foo(int x) {  
    return 5 * 6 + x;  
}
```

```
constexpr a = foo(2); // Скроз ОК. Као да је a = 32;  
constexpr b = foo(a); // b = 62; (5 * 6 + 32)
```

- Али, може и ово:

```
void bar(int y) {  
    int a = foo(y); // И даље у реду, али неће бити срачунато  
    // током превођења, већ ће бити регуларан позив функције  
}
```

## constexpr функције

- Дакле, constexpr функције морају бити срачунљиве током превођења (константне током превођења) ако су им стварни параметри срачунљиви током превођења. У супротном, биће генерисан код за њих као за редовне функције.
- Међутим, constexpr функције ће бити заиста срачунате током превођења, само ако је њихов резултат употребљен у constexpr контексту.

```
constexpr int a = foo(5); // Срачунато током превођења
int b = foo(5); // Биће позвана редовна функција
void bar(int y) {
    constexpr int c = foo(y); // Грешка у превођењу
}
```

- У C++20 је најављена реч constexpr, да означаи функције које увек морају да се срачунају током превођења. (Па би за такве функције друга линија у горњем коду ипак проузроковала грешку у превођењу)

## constexpr функције

- Стандард прописује да constexpr функција мора поштовати одређена ограничења да би се осигурала њена срачунљивост током превођења (у супротном, компајлер је неће прихватити као constexpr функцију).
- Отприлике, њен код мора бити такав да нема споредних (бочних) ефеката (утицаја на нешто ван функције).
- Конкретније, у телу функције се смеју користити сви искази и изрази осим:
  - асемблерског исказа
  - goto
  - лабела (осим case и default)
  - try блок
  - позива функција које нису constexpr
  - дефиниција променљивих без иницијализације
  - дефиниција променљивих статичке трајности (или нитске трајности – о томе касније)
  - дефиниција променљивих нелитералског типа (видећемо касније, али укратко: у обзир долазе само једноставни, плитки типови, са тривијалним деструктором и constexpr конструктором; нпр. std::string не може)

## constexpr функције

- Између C++11 и C++14 стандарда, било је рестриктивније.
- constexpr функција се могла састојати само од једне return наредбе.
- За if се може користити ? :
- И рекурзијом се може опонашати петља.

# constexpr променљиве

- constexpr променљива мора бити литералског типа.
- Литералски тип је дефинисан отприлике овако:
  - void, или
  - Скаларни тип (тзв. основни аритметички типови, показивачи итд.), или
  - Референца, или
  - Низ литералских типова, или
  - Сложени тип (класа) са следећим особинама:
    - поседује тривијални деструктор
    - има бар један constexpr конструктор (или је агрегатног типа – отприлике: има само подразумевани конструктор и сви атрибути су јавни)
    - сви нестатички атрибути су литералског типа
    - сви преци у наслеђивању су литералског типа
- Неформално речено, то су једноставни, плитки типови, који могу бити конструисани (иницијализовани) приликом интерпретације од стране преводиоца, и са којима се рачун може обавити током превођења.

# constexpr променљиве

- Иницијализациони израз за constexpr променљиве може да се састоји од следећих елемената:
  - литерала
  - других constexpr променљивих
  - позива constexpr функција (операције над основним типовима су за потребе ове дефиниције constexpr функције)
  - и const променљивих целобројног или набројивог (енум) типа које су по својој природи константе током превођења.
- Ова последња ставка омогућава да се constexpr дода у стари код, без темељног преправљања.

```
const int g_a = 5;  
const double g_b = 5.0;  
constexpr double g_c = 6.0;
```

```
void bar() {  
    constexpr int k = g_a;  
    constexpr double j = g_b; // ово не може  
    constexpr double i = g_c; // али ово може  
}
```



# Статичке тврдње

## static\_assert

- Још од Цеа имамо механизам тврдњи (assert), дефинисан у заглављу assert.h (#include <cassert> у Це++-у)

```
#include <cassert>
// Прима парне бројеве. У супротном – недефинисано стање.
int foo(int x) {
    assert(x % 2 == 0);
    ...
}
```

- У дебаг билду на месту тврдње наћи ће се код који је проверава током извршавања. Уколико тврдња није задовољена, извршавање програма ће се прекинути уз пријаву грешке.
- У рилис билду тврдње нестају и током извршавања се не обавља никаква провера.

# Статичке тврдње

## static\_assert

- У Це++11 је уведен механизам за статичке тврдње (static\_assert).
- То су тврдње које могу бити проверене током превођења.

```
static_assert(sizeof(int) * CHAR_BIT == 32,  
    "int has to be 32 bits for this code to work");  
// а може и без поруке о грешци:  
static_assert(sizeof(int) * CHAR_BIT == 32);  
// биће генерисана нека општа порука
```

```
constexpr long long freq = 48'000'000;  
constexpr int block_size = 50'000;  
static_assert(block_size > (freq / 1000),  
    "Block has to cover more than 1ms")
```

- Први параметар статичке тврдње мора бити израз срачунљив током превођења.

# Статичке тврдње

## `static_assert`

- Израз у статичкој тврдњи се током превођења срачунава само једном, у једном контексту.
- Зато статичке тврдње не можемо искористити за проверу улазних параметара ни у обичним, али ни у `constexpr` функцијама.

```
// Прима парне бројеве. У супротном – недефинисано стање.  
constexpr int foo(int x) {  
    static_assert(x % 2 == 0); // ГРЕШКА!  
    // израз није срачунљив, због x  
    // које зависи од позива до позива, а static_assert се  
    // срачунава само једном  
    ...  
}
```

# Статичке тврдње

## static\_assert

- Али, обичне тврдње сасвим лепо раде у constexpr функцијама:

```
#include <cassert>
// Прима парне бројеве. У супротном – недефинисано стање.
constexpr int foo(int x) {
    assert(x % 2 == 0);
    ...
}
```

```
constexpr int a = foo(4); // ОК
constexpr int b = foo(5); // Грешка током превођења
```

# Статичке тврдње

## static\_assert

- Такође, constexpr функција може да баци изузетак:

```
#include <cassert>
// Прима парне бројеве. У супротном – недефинисано стање.
constexpr int foo(int x) {
    if (x % 2 != 0) throw std::logic_error("nije parno");
    ...
}
```

```
constexpr int a = foo(4); // ОК
constexpr int b = foo(5); // Грешка током превођења
// а сада може и ово:
void bar(int v) {
    try {
        int r = foo(v); // преводи се као регуларна функција
    }
    catch (...) { // ... }
}
```

# Новине са стринг литералима

- Шта је тип овог израза:

`"...dје је vinjak na Letenci?"`

# Новине са стринг литералима

- Шта је тип овог израза:

`"...dје је vinjak na Letenci?"`

- До недавно је био **char\***
- Зашто не **const char\***?
- Из историјских разлога: прве верзије Цеа нису имале **const** кључну реч али јесу имале стринг литерале, и ето...
- Било како било, сад је то исправљено и тип горњег израза је сада **const char\***, као што је и требало да буде. И очекивано је да компајлер пријави грешку уколико то ваш код не уважава.
- (Свакако је доводило до недефинисаног стања писање у меморију која је додељена стринг литералу)

# Новине са стринг литералима

- А шта је ово?

```
"...dје је vinjak na Letenci?"
```

```
L"...dје је vinjak na Letenci?"
```

```
u8"...dје је vinjak na Letenci?"
```

```
u"...dје је vinjak na Letenci?"
```

```
U"...dје је vinjak na Letenci?"
```

- Префикс одређује који начин кодовања се користи за знакове. `char`, `wchar_t`, UTF-8, UTF-16, UTF-32.
- Самим тим, тип литерала је мало другачији: `const char*`, `const wchar_t*`, `const char*`, `const char16_t*`, `const char32_t*`.

- А шта ако хоћу литерал типа `std::string`?

```
using namespace std::literals; // или std::string_literals  
"...dје је vinjak na Letenci?"s
```



# Новине са стринг литералима

- Ако желимо од оваког текста да направимо стринг литерал:

`U nazivu datoteke ne smeju da se koriste "\" i "'"`

- морали би да пишемо овако:

`"U nazivu datoteke ne smeju da se koriste "\"\\" i "\"'\\""`

- а сада можемо и овако:

`R"(U nazivu datoteke ne smeju da se koriste "\" i "'"")"`

- А ако баш желимо можемо и ово да имамо у стрингу: `)"`

`R"a(U nazivu datoteke ne moze ni ovo da stoji: ")")a"`

- На основу тог **a** између `"` и заграде, зна се шта је заправо крај.

## string\_view

```
void printCommaInNewLine(string s)
{
    if (s.empty()) return;

    string::size_type x = s.find(",");
    while (x != string::npos)
    {
        cout << s.substr(0, x) << endl;
        s = s.substr(x + 1);
        x = s.find(",");
    }
    cout << s << endl;
}
```

Шта се овде све дешава испод хаубе?

# string\_view

```
void printCommaInNewLine(string s)
{
    if (s.empty()) return;

    string::size_type x = s.find(",");
    while (x != string::npos)
    {
        cout << s.substr(0, x) << endl;
        s = s.substr(x + 1);
        x = s.find(",");
    }
    cout << s << endl;
}
```

Овде се заузима  
нова меморија.

# string\_view

```
#include <string_view>

void printCommaInNewLine(string ss)
{
    if (ss.empty()) return;

    string_view s(ss);

    string::size_type x = s.find(",");
    while (x != string::npos)
    {
        cout << s.substr(0, x) << endl;
        s = s.substr(x + 1);
        x = s.find(",");
    }
    cout << s << endl;
}
```

**string\_view** се суштински своди на показивач (на почетак низа) и количину

# string\_view

```
#include <string_view>
void printCommaInNewLine(string_view s)
{
    if (s.empty()) return;

    string::size_type x = s.find(",");
    while (x != string::npos)
    {
        cout << s.substr(0, x) << endl;
        s = s.substr(x + 1);
        x = s.find(",");
    }
    cout << s << endl;
}

printCommaInNewLine(someString);
printCommaInNewLine("asda,asddas,dasda,das"s);
printCommaInNewLine("asda,asddas,dasda,das");
```

# string\_view

```
#include <string_view>
void printCommaInNewLine(string_view s)
{
    if (s.empty()) return;

    string::size_type x = s.find(",");
    while (x != string::npos)
    {
        cout << s.substr(0, x) << endl;
        s.remove_prefix(x + 1);
        x = s.find(",");
    }
    cout << s << endl;
}

printCommaInNewLine(someString);
printCommaInNewLine("asda,asddas,dasda,das"s);
printCommaInNewLine("asda,asddas,dasda,das");
```

## string\_view

```
string_view s1("string literal");
```

```
string_view s2("string literal"s); // грешка!
```

## string\_view

```
string_view s1("string literal");
```

```
string_view s2(string("string literal")); // грешка!
```